Data Analysis with Apache Spark and Zeppelin

Proseminar Data Mining

Philipp Klocke
Fakultät für Informatik
Technische Universität München
Email: klocke@in.tum.de

***Abstract*— Over the last few years, Data Mining has become more and more important. In this paper we give an overview over Data Analysis with Apache Spark as proposed by Zaharia, et al. [1] and visualization of the results with Apache Zeppelin. We mainly present this through an example analysis of Taxi Data with PageRank.**

***Index Terms*— Apache Spark Zeppelin MapReduce**

I. Introduction

In the field of Data Mining, many different frameworks are being used for different problems. A framework is comparable to a toolbox. It implements reusable cross-problem functions like the distribution of tasks to workers or error-handling. Using such frameworks for data analysis is necessary to stop re-inventing the wheel and focus on the actual problem instead.

In this paper we will show two of those frameworks: Apache Spark and Apache Zeppelin. Both of the presented frameworks are open software under the Apache license.

We will first give an overview of Spark and its programming concepts which we will then compare to the interface MapReduce. In the second part, we explain how to use Zeppelin, with focus on the interaction with Spark. Our third part is a real world example on how Zeppelin and Spark can integrate with each other to allow simple and fast Data Analysis with Visualization.

In the end we will give an overview of similar academic work and conclude that frameworks like our examples are very important in terms of Data Mining and Analysis.

II. Apache Spark

Apache Spark is a cluster-computing framework that runs on Hadoop-Clusters[1].

It was built after MapReduce, an interface for big parallel data analysis, which it tries to improve in various ways. The main concept of Apache Spark is the introduction of *resilient distributed datasets* (RDDs) for cyclic data analysis. [1]

A. Programming Model

As mentioned previously, RDDs play an important role in Spark. An RDD is a read-only collection of objects, that can be partioned across a set of machines. When one of the nodes containing data of an RDD fails, the data has to be restored. This happens automatically, because RDDs also contain information about their "lineage" (and thus can easily be recomputed). [1] Also RDDs can be cached for later usage. For more information on RDDs see [2].

Spark exposes the following basic methods to a driver program, written in Java, Python, Scala or R.

- Map (to map a function on an RDD)
- Filter (filtering result sets)
- Reduce (join result sets)

All of these operations can be invoked with a RDD (source) and a Function, like functional programming. Their output is another RDD that can be further processed or saved to disk. Spark will parallelize and schedule the functions for execution.

To create an initial RDD, one can invoke I/O-Operations, interactively define a dataset or load it from a remote server.

B. Comparison to MapReduce

MapReduce in general refers to two different things. First it is an interface for Data Analysis Frameworks that offers Map- and Reduce Methods. Second it is the first implementation of said interface by Google. [3] In the following, MapReduce, means the former.

MapReduce is mainly using a Divide and Conquer approach. Its basic principles are the Map and Reduce methods. They are being distributed across workers, each of them performing either map or reduce but not both (at least at the same time). A worker can be compared to a process, while a node is a server (with possibly multiple processes). Using a distributed system of many workers results in a higher probability of failures, so intermediate values (e.g. generated from map-workers) have to be saved, or must be recomputed if the node fails.

Spark solves this problem by tracking the lineage (creation history) of all intermediate RDDs. When one node fails, one partition of an RDD is lost. Because the lineage is known to all other workers, they can replace the failed worker easily and recompute the data lost. This drastically improves the fault-tolerance and the speed in which errors can be handled. [1]

The biggest improvement of Spark though is its capability for cyclic data analysis. In MapReduce, the driver program specifies input, map- and reduce-functions and gets the output. This is a mostly linear, acyclic operation. It can be shown however, that many problems require a reuse of computed data for efficient operation. These are mostly optimization problems, in which a better approximation is found in every iteration of the algorithm. In Spark this can be easily done because RDDs are automatically cached for later use or can

[1]Hadoop is a framework for network-distributed storage. See the official website for more.

explicitly be cached. The latter explicit variant stop Spark from overriding the cache when needed. [1]

Also Spark has an extension system to increase its applicability across different problems. For more information on this, refer to II-D.

