Apache Spark – An Application of In-Memory Processing

Lukas Pietzschmann
lukas.pietzschmann@uni-ulm.de

ABSTRACT

This report covers how the main concepts and inner workings of Apache Spark work together in order to provide a performant, scalable and fault-tolerant distributed computing platform. It will also be shown how in memory processing influenced not only Spark's design, but also the way a user interfaces with the system.

1 INTRODUCTION

Apache Spark was initially designed to overcome the limitations of the classical MapReduce approach originally proposed in *MapReduce: simplified data processing on large clusters* (Dean and Ghemawat). To be able to adapt to a wider range of applications and improve the general performance by caching intermediate results, Spark implements the concept of resilient distributed datasets (RDDs).

A RDD models a read only collection of records that can easily be distributed across a cluster of nodes. Users can run parallel MapReduce-like operations on RDDs with Spark assuring correctness even in the case of a failing node. Note that the content of an RDD is in general not persisted on hard drives but only cached in RAM which can lead to great performance improvements.

Fault tolerance can be ensured by the notion of lineage. Each RDD stores exactly how it evolved over time so that it can restore its current state at any point. This concept, however, does not only apply to entire datasets. Even individual partitions can be easily recomputed with a RDD's lineage.

While MapReduce is only aware of two operations – namely `map` and `reduce` – Spark can perform a greater variety of actions and transformations. As said, we distinguish between two categories of operations:

Transformations A transformation applies a specific function to every element of the given RDD to produce a completely new dataset.

Action An action, on the other hand, does not produce a new dataset, but will narrow its source RDD down to a single result.

The sequential application of transformations will not lead to an actual computation. Instead, Spark will wait until an action forces the execution of the generated lineage graph.

The following sections will go into greater detail on how the aforementioned components and concepts interact in order to create a performant, scalable and fault-tolerant distributed computing platform.

2 IN-MEMORY PROCESSING

In-memory processing can enable great performance improvements. In *A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench* Nasim Ahmed shows the performance differences between Spark's in-memory processing and Apache Hadoop.

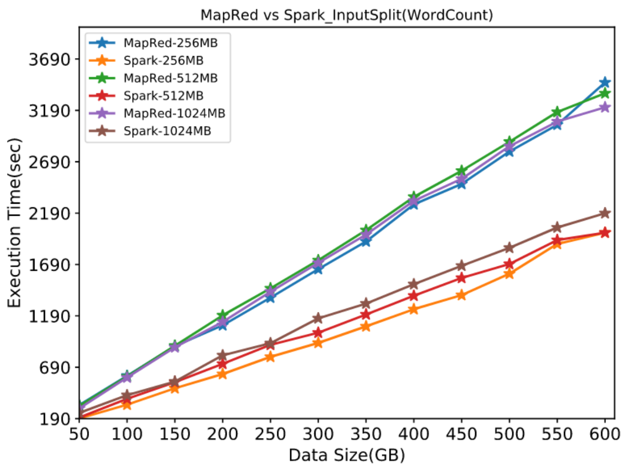

Figure 1: The comparison of Hadoop and Spark with Word-Count workload with varied input splits and shuffle tasks [2]

Figure 1 shows how the performance of both systems diverges as soon as the data load increases. The main reason Spark can achieve such a high throughput of data is its ability to cache intermediate result in memory.

How this concept influenced the design of the whole system will be discussed in the following section.

DCP, WT 2022/23, Institute of Distributed Systems, Ulm University

3 DATA MODEL

Spark provides three different abstractions to let a user interface with the system: Resilient distributed datasets and two types of shared variables [3, P. 2]. Most of Sparks code and it's abstractions are implemented in scala, but can also be accessed in languages like Python, Java or R through bindings. All examples in this report are written in Python.

3.1 Resilient distributed datasets

A RDD is a collection of immutable records that can easily be distributed over a cluster of nodes. Spark's core provides four ways to create a new RDD: (1) from file backed storage, (2) by parallelizing a Scala collection[1], (3) by transforming an existing RDD, or (4) by changing a RDD's persistence [3, P. 2]. Through additional abstractions, Spark also allows for the creation of RDDs from *e.g.* SQL tables or graphs (see section 6.1 and 6.2).

```
rdd: RDD = sc.textFile("some_text.txt")
```

Listing 1: RDD creation

Listing 1 shows how one can create a RDD from a file. Spark will then separate the given list into n partitions, with n typically being the number of nodes in the cluster. To be able to control the number of partitions directly, an RDD can also be explicitly repartitioned.

