Apache Spark: A unified analytics engine for big data processing

Qiushi Teng
Computer Science
Portland State University
Portland, Oregon, USA
qteng@pdx.edu

ABSTRACT
This paper introduces the Apache Spark system, including its history, development, key components and how it runs applications. Also, there are two use cases of Apache Spark following the introduction, large-scale text processing pipeline with Apache Spark and structured streaming for real-time applications in Apache Spark. I mainly focus on the overall setup, the role that Apache Spark played in these applications and also its influence on the performance.

KEYWORDS
Apache Spark, Mapreduce, DataFrames, transformation, execution, Structured Streaming, pipeline

1 Introduction

Apache Spark is so far the most actively developed open source engine for parallel data processing on computer clusters[1] that is suitable in a wide range of circumstances. On top of the Spark core data processing engine, there are libraries for SQL, machine learning, graph computation and stream processing[2]. In addition, Spark runs anywhere from a laptop to a cluster of thousands of servers. Apache Spark supports multiple programming languages, such as Python, Java, Scala and R.

There are two key components in Apache Spark - a unified computing engine and set of libraries for big data[1]. The unified feature supports a wide range of data analytics tasks over the same computing engine and with a consistent set of APIs. At the same time, Spark limits its scope to a computing engine, which means that Spark only works on loading data from storage systems and performing computation on it. The second component is its libraries, including both standard libraries and external libraries, including libraries for SQL and structured data (Spark SQL), machine learning (MLlib), stream processing (Spark Streaming and the newer Structured Streaming, graph analytics (GraphX), and hundreds of other open source external libraries.

1.1 History

In order to understand Spark, it helps to understand its history. Before Spark, there was MapReduce, a resilient distributed processing framework, which enabled Google to index the exploding volume of content on the web, across large clusters of commodity servers. There were three core concepts to this strategy[2]:

1. Distributed data: Data file is splitted into data blocks when uploaded into the cluster and distributed amongst the data nodes and replicated across the cluster.
2. Distributed computation: a map function to process a key/value pair to generate a set of intermediate key/value pairs and a reduce function to merge all intermediate values by the intermediate key
3. Tolerate faults: both data and computation can tolerate failures by failing over to another node for data or processing

One year after Google published a white paper describing the Mapreduce framework[2], Apache Hadoop was created. At the time, Hadoop MapReduce was the dominant parallel programming engine for clusters[1]. Across conversations to understand the benefits and drawbacks of this programming mode, the Spark team worked on designing the API, first using functional programming language and implementing it over a new engine to perform efficient, in-memory data sharing across computation steps. Later on, by plugging the Scala interpreter into Spark, the project could provide a highly usable interactive system for running queries on hundreds of machines. Additionally, powerful new libraries are added to Spark, such as MLlib, Spark Streaming and GraphX.

1.2 Benefits of Apache Spark

The goal of the Spark project was to keep the benefits of MapReduce, while making it more efficient and easier to use. The advantages of Spark over MapReduce are[2]:

- Spark executes much faster by caching data in memory, whereas MapReduce involves more reading and writing from disk.
- Spark runs multi-thread tasks inside JVM processes, which gives Spark faster startup, better parallelism, and better CPU utilization
- Spark provides a richer functional programming model
- Spark is especially useful for parallel processing of distributed data with an iterative algorithm.

1.3 Datasets and DataFrames

Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets(RDDs). A Spark Dataset is a distributed collection of typed objects, which are partitioned across multiple nodes in a cluster and can be operated on in parallel. As shown in Figure 1, a DataFrame is a Dataset of Row objects and represents a table of data with rows and columns[2]. Although all of these abstractions represent distributed collections of data, DataFrames is the easiest and most efficient, and available in all programming languages[1]. A DataFrame consists of partitions, which is a range of rows in cache on a data node[2].