C. Example Usage

A basic example for parallelization of driver programs is the approximation of Pi. To understand the idea of this algorithm we imagine a square from (-1,-1) to (1,1), with a circle of radius 1 inside. The area of the circle is $A_c = \pi \cdot r^2 = \pi$, while the area of the square is $A_s = s^2 = 4$. We now randomly throw darts into our square and check if they land in the inner circle. The ratio of circle-hits to darts is roughly $\frac{A_c}{A_s} = \frac{\pi}{4}$.

```
val count = sc.parallelize(1 to
    ↪ NUM_SAMPLES)
.filter { _ =>                        // Map
          val x = math.random
          val y = math.random
          x*x + y*y < 1
}.count()                             // Reduce
val pi = 4.0 * count / N
println(s"Pi is roughly ${pi}")
```

In this Scala script we define a RDD consisting of the numbers 0 to `NUM_SAMPLES` and then apply a filter. This is comparable to a map-function that maps anything satisfying the predicate to a new RDD and drops the rest. We then count the remaining records with a function that, similar to a reduce-function, increments an accumulator for every element. For a basic imagination of this process, refer to Figure 1.

The example is taken from the official website[2] with slight modifications for display purposes.

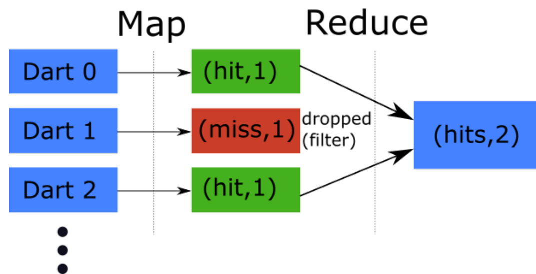

Fig. 1. Map and Reduce of Pi Approximation

D. Components

As Spark in its pure form is only applicable to MapReduce-Problems, there exist many extensions to enhance its field of use. To understand this Extension-System it is necessary to know how different components interact with each other. Spark is following a layer-architecture design, like shown in Figure 2.

[2]https://spark.apache.org

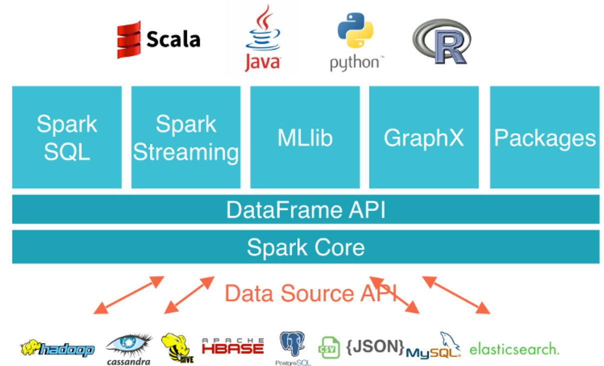

Fig. 2. Illustration of Spark components[3]

We described the Spark Core in the previous paragraphs. To load data into the Core, the lower layer Data Source API is used to access e.g. SQL, hadoop, or other storage solutions.

On top of the Spark Core, the DataFrame API gives driver programs the ability to perform various operations on DataFrames. A Spark DataFrame is similar to a Table in SQL, but distributed. Also Spark's built-in optimizer Catalyst will try to optimize programs based on this API.

The following packages are distributed with spark per default and sit on top of the DataFrame API:

1) SparkSQL: translates SQL-Queries to Spark for execution on manifested RDDs over the DataFrame API.

2) Streaming: allows to apply the default Spark batch-processing methods on input streams.

3) MLLib: is a machine learning extension for Apache Spark that offers different training methods, optimization and a pipeline API, using native C++ Libraries and thus is scalable fast. [4]

4) GraphX: Can be used for operations on graphs. We will use this package in the later example.

The top layer of spark are the language bindings for driver programs, that use features of spark. They can be accessed via an executable program, or with interactive shells.

III. Apache Zeppelin

Apache Zeppelin is a visualization framework, distributed with various interpreters like Apache Spark. Because the latter is only processing data, we need a solution to generate meaningful results out of it.

