An Overview of Apache Spark Ecosystem and Modules

Linxiao Bai
Master of Science
Goergen Institute for Data Science
University of Rochester
lbai2@ur.rochester.edu

Nilesh Patil
Master of Science
Goergen Institute for Data Science
University of Rochester
npatil4@ur.rochester.edu

ABSTRACT

In this paper, we provide an overview of Apache Spark ecosystem and related key Spark modules, including, low-level interfaces to machine and hive-level api interfaces for users (e.g. Spark Core & Spark SQL). Also, we will compare the two modules with their predecessors, Hadoop-MapReduce and relational DataFrame. Performances improvements over the precursors will also be studied. The idea is to extract the excellent innovation and designs of Spark system.

1. INTRODUCTION

In 2009, Netflix held a competition with $1 million for the winner. The competition's aim was to get an improved approach for building Netflix's recommendation engine. The huge prize amount attracted UC Berkley's AMPLab, and they designed an algorithm based on the most popular parallel computation engine at that time, Hadoop. Soon the researchers realized that the bottleneck of the problem at hand is not just the recommendation algorithm, but the ineffective programming model of Hadoop for this task. With great ingenuity, AMPLab designed a brand-new computation engine called Spark.

Although AMPLab did not win the prize in the end, Spark's achievements adaptation as a multipurpose data science tool has gone beyond anyone's imagination. After its first release, Spark-0.5 in 2013, Spark went through 12 major updates. Currently, the most recent version is 2.1.0, and it is still the most popular open-source project of Apache Software Foundation, and the first choice as big-data engine for cluster-computation[5].

Spark was created for massive parallel computation on large-scale distributed cluster. Although Spark Core is a derivative of Hadoop MapReduce framework at basic level, with rigorous optimization and design changes, it quickly went beyond Hadoop's performance, providing faster, more resilient and more convenient computation experience. Also, with extended modules and libraries like Spark SQL, Spark Streaming, GraphX, and MLlib, Spark extended its power to cater different computation needs, and adapts to different environments for applications in data science.

We hope this paper will bring provide a better understanding of Spark ecosystem and its key innovations. We will start the paper with an introduction to Spark ecosystem, and move on to the interfaces and designs of key modules, including file-system, resources management, Spark Core and Spark's extended libraries.

2. Spark Ecosystem

The Spark Ecosystem is also known as the AMPLab's BDAS (Berkley Data Analytics Stack). It is a "big-data platform" that connects - users, algorithms, and machine. It provides a one stop solution to distributed data storage, resource management, and interfaces to data processing for data munging and analytics. The key towards understanding BDAS is its hierarchical structure and module dependencies. Figure-1 shows that BDAS is a structured collection of modules with dedicated functionalities. Together they form a powerful, multi-purpose big data platform.

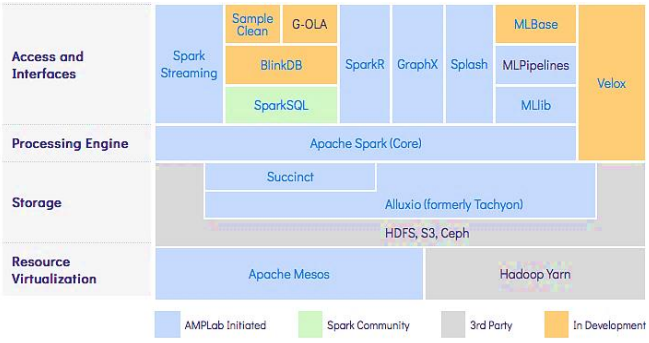

Figure 1. BDAS, Modules at Bottom Faces Machine, Top faces User and Specific Jobs.[6]

Different modules at different hierarchy level of the ecosystem are dedicated to solve different problems for a variety of data processing & analytics applications. Figure 2 shows how Spark ecosystem connects user, machines, and algorithms. The key point is that Spark takes the responsbility of bridging algrithms and machines that are incharge of exection. At the same time, Spark provides a smart and user friendly interface at both ends so that users can work without having to worry about the effectiveness and robustness of distributed execution.

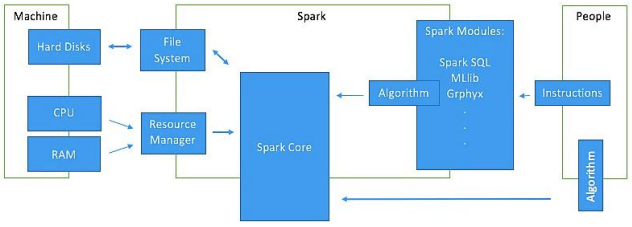

Figure-2. Spark, User, Algorithms, and Machine

Apart from the low-level interface to work with machines, Spark offers high-level APIs that provide a user-friendly environment. Furthermore, there are advanced modules designed for specific analytical tasks. Table-1 lists some of the most popular Spark modules in the community. We will only cover Spark SQL in the following section as a representative of this module system.