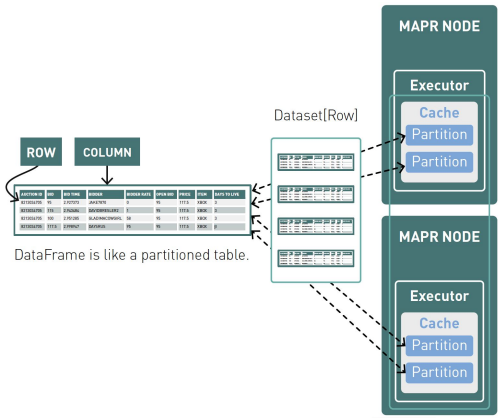

Figure 1. DataFrame in Apache Spark[1]

2 Spark's Basic Architecture

Spark is a tool for managing and coordinating the execution of tasks on data across a cluster of computers. The cluster of machines that Spark uses to execute tasks will be managed by a cluster manager. These cluster managers will grant resources to applications submitted[1]. Running applications on Spark includes dataset transformation and executions.

2.1 Dataset Transformation

Transformations create a new Dataset from an existing one. In Spark, transformations fall into two types, narrow and wide, as shown in Figure 2.

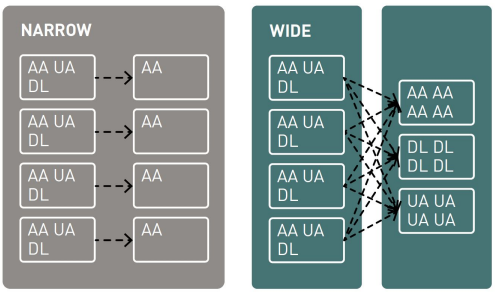

Figure 2. Narrow and wide transformations[1]

Narrow transformations do not have to move data between partitions when creating a new Dataset from an existing one. Some example narrow transformations are "filter" and "select". Also, the process of pipelining allows multiple narrow transformations to be performed on a Dataset in memory, making narrow transformations very efficient[2].

Whereas wide transformation causes data to be moved between partitions when creating a new Dataset, which is known as Shuffle. Shuffle is usually a costly operation since data is sent across the network to other nodes and written to disk, which causes network and disk I/O[2].

2.2 The Spark Execution Model

The Spark execution model can be defined in three phases: creating the logical plan, translating it into a physical plan, and then executing the tasks on a cluster[2]. The logical plan is the plan that shows which steps will be executed when an action gets applied, and triggers the translation into a physical execution plan. Then in the third phase, the tasks are scheduled and executed on the cluster.

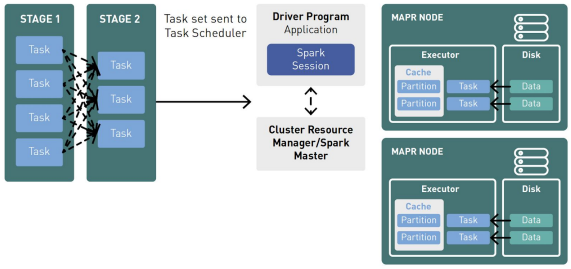

Figure 3. Task Execution on Clusters[1]

The execution in Figure 3 takes a piece of code including FILTER and GROUPBY as an example. As shown in Figure 3, when the tasks are scheduled and executed on the cluster. The scheduler splits the tasks into stages by transformation types. Stage 1 is the narrow transformation and stage 2 is the wide transformation. Each stage performs the computations in parallel. Then the scheduler submits the stage task set to Task Scheduler for launching tasks via a cluster manager. This process can happen multiple times when new datasets are created.

3 Applications

In addition to the general concepts of Spark, there is a lot of research done using Spark. Here are two specific application user cases, for which I will mainly focus on the system setup, what role Spark performs in the research and how the performance was affected.

3.1 Large Scale Text Processing Pipeline

The paper, a large scale text processing pipeline with Apache Spark, evaluates the performance of Apache Spark for a data-intensive machine learning problem, policy diffusion detection across the state legislatures in the United States. This paper provides an implementation of this analysis workflow as a distributed text processing pipeline with Spark dataframes and Scala API. The pipeline is designed to consist of five stages, including the stage of document pair similarity calculation, which is considered to be the most compute and shuffle intensive part of the pipeline. Apache Spark is at the core of the implementation, closely integrated with Apache Hadoop ecosystem, using Hadoop file formats and HDFS storage, which can improve data locality by means of replication. The policy diffusion analysis involves map, filter, join, and aggregateByKey transformations, and all-pair similarity calculations require intensive shuffles.