A. Data Visualization

For interactive work, Zeppelin offers so-called Notebooks, each consisting of multiple sections. A section is basically a place to put Code that can be run either standalone or all sections at once. Each section has their own interpreter, defined in the first line of the script. It could for example be spark,

[3]source: W. Zhou. "Hadoop, HDFS, MapReduce and Spark on Big Data.", https://weidongzhou.wordpress.com/2015/09/08/hadoop-hdfs-mapreduce-and-spark-on-big-data/

markdown, SQL, R, angular, or a custom interpreter. After execution of a section, the `stdout`-Output is interpreted by Zeppelin and then displayed. If the output begins with an interpreter string, the corresponding interpreter is used for visualization. Else it's printed as simple text or with the default output interpreter for the output type. (e.g `%table` is the default for sql)

The standard setup provides some tutorial notebooks, containing various Spark, SQL and other scripts. The first notebook gives an overview of using Zeppelin with Spark. It loads a file from here [5] and maps it with a function to transform int into a table (note how we used the DataFrame API here). When executed, the Output from Spark is displayed below the Script. Because it is not interpreted, it is printed as simple text.

Zeppelin Notebook Job anonymous

Zeppelin Tutorial/Basic Featu... Head default

Load data into table FINISHED

```
1  import org.apache.commons.io.IOUtils
2  import java.net.URL
3  import java.nio.charset.Charset
4
5  // Zeppelin creates and injects sc (SparkContext) and sqlContext (HiveContext or SqlContext)
6  // So you don't need create them manually
7
8  // load bank data
9  val bankText = sc.parallelize(
10     IOUtils.toString(
11         new URL("https://s3.amazonaws.com/apache-zeppelin/tutorial/bank/bank.csv"),
12         Charset.forName("utf8")).split("\n"))
13
14 case class Bank(age: Integer, job: String, marital: String, education: String, balance: Integer)
15
16 val bank = bankText.map(s => s.split(";")).filter(s => s(0) != "\"age\"").map(
17     s => Bank(s(0).toInt,
18             s(1).replaceAll("\"", ""),
19             s(2).replaceAll("\"", ""),
20             s(3).replaceAll("\"", ""),
21             s(5).replaceAll("\"", "").toInt
22         )
23 ).toDF()
24 bank.registerTempTable("bank")

import org.apache.commons.io.IOUtils
import java.net.URL
import java.nio.charset.Charset
bankText: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[3] at parallelize at <console>:30
```

Fig. 3. First Zeppelin Note

It then gives examples for SQL statements to query the data. This output is visualized with the `%table` interpreter and can be displayed using a plain table-view or various diagrams, like bar-, pie-, line-area-, and line-charts or a scatter plot. The plot-type can be changed with one click. In both Notes of Figure 4, the following SQL-Statement was used:

```
%sql
 select marital, count(1) value
 from bank
 group by marital
 order by marital
```

B. Interpreter Status

Zeppelin can not only visualize the output of Spark or other interpreters, but it can also display the status of Spark. This includes a list of running and finished jobs, as well as the Execution plan for each of them.

For our example in Figure 3 and 4, only one job is created. This happens due to the lazy evaluation of Spark. It does not actually execute the job and create the bank-Table until it is queried.

For the sake of simplicity we only assume one SQL-Query Execution in the following paragraphs.

Thus the job is divided into two separate stages. The creation stage and the query stage. For each of those, zeppelin

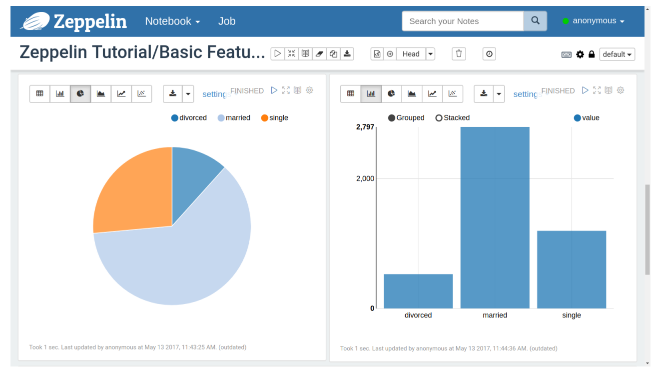

Fig. 4. Visualization of SQL output as pie and bar chart

will display various information including a timeline, summary metrics per task, etc. but also a DAG (directed acyclic graph), showing the execution plan of spark.