Table-1. Sample list of Extended Spark Modules [7]

Name	Usage
Spark Streaming	For real-time streaming Spark applications.
GraphX	API for graph computation
MLlib	API for Spark machine learning library
Spark R	Spark interface in R
Spark SQL	SQL engine for structured data
Zepplin	Notebook-like interactive Spark interface
GeoSpark	Spatial data engine on top of Spark

3. Spark Modules Analysis

As described in the previous section, Spark ecosystem is a collection of modules that depend on each other, and together they form a powerful platform. This section will provide an independent analysis for key modules of Spark. Our primary focus is on Spark Core and Spark SQL because they not only represent Spark's interface to low-level algorithms and abstractions but are also most frequently used tools by Spark users.

3.1 Storage

The relationship of storage system to a computation engine is like an armory to an army. Spark relies on distributed file systems to input data from and output results to persistent hard disks. Spark provides support to a wide range of storage systems. Its API supports main-stream file systems like HDFS, HBase, and Amazon S3. The polymorphism to file systems allows Spark to be adaptive and easy to deploy without sacrificing performance.

3.2 Resource Management

Resource management to a computation engine is like a commander to an army. Spark needs resource manager to schedule jobs and allocate CPU and RAM for computation across multiple machines. Depending on the file system and the environment, different choice of resource management may be deployed. For example, Yarn[8] is most commonly used on HDFS. The *BlueHive* cluster at University of Rochester uses SLURM[9].

Table-2 shows spark deployment modes and the corresponding resource manager used in different environments.

Table 2. Spark Deployment and Environment

Environment	Mode	Resource Management
Local machine, PC	Local	None
Small cluster w/o resource negotiation	Standalone	Spark-Standalone
Big cluster	Cluster	YARN, Apache Mesos, SLURM, EC2

3.3 Spark Core

As the name suggests, Spark Core is the most essential part of Spark ecosystem. Almost every other module is either designed completely on top of Spark Core or implemented to extend its functionality. In fact, usually the references to Spark refer to the basic functionality provided by Spark Core.

The game changing designs of Spark Core contain two major components:

- An in-memory data abstraction object called Resilient Distributed Datasets (RDD)
- A scheduler called DAGScheduler.

3.3.1 In-memory Computation

Unlike Hadoop MapReduce, where the result of each map-reduce operations is written to disk, Spark and its RDD abstraction allows application to cache the intermediate result of calculation in RAM as Java object for further computation [1]. This so called ***in-memory computation*** is Spark's biggest improvement over Hadoop. The in-memory computation makes consecutive data transformations easier to code - for the user and faster to implement – for the machine. Also, it saves the system from huge amount of disk I/O and serialization/deserialization cost. Benchmark experiments carried out by the AMPLab suggest that Spark outperforms Hadoop by up to 20 times in different application scenario [10].

The experiment runs iterative machine learning algorithm with 100GB of data. Hadoop, HadoopBinMem (an in-memory optimization of Hadoop), and Spark RDD are tested in this scenario. Results are measured as – "training time cost in seconds". Figure-3 shows the performance of each computation framework.

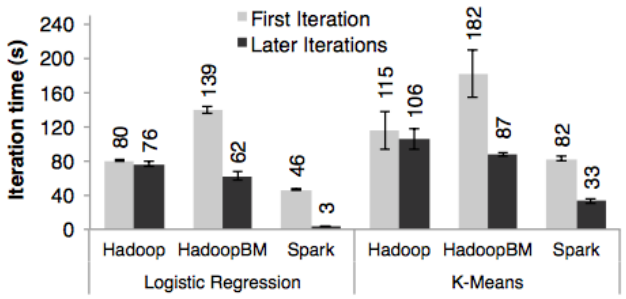

Figure-3. Spark Performance Compared to Hadoop[10]

Figure-3 shows a clear gain of Spark over Hadoop in computation time. Notice that core Hadoop shows almost constant time cost at every iteration, and the first iteration is guaranteed to have a disk output operation to load the data. The time constancy is because for the remaining iterations, Hadoop also reads data from the disk.