The `RDD` type enables access to a whole class of different actions and transformations. The following table shows some of the most common operations:

Type	Name
Transformation	map, flatMap, filter, reduceByKey union, intersection, distinct groupBy, join, fullOuterJoin sortBy, keys, values
Actions	reduce, fold, aggregate first, take, collect min, max, mean count, countByKey, countByValue

Table 1: Common transformations and actions on RDDs

Note that not all operations shown in Table 1 are implemented in Spark's core. Especially ByKey Operations are provided by Spark's SQL library (see section 6.1).

As each transformation returns a new RDD one can easily chain multiple transformations together.

[1]Apache Spark is mostly written in Scala, but there are bindings for other languages, too.

```
from operator import add
rdd2 = rdd.flatMap(lambda l: l.split(" ")) \
  .map(lambda w: (w, 1))                   \
  .reduceByKey(add)
```

Listing 2: Transformation chaining

The example shown in listing 2 implements the main part of the "Hello World" of distributed systems: WordCount. Instead of triggering an actual computation, Spark will keep track of `rdd2`'s lineage by generating a directed acyclic graph (DAG). Listing 2's transformation pipeline will – at least in theory – genrate the following DAG:

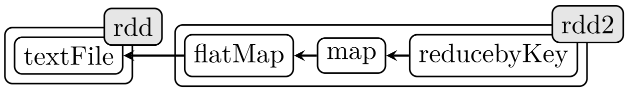

Figure 2: Representation of a lineage graph

The DAG should be read like a dependency graph. If `rdd2` has to be computed, Spark will follow all vertices until it arrives at the root node (In this case: textFile). Spark will then backtrack and compute all the remaining nodes on its way up.

To actually trigger a computation, an action has to be used.

```
result = rdd2.collect()
```

Listing 3: Forcing a computation

Spark will now heavily optimize the graph shown in figure 2 in order to minimize the number of transformations that have to be performed. In this simple example, the optimizer will likely combine all transformations needed to compute `rdd2`. We call this a logical plan [4, P. 3].

3.2 Shared Variables

Besides RDDs, there is another way to store data: Shared variables. In general, Spark will package a closure for every operation, containing every piece of local data that is necessary for the specific operation and the operation itself. Local data meaning everything that is accessed besides the RDD (*e.g.* local variables or other user defined functions). This closure is then sent along with every task (see section 4.3). Shared variables allow the user to optimize this behavior for two specific use cases.

Broadcast variables Broadcast variables are designed to improve the performance of large read only datasets. Instead of being copied with every closure, a broadcast variable will only be distributed once and is then accessible in an immutable manner [5].

Accumulators Accumulators are – as the name implies – used to accumulate data. It should be noted that they are restricted in the sense that only commutative and associative operations can be applied to them. This is due to Spark's inability to guarantee the order of execution and the requirement for accumulators to be deterministic. On the other hand, Spark tries to guarantee exactly-once semantics. This, unfortunately, falls apart as soon as accumulators are used inside transformations that have to be recalculated [6].

3.3 Fault tolerance

Fault tolerance is a key aspect for ensuring the reliability of a distributed system. As previously said, Spark's operations are deterministic in the sense that they will compute the same result even if executed multiple times[2]. Additionally, Spark is able to recompute whole RDDs or even single partitions with its notion of a lineage graph [3, P. 1].

This allows for easy recomputation of a node's calculations in case of a failure. Because of this determinism and lineage, Spark can even redistribute a slow node's workload to other nodes to ensure efficient processing.

3.4 Persistence

By default, Spark discards data from memory as soon as it was used. Another usage at a later point in time will require a recomputation of the entire dataset. In order to prevent this behavior, a user can persist a RDD in either RAM or on the disk. Persisting data won't alter its laziness, meaning that the RDD's lineage graph is not evaluated at the time of persisting. Spark will wait until an action forces evaluation and will then persist everything [3, P. 2].