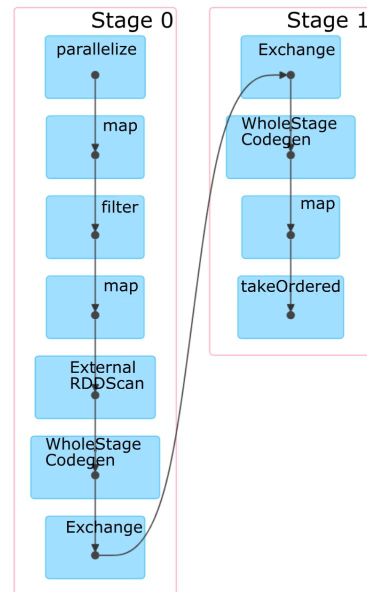

Fig. 5. Spark Execution Plan as DAG

We can see how the two stages are processed separately and connected via two Exchange Nodes. This marks the DataFrame API described above.

IV. Example Usage

We have shown how Spark and Zeppelin integrate with each other and will now demonstrate this on a small real-world example.

The city of New York provides open data of taxi routes on a monthly basis. This dataset includes pickup and dropoff location, time, distance, fares, etc.

For our analysis the Dataset from June 2016 of Manhattan from here was chosen. It consists of 11135470 records with a total size of approximately 1.75GB.

In the real world, one would have to analyze Datasets of the whole year to get meaningful results, as there are e.g. seasonal changes. Still we can analyze one month to give a rough overview over the processes and capabilities of our frameworks.

```
val taxiText = sc.textFile("./
    ↪ yellow_tripdata_2016-06.csv")

case class Taxi(
  pickup_longitude : Double,
  pickup_latitude : Double,
  dropoff_longitude : Double,
  dropoff_latitude : Double
)

val taxi = taxiText.
    ↪ mapPartitionsWithIndex { (idx, iter
    ↪ ) => if (idx == 0) iter.drop(1)
    ↪ else iter }.map(
line => line.split(",")).map(s => {
new Taxi(
  toLon(s(5).toDouble),
  toLat(s(6).toDouble),
  toLon(s(9).toDouble),
  toLat(s(10).toDouble)

}).filter(x => x.pickup_longitude != 0).
    ↪ toDF()

taxi.registerTempTable("taxi")
```

In this code snippet, we load our downloaded csv file into a simplified model consisting only of pickup and dropoff locations. We map the geographic coordinates to a simplified x/y model, for easier plotting. The code for this step is omitted. (It could be improved by using Mercator projection, but that would go beyond our scope)

Now that we've loaded all relevant columns into a DataFrame, called `taxi` we can query with SparkSQL. When simply querying for all pickup locations (limited by 10000 due to performance issues), and plot them, we can get a good approximation of the shape of Manhattan. The central park for example is white, because no taxis pick up passengers there. Also one can see a grid in the data points. This is caused on the one hand by the grid of Manhattan, but also by computational floating-point losses.

At the moment, Zeppelin has no support for heatmaps/density maps which could be used for a better visualization.

After executing the code above, SQL-Queries like average drivetime, the sum of all fares (income of taxi drivers), etc. could be easily executed. We leave it to the user to think of interesting applications.

We now use a different approach on the data: the PageRank algorithm by Google. PageRank was designed to order the relevance of websites by interpreting them as vertices and links from one page to another as edges. This should help to display good query results in the Google Search Engine. Sparks GraphX comes with a Scala implementation of PageRank[4].