Also, even with in-memory optimizaiton, Hadoop is still slower than Spark. This phenomenon is explained by the AMPLab as the overhead of parsing Hadoop file. Although data is stored in memory, but the format is not desgned for computation tasks by Hadoop. Extra computation is still needed to convert Hadoop file to Java object that can be processed with map-reduce. While Spark directly stores RDD as Java objects, which avoids the overhead. Other drawbacks of core Hadoop may also influence its performance. For example, the minimum overhead of Hadoop software stack.

3.3.2 Resiliency

Another distinct improvement Spark proposed is its fault recovery strategy, lineage using DAGScheduler.

In Hadoop, when a map-reduce task fails at any node, everything in the memory is lost for that node and recomputed at another node i.e. the task begins anew at a different node. The resilience strategy of Hadoop relies on the recover mechanism of HDFS, which is essentially making multiple copies of all disk files and storing at different locations. This is also the reason why Hadoop needs to save its result at each step. So, when a fault occurs, Hadoop can load to the closest checkpoint.

However, this resiliency mechanism has several drawbacks. First, it consumes huge amount of disk I/O and communication bandwidth when making copies of disk, which is the most expensive operation in parallel computation. Second, a computation task success means successful completion of all subtasks for all nodes. Should any node fail, the whole system must roll back and re-compute. This not only means that the chance of failure is multiplied because of the "bound success", but also means that correct computation done by other nodes is of no use anymore.

The lineage concept proposed in Spark fixes theese drawbacks in Hadoop. Lineage is formally defined as logging transformations used to build a RDD at coarse-grained level. As an application launches, DAGScheduler process will log the relationship of previous data and current data at partition level. When a fault at certain partition occurs, Spark will trace back its lineage, and find all the actions used to compute results from the faulty partition, and re-compute this portion only. While the computation results at rest of the partitions remain as it is. This recovery strategy is more advanced than Hadoop's, and provides faster reliable computation experience.

In a fault-recovery experiment AMPLab carried, performance of Spark RDD are evaluated under the scenario of a node failure. Figure 4 shows the iteration time cost at the presence of a scheduled node failure at 6th iteration.

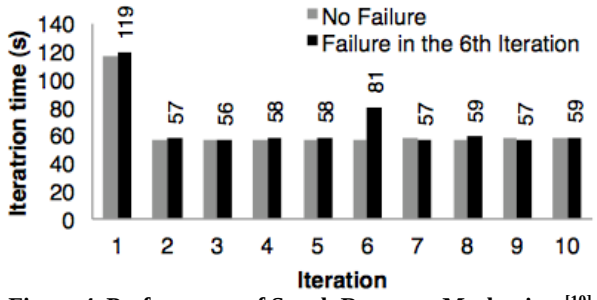

Figure 4. Performance of Spark Recovery Mechanism [10]

Notice that recovery time at the 6 iteration is less than starting over the iteration. This is a good indicator that partial reconstruction of RDD is faster than starting over the job, which is the mechanism that Hadoop uses.

Besides lineage, there are other resiliency policies used by Spark. For example, detecting struggling executor and allocating additional helper/ backup-executors to it, data locality optimization to reduce communication, and so on.

3.3.3 Generality

Another great feature of Spark RDD is the encapsulation of parallelism. Although Hadoop provides a rather friendly API for map-reduce paradigm, its execution contains massive amount of initialization and declaration. In the official API for Hadoop, to perform a WordCount, it uses 120 lines of Java code. On the other hand, Spark RDD API uses only 5 lines of code to achieve the same functionality making Spark a more suitable tool for iterative processes which are prevalent in analytics and data science applications. Besides, Spark offers customized features including manual in-memory check-point, flexible parallelism by repartitioning RDD, friendly broadcasting interfaces, and so on.

3.4 An Example of Spark Extended Interface, Spark-SQL

A compelling advantage of Spark over other distributed systems is its powerful libraries/modules that are designed to integrate algorithms or tools to solve specific problems ranging from data munging to machine learning. One of the most frequently used modules is Spark SQL, a relational processing engine based on Spark.

Key features of Spark SQL can be summarized as the follows [2]:

- Support for relational processing on RDD, external data sources, and structured databases
- Provide high performance using standard DBMS techniques
- Support for other modules like MLlib, GraphX etc.

To achieve these features, Spark SQL implements two important APIs - DataFrame, and Catalyst.

3.4.1 DataFrame

Just like DataFrame abstraction in R and Python.pandas, Spark SQL DataFrame is a collection of rows of data with the same schema. In addition to the underlying RDD structure, each partition of the DataFrame keeps an identical structured schema. The schema is the key for recording relational transformation of the DataFrame. Also, schema will be lazy-executed and optimized by *Catalyst*. This feature is different from R and Python DataFrames, because their execution of DataFrame is sequential and unoptimized for any distributed tasks.

