Super Computing for Big Data - Assignment 1

Chia-Lun Yeh (4718836) and Shikhar Dev (4773071)

September 23, 2018

1 Implementation

1.1 Resilient Distributed Dataset (RDD)

The approach is to shuffle as less as possible. We ended up shuffling twice, the first time to group by the key-value pair with key being (date, word) and the second time to group by date. The outline of the implementation is as follows:

- Stage 1: Read in all the files in the directory into one RDD. Split each file by tab, filter out incomplete data, and extract the date and words.
- Stage 2: Use *reduceByKey* to count all the (date, word) pair occurrences.
- Stage 3: Use *groupByKey* to group all the (word, count) by date. Sort each date by count and return the top 10 words and their counts.

1.2 Dataset

We followed the same approach as in RDD, while implementing the solution using Dataset APIs. The program incorporates two shuffle operations - 1 for grouping multiple rows of a day into one Dataset, and the second for grouping words, to count their frequencies. The algorithm followed by the program is as follows.

- Step 1: Read all CSVs into one Dataset, and map it with schema. Discard columns that were not of interest for the computation. Filter empty cells and cells that had erroneous values ("Type ParentCategory" and "CategoryType ParentCategory").
- Step 2: Parse the "timeline" and "allTopics" columns to extract relevant information and group in (Date - [Topics]) format.
- Step 3: Group topics and organize in format (Date - [(Topic, count), (Topic, count), ...]). Sort this group in descending order and return the top 10 values.

2 Questions

2.1 General

1. **In typical use, what kind of operation would be more expensive, a narrow dependency or a wide dependency? Why?**

   Typically, wide dependency transformations are more expensive than narrow dependency transformations.

   A transformation has narrow dependency when the parent RDD is used by at most one partition of the child RDD. On the other hand, a transformation has wide dependency when the parent RDD is used by multiple child RDDs.

   Since the narrow transformations are only dependent on one parent RDD, they don't typically involve shuffling, or any network transactions. Wide transformations, being dependent on multiple elements from the same or different RDD typically involve shuffling, which is an expensive operation. Also, narrow transformations can leverage optimized pipe-lining to make operations faster. Wide transformation loose out on this optimization, due to its dependency, making it more expensive.

   This speed trajectory also follows for fault tolerance mechanism. Transformations with narrow dependencies are faster than wide dependency transformations, since we essentially need to re-trace the lineage graph from the DAG and re-perform operations leading to the failed operation [4].

2. **What is the shuffle operation and why is it such an important topic in Spark optimization?**

   Shuffling is redistributing data across partitions. Since data are stored in multiple nodes, when we do operations such as **groupByKey**, the data needs to be moved from one node to another to be grouped. It is important to Spark optimization because shuffling can entail network communication and is thus an expensive operation. Therefore, one should do all the transformations and actions that are possible on the data before shuffling them to minimize the data that are transferred thorough the network [2].

3. **In what way can Dataframes and Datasets improve performance both in compute, but also in the distributing of data compared to RDDs? Will Dataframes and Datasets always perform better than RDDs?**

   Performance improvements is introduced in Dataframes and Datasets, primarily via use of **Catalyst** for query optimization and **Tungsten** for optimized memory efficiency.

   Catalyst is an extensible query optimizer abstracted under the Dataframe- Dataset APIs. It turns relational code into highly-optimized RDDs that are run on regular Spark. Since the schema of the data is provided as well as the computation that a user wants to do, Catalyst can optimize by reordering operations, reduce the data that needs to be sent through the network, and skip partitions that are not needed in the computation [4].

   Tungsten optimizes memory management by leveraging application semantics to manage memory explicitly and discarding the overhead of JVM object model / garbage collection. Tungsten also improves storage and retrieval mechanism through a more effective use of CPU cache memory.

   These optimization however are available only for Dataframes and Datasets. So, even if optimized scripts for performing an operation is written in RDDs, Dataframes/Datasets would still outperform RDD equivalent scripts.