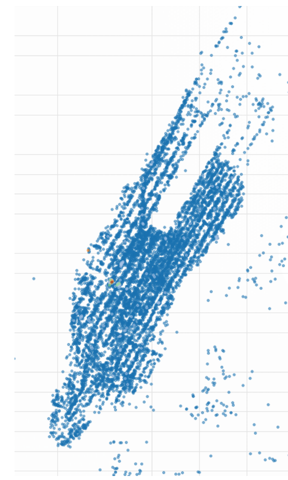

Fig. 6. Visualization of pickup points

With this algorithm we can determine the relevance of locations in Manhattan. In our simple implementation the weight of an edge represents the amount of taxis that drove this route. To improve the results, one could modify the weights by multiplying them with the passenger count or include other data.

```
import org.apache.spark.graphx._
import org.apache.spark.graphx.lib
import org.apache.spark.graphx.lib.
    ↪ PageRank
import java.security.MessageDigest
import scala.math.BigInt


def hash_tuple (x: Double, y: Double) :
    ↪ Long = {
        return BigInt(MessageDigest.
            ↪ getInstance("MD5").digest((
            ↪ x.toString + y.toString).
            ↪ getBytes)).toLong
}

val edges = taxi.map(x => (hash_tuple(x.
    ↪ pickup_longitude, x.pickup_latitude
    ↪ ), hash_tuple(x.dropoff_longitude,
    ↪ x.dropoff_latitude)))

val vertices = taxi.map(x => List((x.
    ↪ pickup_longitude, x.pickup_latitude
    ↪ ),(x.dropoff_longitude, x.
    ↪ dropoff_latitude))).flatMap(
    ↪ identity).distinct().map(x => (
    ↪ hash_tuple(x._1, x._2), x))
```

[4]refer to the implementation on github or [6] for details

```
val graph = Graph.fromEdgeTuples(edges,
  ↪ (-1,-1), None).groupEdges((x,y) =>
  ↪ x+y)
val pr = PageRank.run(graph,
  ↪ NUM_ITERATIONS, 0.001)
vertices.join(pr.vertices).map({case (id,
  ↪ (location, rank)) => (location._1,
  ↪ location._2,rank)}).toDF().
  ↪ registerTempTable("taxiranks")
```

We now have a new DataFrame called `taxiranks`, that consists of a position in x and y coordinates and the rank, or importance of the location. Selecting the top 10000 locations and plotting them with the rank as point-size yields the following output.

Fig. 7. R plot of PageRank result

When comparing this figure to a map of Manhattan, one can find out what the biggest points are: For example one can see a big dot corresponding to the Whitney Museum of American Art in the lower left, or Pennsylvania Station in the left mid.

With this data, taxi companies could calculate which areas they should assign more drivers to, or the city of New York could improve it's streets or build new taxi ranks.

We have shown that implementing a standard analysis algorithm and reading relevant results is straight-forward when using Spark and Zeppelin.

For running Spark and Zeppelin, a personal Laptop with a current-gen Intel Core i5 was used. Thus only 3 iterations of PageRank could be executed successfully. For more precise results, the reader is encouraged to try out the algorithm himself on a larger scale.

V. Related Work

The interface MapReduce was presented by Dean and Ghemawat in 2008. [3]

In 2010 Zaharia et al. presented the concept of Spark including a comparison to other frameworks [1]. He also explained the concept of RDDs [2] and the Spark Streaming Model in 2012 [7].

MLLib was introduced in 2016 by Meng et al. [4]

Armbrust et al. showed, how Spark is scaling in real world applications in their work of 2015 [8].

VI. Conclusion

In this paper we have shown how Apache Spark and Zeppelin can be used together to analyze and visualize data. Relying on such mighty tools is essential for Data Analysis to stop reinventing the wheel and focus on meaningful results.

In the future, more interpreters for zeppelin and more packages for spark can be imagined, improving the applicability for these frameworks even more.

Also, some of the big IT-Companies like Google, Facebook, etc. are always developing new frameworks for even faster computation or different problems.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[4] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016.

[5] S. Moro, R. Laureano, and P. Cortez, "Using data mining for bank direct marketing: An application of the crisp-dm methodology," in *Proceedings of the European Simulation and Modelling Conference - ESM'2011*, P. N. et al., Ed. Guimaraes, Portugal: EUROSIS, Oct. 2011, pp. 117–121.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[7] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters." *HotCloud*, vol. 12, pp. 10–10, 2012.

[8] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia, "Scaling spark in the real world: performance and usability," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1840–1843, 2015.