Note that in memory caching is only a hint. If there's not enough RAM accessible, Spark will fall back to recomputing the RDD's content on every usage. This is mainly done to keep the system going even in tight situations [3, P. 2].

In his 2010 paper *Spark: Cluster Computing with Working Sets* Matei Zaharia wrote:

> We also plan to extend Spark to support other levels of persistence (*e.g.*, in-memory replication across multiple nodes).

[2]If user provided code involves randomness, this claim cannot hold.

Between this publication and today (Feb. 2023) this was actually implemented. Spark's persistence infrastructure supports the replication of datasets on two nodes. This can immensely help to avoid costly recomputation in case of a failure.

4 ARCHITECTURE

This chapter will delve into the internal architecture of Apache Spark, exploring the components that make up the system and how they work together. The following figure shows how the different components are connected. The subsequent subsections will go into greater detail on every important component.

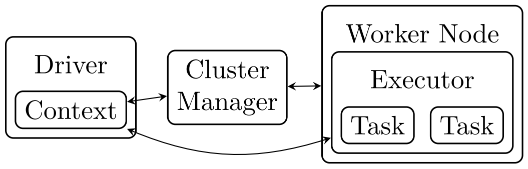

Figure 3: Apache Spark's Architecture

4.1 Driver and Context

The driver is the governing node in a cluster. It acts as the coordinator of the application and does not execute any operations[3]. The driver node is also responsible to instantiate a context, which will then run in its address space.

This context is the main entry point into Spark's API. It is accessible in code through a variable called `sc` (see listing 1). Its responsibilities include defining the application's logic, scheduling tasks, persisting RDDs, creating closures, keeping track of all lineage graphs, and communicating with the cluster manager to execute the application [5].

To make horizontal scaling trivial, the driver – and context – need to be reachable over the network in order to be able to accept new worker nodes on the fly.

4.2 Cluster Manager

The cluster manager is responsible for allocating resources, such as CPU and memory. Spark supports four different cluster managers: (1) its own standalone cluster manager, (2) Apache Mesos, (3) Apache YARN and (4) Kubernetes [5, Cluster Mode Overview]. While being extremely easy to set up, Spark's standalone cluster manager unfortunately has the severe disadvantage of not supporting a distributed file system, which is a hard requirement for many operations.

[3]Assuming there are real worker nodes.

4.3 Worker Node

While a driver node won't execute the operations it schedules[4], a worker node's sole purpose is to adhere to the commands distributed by the driver. Each worker node runs a unique instance of the Spark executor, which is the process in which all operations are executed. Each executor can run one or more tasks at the same time in separate threads.

A task is the smallest assignable unit of work. The results of the tasks are returned to the driver program for aggregation. As each task will receive its own closure, broadcast variables (see section 3.2) become especially useful if an executor contains more than one task.

5 EXECUTION OF AN APPLICATION

This section will fit all the pieces mentioned in the last pages together and will explain step by step how a specific application will be executed on an Apache Spark cluster. As an example the previously constructed Word-Count application will be used:

```python
from operator import add
rdd = sc.textFile("some_text.txt")
rdd2 = rdd.flatMap(lambda l: l.split(" ")) \
  .map(lambda w: (w, 1))                   \
  .reduceByKey(add)
result = rdd2.collect()
```

Listing 4: Word-Count

The first step Spark performs is generating the lineage graph and logic plan (see figure 2). This logical plan will then be transformed into a physical plan, which plan removes any abstractions and describes exactly which task will be run on which machine [4, P. 3]. However, the transformation from logical to physical plan is not as simple as presented here. More details are provided in Michael Armbrust's paper *Spark SQL: Relational Data Processing in Spark*. Especially section 4 and figure 3 detail this process.

Let's imagine a cluster of one driver and two worker nodes. For simplicity, we will also assume that `rdd` will be subdivided into two partitions, one for each worker. Spark will then – at least in theory – generate eight tasks[5] (as a reminder: Word-Count consists of four transformations). Those eight tasks will now get their respective closure, which will only contain the operation itself, as there's no other local data necessary. This sequence of tasks is then grouped so that each group will end with a shuffle.

