Delft University of Technology

Super Computing for Big Data
ET4310

Lab 1 Report

Authors:
Kyriakos Psarakis (4909437)
Rafail Skoulos (4847482)

September 23, 2018

RDD Based Implementation

```
val data = spark.textFile("./100Seg.csv") //Read the input file.
      .map(_.split("\t")) // Each column is separated with tabs.
      .filter(_.length > 23)  // Remove the columns that do not have the field AllNames.
      .map(k => dateAllNamesKVP(k(1), k(23)))  // Create the key value pair(Date,AllNames
          ).
      .flatMapValues(x => x) // Flatten it to the key value pair (Date,topic).
      .filter(!_._2.contains("Type␣ParentCategory")) //Filter out the topic Type
          ParentCategory.
      .map(k => (k, 1)) // Add to each pair the integer 1 in order to do the reduce
          operation in the next step.
      .reduceByKey(_ + _) // Do the reduce operation with key the date,topic pair.
      .map(k => (k._1._1, (k._1._2, k._2))) // Map them to the expected output format.
      .groupByKey() // Group them by the Date.
      .map(g => (g._1, g._2.toList.sortWith(_._2 > _._2).take(10))) // Take the top 10
          per date.
```

The above code is the main part of our RDD implementation of the GDelt analysis. At first, we use narrow transformations in order to format the input dataset into the key value pair we need in order to do the reduce operation. In addition we filter out any unwanted datapoints. Finally, when we have the result we do a wide transformation in order to group our results into the requested output format of top 10 topics per date.

Dataset/Dataframe Based Implementation

```
val ds2 = ds
      // Filter out the null column
      .filter(a => a.DATE != null && a.AllNames != null)
      // Take the Date and AllNames columns.
      .select("Date", "AllNames").as[(Date, String)]
      // Flatten them to take the key value pair (Date,topic).
      .flatMap { case (x1, x2) => x2.replaceAll("[,0-9]", "").split(";").map((x1, _))}
      // Filter out the topic Type ParentCategory.
      .filter(!_._2.equals("Type␣ParentCategory"))
      // Count the times each (date,topic) tuple is in the dataset
      .groupBy('_1,␣'_2).count
      // Find the rank of each topic in each date
      .withColumn("rank", rank.over(Window.partitionBy('_1).orderBy('count.desc)))
      // Take the top 10 of each Date
      .filter("rank"<=10)
      // Drop the extra column rank
      .drop("rank")
      // Create the expected output in JSON format
      .groupBy("_1" as "data")
      .agg(asList(collect_list("_2"), collect_list("count")) as "result")
      .select("data", "result".cast(finalListSchema))
      .toJSON
```

The above code is the main part of our Dataset/Dataframe implementation of the GDelt analysis. Firstly, we filter out the unwanted columns and rows. Then, we flatten the input in order to obtain the required tuple format in order to proceed with further computations. Furthermore, we count the occurrence of each date,topic pair. Finally, we calculate the rank of each counted date,topic pair in order to obtain the top 10 per date. Also, as a last step we format the output in the required format.

General Questions

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

In a typical use the wide dependencies are more expensive, case being, that they require shuffling of data over the network. For example, the sorting or group-by operations, when the data are distributed across many different machines can not happen without extensive communication over the network between the data nodes. On the contrary narrow operations do not require any over the network communication. Only in memory processing occurs, which is very fast.

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

The shuffle operation is used by Spark to re-distribute data so that it's grouped differently across partitions. It is a complex and costly operation as it involves copying data across executors and machines. Shuffle operations can consume significant amounts of heap memory since they employ in-memory data structures to organize records before or after transferring them. Furthermore, when data does not fit in memory Spark will spill these tables to disk, causing the additional overhead of disk I/O and increased garbage collection. So, in order to increase the performance of Spark, the number of shuffle operations should be reduced.

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?

First of all, in Dataframes a schema is inferred, making structured data ingestion faster and easier. In addition, in the Dataframe two optimizing methods are implemented. The first is project Tungsten, which stores the data in off-heap memory in binary format saving a lot of memory space, removes the Garbage Collection overhead and expensive java Serialization is also avoided. The second is the Catalyst Optimizer, with primary goal to optimize the execution plan. Furthermore, the Datasets add Compile-time type safety. However, the RDDs still perform better when the data are unstructured.

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 Spark's programming model help you recover the lost work?

RDD is the main abstraction of Spark. RDDs are immutable dataset. It remembers all the transformations done on every step through the lineage graph created in the DAG (Directed Acyclic Graph), and in case of failure, Spark refers to the lineage graph to apply the same operations.In case of worker node failure, the executors in that worker node will be killed, along with the data in their memory. The cluster manager will assign another node to continue processing. This node will operate on the particular partition of the RDD and by using the lineage graph, it will execute the series of operation needed to complete the task. The data is also replicated to other worker nodes to achieve fault tolerance.

5) Can you think of a problem/computation that does not fit Spark's MapReduce-esque programming model efficiently.

A problem that does not feat the Spark's programming model is real time stream processing. Both the Sprak Streaming and Spark Structured streaming APIS work with micro batches of data, ingesting them between some time intervals. Moreover, the processing can not continue if a previous batch has not finished processing. So for real time streaming Apache Storm and Apache Flink are better alternatives.

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

MapReduce provides a clearer storage layer for accessing the address space from any distributed cluster through shared-memory programming. Also, contains the internal functions for synchronization and barrier between the map and reduce phases. This makes it a good abstraction for distributed shared memory processing and fault-tolerance because it utilizes all the necessary characteristics of these two.

Implementation analysis questions

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

We expect a performance difference in favor of the Dataframe/Dataset implementation. Reason being that the GDelt data are structured, so all the optimizers can work to increase performance. But, the performance increase will show only when the data are really big and being proccesed in a distributed environment.

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?

In our machine the processing of 1 segment(30MB) takes 8 second, the one of 10 segments (300MB) takes 8 seconds too, probably because of the time it takes for SQL to launch, and the one of 100 segments (30GB) takes 24 seconds.From the runtime of 10 and 100 segments we can create an equation about the expected execution time for any amount of segments: $execution_time = (segments + 35) * 0.17(seconds)$. So the expected execution time for 1000 segments will be 175.95 seconds and the one for 10000 segments will be 28.4 minutes.

3) If you extrapolate the scaling behavior on your machine (for instance for 10, 50, 100 segments) to the entire dataset, how much time will it take to process the entire dataset? Is this extrapolation reasonable for a single machine?

As we found in the question above the expected runtime is given by the equaton: $runtime = (segments + 35) * 0.17(seconds)$. So, if we assume that we have 100.000 segments in the entire dataset, the runtime would be 4.7 hours. This extrapolation is not reasonable for a single machine due to memory and storage limitations reason being that the dataset is far larger than the computers ram capacity.

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?

In the previous question we derived that the processing of the entire dataset by one machine will take 4.7 hours. So we will need 5 of these machines in order to process the entire dataset in under an hour. This extrapolation may be invalid because except from the memory and storage limitations mentioned above, now we should consider about the cost of the communication between the machines, which increases the execution time.

5) Suppose you would run this analysis for a company. What do you think would be an appropriate way to measure the performance? Would it be the time it takes to execute? The amount of money it takes to perform the analysis on the cluster? A combination of these two, or something else?

We think it should be a combination of both. If the expected profit of the project that we do data analysis on is big and requires fast decision making the company must pay more for faster data analysis. If the data analysis can wait and is not critical for the company's decision making in the near future the best way to measure performance is the money it costs, and that should be minimized. So, a nice metric would be a price-performance ratio (Thoughtput/Price).