   Additionally, unlike in RDD, Dataframe/Dataset organizes data in a relational tabular format, allowing applications to impose structure even in large set of distributed data. This, coupled with abstracted high-level APIs allow developers to focus on solving problems rather than worrying about implementation details.

In essence, it is nearly always preferred to develop applications using Dataframe or Dataset.

4. **Consider the following scenario. You are running a Spark program on a big data cluster with 10 worker nodes and a single master node. One of the worker nodes fails. In what way does Sparks programming model help you recover the lost work?**

   Spark creates a Direct Acyclic Graph (DAG) when a RDD is created, and each transformation is an update on the DAG. Since a RDD is partitioned and resides in multiple nodes, several nodes would have the same DAG. When a worker is lost, the cluster manager can assign another node to continue processing. Or the node can retrieve the DAG from other nodes after it is once again alive and rebuild the graph to continue the process.

5. **Can you think of a problem/computation that does not fit Sparks MapReduce programming model efficiently.**

   Spark is primarily designed to optimize batch operations on massive data set. Map-Reduce paradigm makes these operations significantly faster, as compared to traditional procedural paradigms. However, this also introduces latency into the system, and makes it unsuitable for hard real time operations. For example, in space-shuttles, or even in modern airplanes, GBs of data is collected every second. This data, processed into relevant information in real time could help avoid catastrophes, and streamline processes to save potentially billions of dollars every year.

   With Spark Stream APIs, these gaps are closing in, and performance near real time have been achieved. However, since Spark primarily follows a batch processing model, hard real time applications cannot be addressed by Spark's MapReduce paradigm [3].

6. **Why do you think the MapReduce paradigm is such a widely utilized abstraction for distributed shared memory processing and fault-tolerance?**

   In the case when data is distributed on several nodes, it is a good strategy to let each node does all the computations possible on the local data before the data is aggregated. This correspond to the MapReduce paradigm where the each worker node applies the map function before shuffling them to a common node for the reduce function. In MapReduce, each worker node periodically reports heartbeat to the master node to signal its state. When the heartbeat is missing for some period of time, the master node would assume that the worker is dead and restart the job on another worker that has the data partition. Therefore, fault tolerance is achieved by storing data on multiple nodes and heartbeats.

2.2 Analysis

1. **Do you expect and observe big performance differences between the RDD and Dataframe/Dataset implementation of the GDelt analysis?**

   In principle, we were expecting Dataset program to perform significantly faster than the RDD program. However in our analysis, we observed that RDD was in fact faster than the Dataset counterpart. As shown in graph for fifth question under Analysis, we have tried running the program on the same computing platform for different size of Dataset. This speed observation holds for up to 50 GDelt CSVs (1.5 GB). On average, RDDs are observed to perform roughly twice as fast as the Dataset equivalent.

2. **How will your application scale when increasing the amount of analyzed segments? What do you expect the progression in execution time will be for, 100, 1000, 10000 files?**

   We broke down the algorithm for analysis, and as per our observation, the algorithm follows the complexity as shown in Table 1, for a given file as an input. For this analysis, we ignore shuffle and other spark internal operations among the partitions, and focus on the algorithm itself. Here, n represents the number of entries in a given file.

Table 1: Complexity analysis of each step [1].

S.no	Operation	Order
1	Read all files into one dataset	$O(n)$
2	Select relevant columns in the table	$O(1)$
3	Filter rows with null columns	$O(n)$
4	[Map] Extract relevant information for Date and Topics	$O(n)$
5	Group rows by Date	$O(n)$
6	Group Topics for counting purpose	$O(n)$
7	[Map] Count the size of each Topics group	$O(n)$
8	Sort the Topics field in descending order of frequency count	$O(n \log n)$
9	Return top 10 fields	$O(n)$

Based on this analysis, we concluded that for one file, the processing involves computation of order $O(n \log n)$.

