Christian Rachmaninoff
Unnikrishnan
CS6963

Project Final Report

Problem:

We set out to reproduce the Spark and Hadoop benchmarks from "Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing". Although several potential bottlenecks exist within the MapReduce framework, we chose to explore the performance of java object serialization as our primary focus. As recommended we looked into Kay Osterhout's "Making Sense of Performance in Data Analytics Frameworks", where they had determined that reading from disk was the primary bottleneck in Spark, not network. Additionally they attribute most of Spark's speedup compared to MapReduce to eliminating repeated serialization of objects during iterative algorithms.

We also wanted to see how performance of both frameworks would be affected using a more sophisticated serialization method, such as Kryo, a more efficient object graph serialization framework in Java. After reviewing [1], we discovered that Spark provides three options for storage of persistent RDDs: in-memory storage as deserialized Java objects, in-memory storage as serialized data, and on-disk storage. We believed that experimenting with these persistence parameters could give us more insight into the true costs of object serialization in both MapReduce and Spark. We hypothesize that the unserialized RDDS in memory, would provide the best performance, followed by in-memory serialized RDDS and then on-disk RDDS.

Our initial plan to test these hypotheses was to measure the iteration times of the following jobs:

1. Spark, in-memory storage, with default settings
2. Spark, mo persistence whatsoever, RDDS are recomputed every time
3. In-memory RDDS, with java serialization
4. In-memory RDDS, with kryo serialization
5. On-disk RDDs, with java Serialization
6. On-disk RDDs with kryo Serialization
7. Mapreduce with on disk text file
8. Mapreduce with in memory text file
9. Mapreduce, with on disk binary file
10. MapReduce with in memory binary file

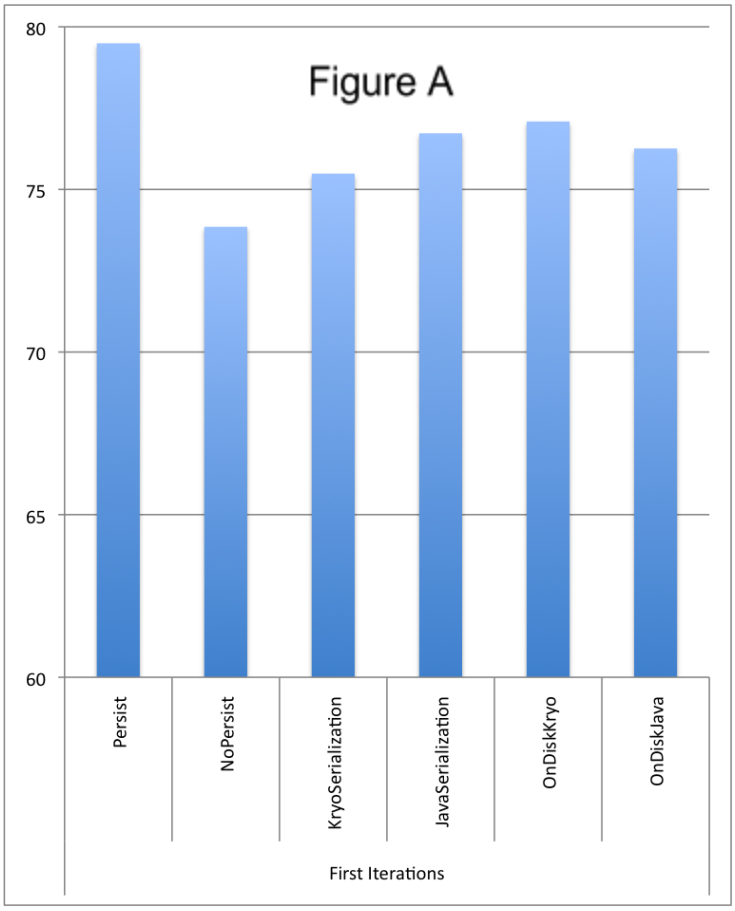
Similar to [1], we planned to test both Logistic Regression and PageRank, using the above methodology. Since are both iterative algorithms, we expected to see significant benefits from Spark's RDD abstraction.