Spark SQL DataFrame also provides friendly interface to perform relational transformations like groupby, filter, join, aggregate, and so on. Besides that, it supports RDD operations like map, row-wise transformations and other non-relational operations. Figure-5 shows an example of structured query using Spark SQL.

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

Figure-5. Count Number of Female Employee by Department Using Spark SQL

3.4.2 *Catalyst*

As most DBMS optimize the logic of structured query. Spark SQL optimizes its execution through a rule-based extensible logic optimizer called Catalyst.

Catalyst uses functional programming constructs in Scala language and a tree-like structure to recognize the logic in input query. Then it uses saved rules to optimize the logic.

The structured query in Figure-5 can be optimized by switching "where" clause and "join" clause. It will reduce the cross-reference burden by filtering the table. Simple optimizations like this will be captured by Catalyst, so user can focus on completing the job without worrying about optimizing their code [4].

The tree-like logic structure and extensible ability allows Catalyst to be easily maintained and updated. Figure-6 shows a simple logic expression x+(1+2) and how it looks like in a Catalyst tree.

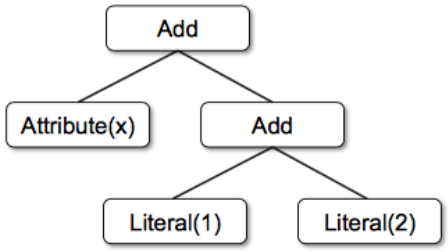

Figure-6. Catalyst Tree of x+(1+2) [11]

Each node of the tree is an immutable Scala object that can be transformed by tree class functions. This design allows the tree to be manipulated by embedded rules that manipulate nodes and corresponding logic to the tree. This process is how Catalyst optimizes an input logic structure. For example, optimization of expression in Figure-6 will result in the combination of Literal 1 and Literal 2, and the two nodes will be merged. To an additive result of (1+2=3). By adding cases and rules to the embedded transform function, Catalyst can be updated and used for customized scenario [3].

3.4.3 *Spark SQL Execution Plan*

A general process for how Spark SQL executes a user input can be represented as Figure-7.

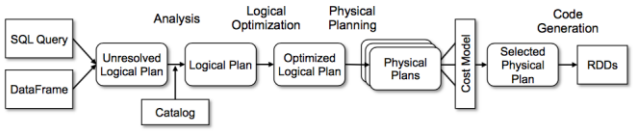

Figure-7. Spark SQL Execution Plan [11]

In the input-phase, Spark SQL takes in the query instruction and related data. Then, it uses its embedded SQL parser or HiveQL parser to parse/ covert the query into logical steps. During the optimization phase, Spark SQL constructs a Catalyst tree for the input logic, and uses its embedded rules to perform optimization transforms on the tree. Finally, during the execution phase, logic plan is transformed to physical plan that carries the code to actual execution and returns the result to user.

4. Summary

This paper gives an overview of Apache Spark ecosystem, and its key modules. Starting from its interface to machine, we listed its supported platforms of storages and resource management. Then we cover its higher-level interfaces, Spark Core and Spark SQL. For these two modules, we explain their core design choices, and provide thorough analysis of the concept and its implementation in Spark. Comparison to their predecessors is also provided based on both benchmarked tests and design choices.

In general, the paper provides an elementary-intermediate level summary to major components of Spark. Although other popular modules like Spark Streaming, MLlib are not covered, the paper could be useful to Spark application developer and intermediate Spark learners.

5. ACKNOWLEDGMENTS

Our thanks to AMPLab and Spark community for their extraordinary work.

6. REFERENCES

[1] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. NSDI 2012. April 2012.

[2] MLlib: Machine Learning in Apache Spark, Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar.

[3] Spark SQL: Relational Data Processing in Spark. Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia.

[4] High Performance Spark Best Practices for Scaling and Optimizing Apache Spark; Oreilly & Associates Inc, 2016.

[5] Karau, H. Learning Spark: O'Reilly: Sebastopol, 2015.

[6] **BDAS, the Berkeley Data Analytics Stack** https://amplab.cs.berkeley.edu/software

[7] **Spark packages**: https://spark-packages.org/

[8] **YARN,** Apache Hadoop YARN: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/index.html

[9] **SLURM,** Cluster management: https://slurm.schedmd.com

[10] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I., 2012, April. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.

[11] Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A. and Zaharia, M., 2015, May. Spark sql: Relational data processing in spar