However, if we change our perspective as directed in the question - understanding the growth as per the number of files, and not the number of rows in a file, we predict that the total processing time increases linearly. Assuming that the number of entries in each file is roughly the same, processing each file can be abstracted as constant time process, and each file would need to be processed once, making the entire operation of order $O(n)$; where n is the number of files.

3. **If you extrapolate the scaling behavior on your machine to the entire dataset, how much time will it take to process the entire dataset? Is this extrapolation reasonable for a single machine?**

   The entire dataset contains more than 120000 files. We ran analysis from 5 files to 50 files on our local machine and the execution time is shown in Figure 1. From this figure, we fit a line on the blue dots (RDD) and use the line equation to calculate the time that is needed to analyze 120000 files. We fit a linear line and estimated that it takes around 10 hours to process the entire dataset. This extrapolation should be somewhat reasonable since we estimated a linear growth as a worst case scenario. Moreover, the communication between cores on a single machine is expected to be stable. Of course, if the data that needs to be transferred excess the bandwidth, it can take up more time.

4. **Now suppose you had a cluster of identical machines with that you performed the analysis on. How many machines do you think you would need to process the entire dataset in under an hour? Do you think this is a valid extrapolation?**

   Since in question 3, we estimated that it takes 10 hours to process the entire dataset, it is expected that 10 machines are required to achieve the process time of under an hour. However, this is not valid due to the fact that these machines must communicate through the network, which has limited bandwidth. We need more than 10 machines to increase the parallelism and spare more time for them to shuffle the data around during the grouping stages.

5. **Suppose you would run this analysis for a company. What do you think would be an appropriate way to measure the performance?**

   An appropriate metric depends on the objective and use case of the company. Since our analysis is finding the top 10 topics per day, we can imagine a use case where the company wants to monitor emerging popular topics or change of topics throughout the years. In this case, one-pass of all past data is enough to collect information from the past. For each new day, an analysis on the new 96 files (a file per 15 minutes) needs to be run. The executing time is not crucial in this case and an appropriate metric would be the money spent on running the analysis.

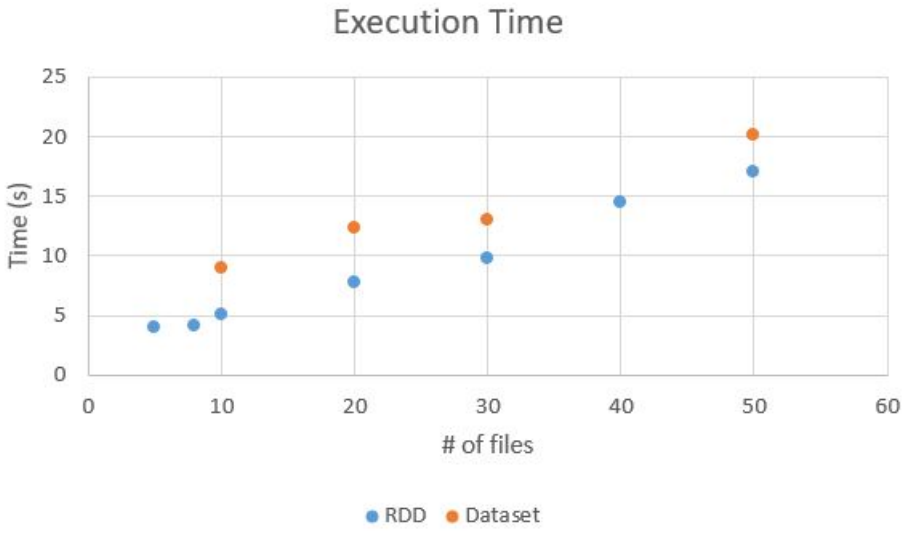

Figure 1: Execution time of the programs on a local machine with 8 logical processors.

References

[1] Apache spark github. https://github.com/apache/spark.

[2] Apache spark website. http://spark.apache.org/.

[3] Data flair training, apache spark tutorials. https://data-flair.training/blogs/category/spark/.

[4] Github scala-spark from rohgar. https://github.com/rohgar/scala-spark-4/wiki.