Apache Spark

Apache Spark

1 Definition

Apache Spark is a cluster computing solution and in-memory processing framework that extends the MapReduce model to support other types of computations such as interactive queries and stream processing [1]. Designed to cover a variety of workloads, Spark introduces an abstraction called Resilient Distributed Datasets (RDDs) that enables running computations in memory in a fault-tolerant manner. RDDs, which are immutable and partitioned collections of records, provide a programming interface for per-forming operations, such as map, filter and join, over multiple data items. For fault-tolerance purposes, Spark records all transformations carried out to build a dataset, thus forming a *lineage graph*.

2 Overview

Spark [2] is an open-source big data framework originally developed at the University of California at Berkeley and later adopted by the Apache Foun-dation, which has maintained it ever since. Spark was designed to address some of the limitations of the MapReduce model, especially the need for speed processing of large datasets. By using RDDs, purposely designed to store restricted amounts of data in memory, Spark enables performing com-putations more efficiently than MapReduce, which runs computations on the disk.

Although the project contains multiple components, at its core (Fig-ure 1) Spark is a computing engine that schedules, distributes, and mon-itors applications comprising multiple tasks across nodes of a computing cluster [3]. For cluster management, Spark supports its native Spark cluster (standalone), Apache YARN [4], or Apache Mesos [5]. At the core also lies the RDD abstraction. RDDs are sets of data items distributed across the cluster nodes and that can be manipulated in parallel. At a higher level,

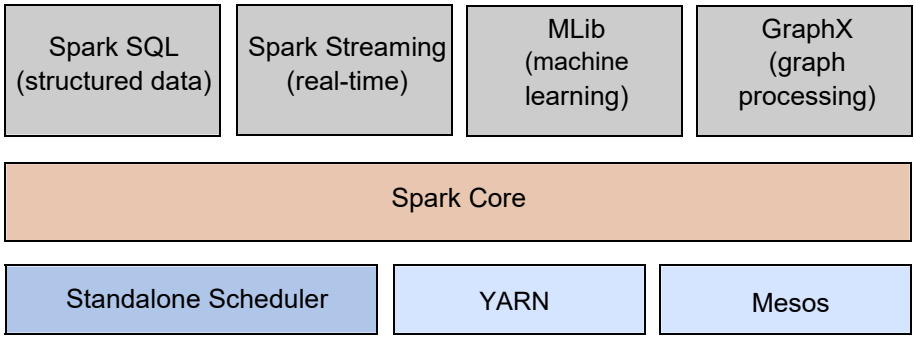

Figure 1: The Apache Spark stack [3].

```
// Create a Scala Spark configuration context
val config = new SparkConf().settAppName("WordCount")
val sc = new SparkContext(config)

// Load the input data
val input = sc.textFile(theInputFile)

// Split it into words
val words = input.flatMap(line => line.split(" "))

//    Transform into pairs and count val
counts = words.map(word =>
        (word, 1)).reduceByKey{case (x, y) => x + y}

//    Save the word count to a text file
counts.saveAsTextFile(theOutputFile)
```

Figure 2: Word count example using Spark's Scala API [3].

it provides support for multiple tightly integrated components for handling various types of workloads such as SQL, streaming and machine learning.

Figure 2 depicts an example of a word count application using Spark's Scala API for manipulating datasets. Spark computes RDDs in a lazy fashion, the first time they are used. Hence, the code in the example is evaluated when the counts are saved to disk, in which moment the results of the computation are required. Spark can also read data from various sources, such as Hadoop Distributed File System (HDFS), Cassandra, OpenStack Swift[1], and Amazon Simple Storage Service (S3)[2].

1 https://wiki.openstack.org/wiki/Swift
2 https://aws.amazon.com/s3/

2.1 Spark SQL

Spark SQL [6] is a module for processing structured data[3]. It builds on the RDD abstraction by providing Spark core engine with more information about the structure of the data and the computation being performed. In addition to enabling users to perform SQL queries, Spark SQL provides the Dataset API, which offers Datasets and DataFrames. A Dataset can be built using JVM objects that can then be manipulated using functional transformations. A DataFrame can be built from a large number of sources and is analogous to a table in a relational database; it is a Dataset organised into named columns.

2.2 Spark Streaming

Spark Streaming provides a micro-batch based framework for processing data streams. Data can be ingested from systems such as Apache Kafka[4], Flume or Amazon Kinesis[5]. Under the traditional stream processing approach based on a graph of continuous operators that process tuples as they arrive (*i.e.* the dataflow model), it is arguably difficult to achieve fault tolerance and handle stragglers. As application state is often kept by multiple operators, fault tolerance is achieved either by replicating sections of the processing graph or via upstream backup. The former demands synchronisation of operators via a protocol such as Flux [7] or other transactional protocols [8], whereas the latter, when a node fails, requires parents to replay previously sent messages to rebuild the state.