As discussed in the performance evaluation section, external shuffle service is used to observe an improved stability of memory-intensive jobs with more executor containers, which is 40 in this analysis. Efficiency is measured as a function of the number of executor containers for different dataset sizes[3], by varying the data from different numbers of states, from 2 to 10. In Figure 4, it shows that, in general , the efficiency in the high-executor region is improved as the sample size increases, which means the larger the dataset, the less efficiency decreases.

An intensive shuffle is identified to be the main cause of efficiency decrease. If the specified threshold for in-memory maps used for the shuffle is exceeded, the

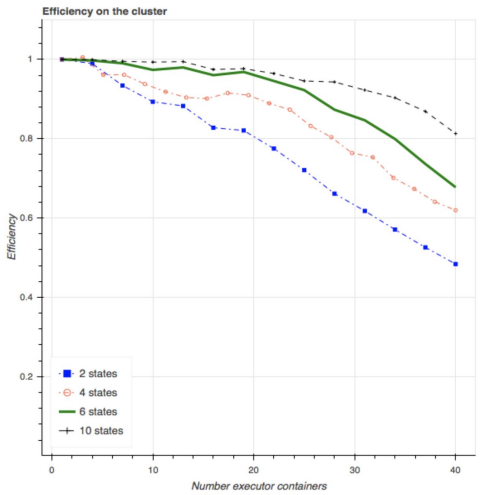

Figure 4. Efficiency for different dataset sizes[3]

contents will begin to spill to the disk. As in figure 5, if the processing time is measured as a function of the number of virtual cores in the Spark clusters, the processing time of the increased shuffle memory is lower than that of the default shuffle memory. In conclusion, the proposed framework allows efficient calculation of all-pairs comparison, and is potentially applicable to problems of finding similar items[3].

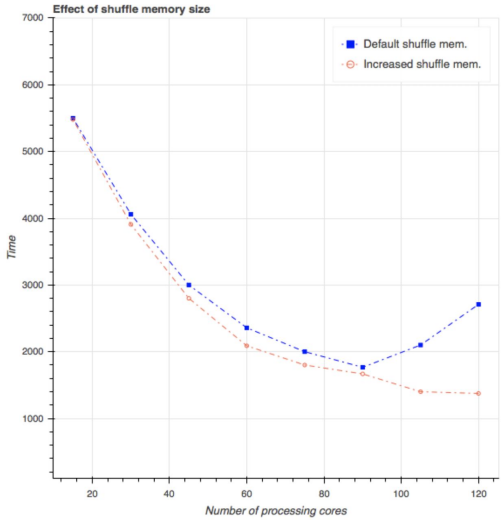

Figure 5. Processing time with different shuffle memory[3]

3.2 Structured Streaming: a declarative API for real-time applications in Apache Spark

Structured Streaming implemented in this paper[4] includes several main components, as shown in Figure 6.

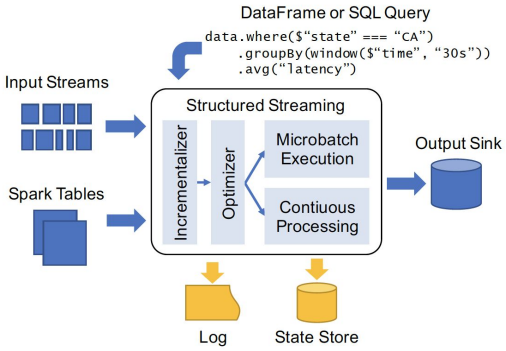

Figure 6. The components of Structured Streaming[4]

Input and Output. Structured Streaming connects to a variety of input sources and output sinks for I/O. In where, input sources must be replayable to allow the system to re-read recent inputdata if a node crashes, and output sink must support idempotent writes to ensure reliable recovery.