Experiment Setup:

Our setup consisted of 4 OpenStack VMs running Ubuntu 14.04, 3 of which were configured as Hadoop slave nodes, and one node acting as both the ResourceManager and NameNode. We installed Spark on top of this cluster, so that we could access both YARN, the Hadoop ResourceManager, and HDFS. The VMs were created on 4 CloudLab compute nodes. Each VM was allocated 80 GB of Storage, 8GB RAM, and 4 virtual CPUs. For each benchmark trial, all input data and jar files were loaded from HDFS, and iteration times were printed to stdout on the Spark driver. We used the default settings for HDFS.

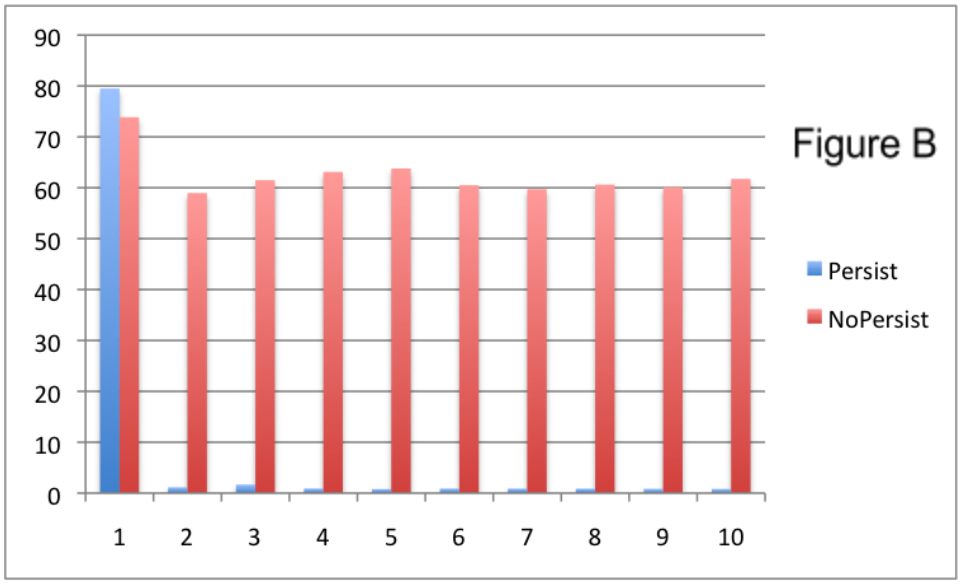
Results:

Prior to executing each job, we clear the page cache and swap space on all nodes to ensure that no data is cached between executions.

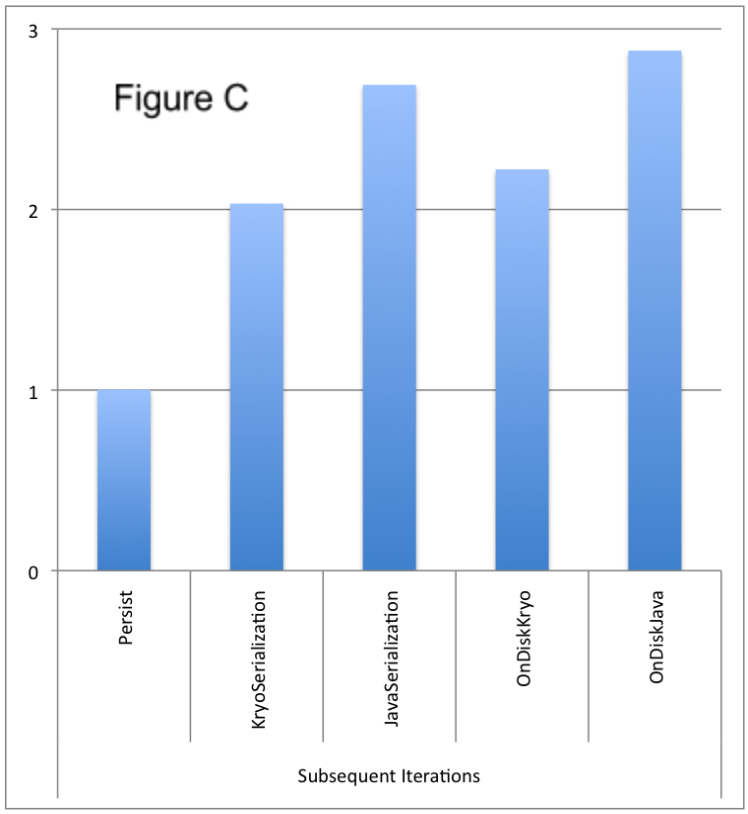


For our first experiment we wrote a scala program that trains a Logistic Regression classifier and outputs the first ten iteration times of the algorithm. The program reads in a text file, and parses data into an RDD. The choice of persistence level at this point in the execution was the main parameter that we varied between jobs. We have plotted the first iteration times for each job in Figure A. Each job was run 3 times, we then take the average of these values. Between the first iteration times, no significant differences were observed, although it was apparent that persisting the data in a standard RDD resulted in the longest iteration time. Additionally, the job that did not persist the data, had the shortest first iteration time. But, as clearly demonstrated in Figure B, this mild benefit is clearly outweighed by the enormous