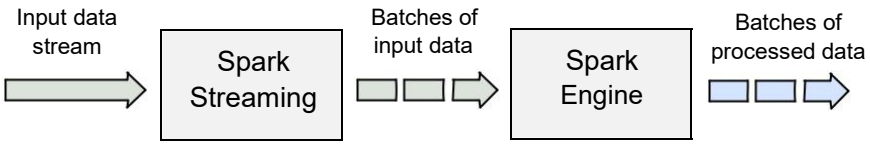

Figure 3: High-level view of discretised streams.

Spark Streaming uses a high-level abstraction called *discretised stream* or *DStream* [9]. As depicted in Figure 3, DStreams follows a micro-batch approach that organises stream processing as batch computations carried out periodically over small time windows. During a short time interval, D-Streams stores the received data, which the cluster resources then use as input dataset for performing parallel computations once the interval elapses. These computations produce new datasets that represent an intermediate state or computation outputs. The intermediate state consists of RDDs that DStreams processes along with the datasets stored during the next interval.

3 https://spark.apache.org/docs/latest/sql-programming-guide.html
4 https://kafka.apache.org/
5 https://aws.amazon.com/kinesis/

In addition to providing a strong unification with batch processing, this model stores the state in memory as RDDs that DStreams can determinis-tically recompute. This micro-batch approach, however, sacrifices response time as the delay for processing events is dependent on the length of the micro batches.

2.3 MLlib

Spark contains a library with common Machine Learning (ML) function-ality such as learning algorithms for classification, regression, clustering, and collaborative filtering; and featurisation including feature extraction, transformation, dimensionality reduction, and selection [10]. MLlib also en-ables the creation of ML pipelines and persistence of algorithms, models and pipelines. These features are designed to scale out across large computing clusters using Spark's core engine.

2.4 GraphX

GraphX [11] extends the RDD API by enabling the creation of a multigraph (*i.e.*, the property graph) with arbitrary properties attached to vertices and edges. The library is designed for manipulating graphs, exposing a set of operators (*e.g.*, subgraph, joinVertices), and for carrying out parallel computations. It contains a library of common graph algorithms, such as PageRank and triangle counting.

3 Examples of Applications

Spark has been used for several data processing and data science tasks, but the range of applications that it enables is endless. Freeman *et al.* [12], for instance, designed a library called Thunder on top of Spark for large-scale analysis of neural data. Many machine learning and statistical algorithms have been implemented for MLlib, which simplifies the construction of ma-chine learning pipelines. The source code of Spark has also grown substan-tially since it became an Apache project. Numerous third party libraries and packages have been included for performing tasks in certain domains or for simplifying the use of existing APIs.

Spark provides the functionalities that data scientists need to perform data transformation, processing, and analysis. Data scientists often need to perform ad hoc exploration during which they have to test new algorithms or verify the results in the least amount of time. Spark provides APIs, libraries and shells that allow scientists to perform such tasks while enabling them to test their algorithms on large problem sizes. Once the exploration phase is performed, the solution is productised by engineers who integrate the data analysis tasks into an often more complex business application.

Examples of applications built using Apache Spark include analysis of data from mobile devices [13] and Internet of Things (IoT), web-scale graph analytics, anomaly detection of user behaviour and network traffic for in-formation security [14], real-time machine learning, data stream processing pipelines, engineering workloads, geospatial data processing, to cite just a few. New application scenarios are presented each year during Spark's Summit[6], an event that has become a showcase of next-generation big data applications.

4 Future Directions of Research

Spark APIs shine both during exploratory work and when engineering a so-lution deployed in production. Over the years, much effort has been paid towards making APIs easier to use and to optimise; for instance, the intro-duction of the Dataset API to avoid certain perform degradations that could occur if a user did not properly design the chain of operations executed by the Spark engine. As mentioned earlier, considerable work has also focused on creating new libraries and packages, including for processing live streams. The discretised model employed by Spark's stream processing API, however, introduces some delays depending on the time length of micro batches.

More recent and emerging application scenarios, such as analysis of ve-hicular traffic and networks, monitoring of operational infrastructure, wear-able assistance [15], and 5G services, require data processing and service response under very short delays. Spark can be used as part of a larger ser-vice workflow, but alone it does not provide means to address some of the challenging scenarios that require very short response times. This requires the use of other frameworks such as Apache Storm.

To reduce the latency of applications delivered to users, many service components are also increasingly being deployed at the edges of the In-ternet [16] under a model commonly called edge computing. Some frame-works are available for processing streams of data using resources at the edge (*e.g.* Apache Edgent[7]), whereas others are emerging. There are also frameworks that aim to provide high-level programming abstractions for cre-ating dataflows (*e.g.* Apache Beam[8]) with underlying execution engines for several processing solutions. There is, however, a lack of unifying solutions on programming models and abstractions for exploring resources from both cloud and edge computing deployments, as well as scheduling and resource management tools for deciding what processing tasks need to be offloaded to the edge.

5 Cross References

- Cloud computing for big data analysis.
- Streaming microservices.
- Hadoop.
- In-Memory Processing.
- Advances in MapReduce frameworks.
- Large-scale graph processing.
- Definition of Data Streams.
- Micro-batching vs true streaming.
- Graph processing frameworks.