5.1 Stages and Shuffles

Stages and shuffles directly influence the data throughput Spark can achieve by controlling how many tasks can be executed in parallel.

A stage refers to a group of tasks that can be executed in parallel. While tasks can overtake other tasks within the same stage, stages, on the other hand, have to be executed in order. A Stage can be terminated by two operations: (1) actions and (2) shuffles.

Shuffles are the redistribution of data between stages. They are a critical component of Spark's execution engine and are responsible for the reordering of data to ensure proper partitioned RDDs for the following stage. Shuffle operations happen – with some exceptions – typically as a result of ByKey or explicit repartitioning operations. As parallel execution has to stop in order for a shuffle to happen, it should always be tried to minimize the number of their occurrences.

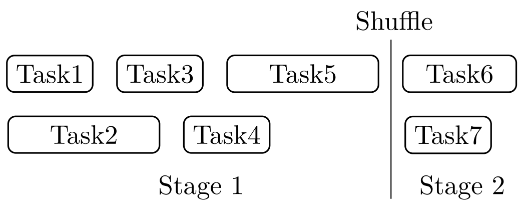

Figure 4: Shuffle terminating parallel execution

Figure 4 shows how the shuffle operation stops the parallel execution and prevents Task7 to slip into the gap after Task4. Back to the execution of listing 4's Word-Count:

To determine the number of stages Spark will fit our eight tasks into, we could look out for ByKey operations. This naive approach leads to the following stages:

(1) `textFile`, `flatMap`, `map`, `reduceByKey`
(2) `collect`

Both worker nodes will now go through stage one, executing the stage's tasks on their respective partition. Meanwhile, the driver node will wait until both workers have finished stage one and will then perform a shuffle. After shuffling, stage two's action is executed as the last operation of the application. `collect` will make `rdd2`'s content available in the driver's address space.

6 EXTENDING SPARK

While the preceding sections focused on Spark's core mechanics and abstractions, there are also four popular libraries extending Spark's default functionality.

[4]Again assuming there are real worker nodes.
[5]In practice this number will definitely be lower because, as stated previously, Spark heavily optimizes every step of the execution.

6.1 Spark SQL

Spark SQL provides – as the name implies – a convenient way to interact with SQL databases. In his paper *Spark SQL: Relational Data Processing in Spark* Michael Armbrust summarized the core idea of Spark SQL.

> Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.*, declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.*, machine learning).

Spark SQL provides two new abstractions [5]:

(1) Dataset
(2) DataFrame

While DataFrames provide a flexible and dynamic way to work with structured data, Datasets offer a more efficient and type-safe alternative for working with structured data in Spark [4, P. 1].

6.2 GraphX

GraphX provides a new graph abstraction for a directed multigraph with properties attached to each vertex and edge. In order to make working with this new abstraction easier, the library also provides additional functions targeted to calculations on graphs. Notable additions are: `subgraph`, `joinVertices`, `collectNeighbors` and `groupEdges` [5, GraphX].

6.3 MLlib

Just like GraphX, MLlib only provides new abstractions that sit on top of Spark's code. Machine learning tasks consist of mostly linear algebra and statistics, so MLlib provides functions tailored to those parts of math [5, MLlib: Main Guide].

6.4 Structured Streaming

Structured streaming uses the same concepts and ideas already known from Spark's core. In order to achieve stream processing, the stream is subdivided into smaller windows, which will then be processed like any other RDD [5, Spark Streaming].

ACRONYMS

RDD	resilient distributed dataset
DAG	directed acyclic graph

REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. (2004).

[2] Nasim Ahmed, Andre LC Barczak, Teo Susnjak, and Mohammed A Rashid. 2020. A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench. *Journal of Big Data* 7, 1 (2020), 1–18. DOI:http://dx.doi.org/https://doi.org/10.1186/s40537-020-00388-5

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, and others. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[4] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and others. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[5] Spark Team. 2023. Apache Spark Documentation. (2023). https://spark.apache.org/docs/latest/index.html

[6] Imran Rashid. 2015. Spark Accumulators, What Are They Good For? (2015). http://imranrashid.com/posts/Spark-Accumulators/