performance gains from persisting the data. Finally, we have plotted the iteration times of the subsequent iterations for the logistic regression algorithm which involves each serialization Strategy.



Surprisingly, the in-memory serialization was not significantly faster than on-disk persistence for both java and kryo serialization. One possible explanation for this behavior, could be that the latency of the disk read is being dwarfed by the cost of deserializing the object as is it is read into memory. One other thing to note is that all of the serialization strategies resulted in execution times that were at least twice as long as default persistence with no serialization.



Applications of course Material:

Much of the motivation for this work came from the spark and map-reduce papers discussed in the class. While it was interesting to reason about the design and theoretical aspects of these systems in class, actually deploying a cluster and writing jobs gave us much more insight into how these actually functioned, and the type of workloads that these systems are best suited for. The initial experiment to be performed required a cluster with 25-100 nodes but because of resource constraints in the cloudlab, we had to restrict the setup to 4 nodes.

Obstacles:

A large portion of our initial work consisted of familiarizing ourselves with all the tools that we used throughout the project. We spent several days attempting to configure OpenStack Sahara, in order to streamline the job submission and cluster creation that would be required later on in the project. Unfortunately, we were unable to successfully start a cluster using Sahara's UI, and subsequently chose to install hadoop and spark manually onto each node in the cluster. The other problem we found was when our project was swapped out of the cloudlab we had to reinstall spark and hadoop over on each of the nodes in the cluster, so we decided to streamline this process by scripting a significant portion of the hadoop and spark installation process.

Surprisingly, configuring and submitting jobs was much more straightforward on Spark compared to that of Hadoop. As a result, we were unable to successfully run jobs in cluster mode on hadoop even after countless hours of trial and error with different configurations. At the moment, jobs are are being submitted successfully but the execution thread gets stuck indefinitely about halfway through the computation due to some type of connection issue. Although it was very frustrating at times, going through all of the configuration and troubleshooting these issues forced us to really familiarize ourselves with these systems.

Future Work:

In terms of immediate goals, we would like to successfully complete hadoop jobs in cluster mode. If we are are able to get hadoop working correctly, we also plan to evaluate the performance of Apache ignite(in memory file system) on top of hadoop. If we are still unable to run jobs in cluster mode, we plan to run both pagerank and logistic regression on single nodes for both systems in order to get an idea of the relative performance between the two.

One of our long term goals would to experiment with different resource managers for each system. During our experiments we noted that YARN, hadoop's resource manager could also played significant role in execution times. We ran spark jobs on both YARN and Spark standalone clusters, and the Spark execution times were almost 4 times faster in our experiments. Given this large speedup, we believe that one of the major factors contributing to performance in [1] were due to Spark's improved resource management system, as represented in the Figure 7 of the Spark paper. We chose not to fully explore this problem, given our limited time budget and also the obstacles we encountered during configuration of YARN.

Other interesting topics we ran across during our project include query engines built on top of Spark and Hadoop such as Impala and Shark. These query engines in combination with columnar file storage formats such as Parquet and Avro allow these systems to rival traditional parallel dbms in terms of performance by allowing them to index keys and avoid full table scans. The issue of the full table scans was one of the early shortcomings of the Hadoop ecosystem, and was explored more thoroughly in "A Comparison of Approaches to Large-Scale Data Analysis" [3]. With more time, we would have liked to experiment these systems given that a

large portion of Hadoop and Spark workloads have moved toward these scripting and SQL like queries rather than traditional jobs.

[1]: http://www-bcf.usc.edu/~minlanyu/teach/csci599-fall12/papers/nsdi_spark.pdf
[2]: https://people.eecs.berkeley.edu/~keo/publications/nsdi15-final147.pdf
[3]: http://database.cs.brown.edu/projects/mapreduce-vs-dbms/