API. User program Structured Streaming by writing a query against one or more streams and tables using Spark SQL's batch APIs: SQL and DataFrames.

Execution. Structure Streaming execution optimizes, incrementalism, and begins executing it. By default, a microbatch model is used, which supports dynamic load balancing, rescaling, fault recovery, and straggler mitigation by dividing work into small tasks. Additionally, there is a continuous processing mode based on traditional long-running operators.

In both cases, there are two forms of durable storage to achieve fault tolerance. The first one is a write-ahead log that keeps track of which data has been processed. The second one is a large-scale state store to hold snapshots of operator states for long-running aggregate operators.

Operational Features. The two forms of durable storage allow users to achieve several forms of rollback and recoveries. First of all, an entire Structured Streaming application can be shut down or restart on new hardware and running applications tolerate node crashes, additions, and stragglers automatically. Also, a user can manually roll back the application to a certain point in the log and redo the part of computation. Executing with micro batches allows quick catch up with input data if the load spikes or if a job is rolled back.

There are four production use cases mentioned in this paper, which have different customer workloads that leverage various aspects of Structured Streaming and their internal use case.

3.2.1 Information Security Platform

The information Security Platform allows over 100 analysts to scour through network traffic logs to quickly identify and respond to security incidents, as well as to generate automated alerts.

There are two challenges in bundling this platform, building a robust and scalable pipeline and allowing effective queries for fresh and historical data. Structured Streaming has several advantages and make this possible. The first advantage is that Structured Streaming has the ability to vary the batch size, which allows the developer to build a pipeline that deals with spike in data load, failures and code update. The second advantage is that one stream can join other streams, including historical tables, which greatly simplifies analysis. Finally, using the same system for streaming , interactive queries marks iteration and alert deployment faster.

3.2.2 Monitoring Live Video Delivery

Media companies can use Structured Streaming to compute quality metrics for live video traffic and interactively identify delivery problems. Live video delivery can be very challenging due to the fact that network problems can severely disrupt utility. The companies collect real-time data, perform ETL (an Extract, Transform, and Load job) operations and aggregations using Structured Streaming. This allows interactive queries of fresh data to detect and diagnose quality issues.

3.2.3 Other Production Use Cases

There are other use cases of Structured Streaming. Gaming companies use Structured Streaming to monitor latency experiences of players. Similar to the live video use case, network performance is essential for user experience. Latency logs are collected for a variety of streaming analysis.

Another use case is cloud monitoring at Databricks. They have been using Apache Spark since the start of the company to monitor cloud service, and understand workload statistics. Their experiences with Structured Streaming shows that it successfully addresses many challenges. They were able to achieve restartability,

simpler fault recovery, updates and rollbacks to fix historical results.

In this paper, they evaluate the performance of this system using the Yahoo! Streaming Benchmark[4].They state that structured streaming achieves high performance via Spark's SQL's code generation engine and can outperform Apache Flink by up to 2X and Apache Kafka Streams by 90X.

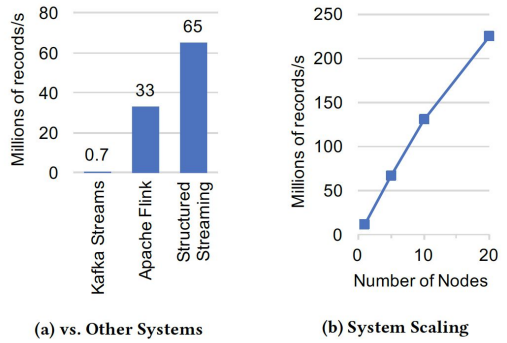

Figure 7. Throughput results on the Yahoo! benchmark[4]

As shown in Figure 7, Spark Streaming took less time to perform the query, and they believed that the performance comes solely from Spark SQL's built in execution optimization. In addition, this system presents a linear scalability between the size of the cluster and the processing time. They also measured the latency of continuous processing. In a map job of reading, the continuous processing achieves much lower latency without a significant drop in throughput.

4 Conclusion

Although streaming systems are still difficult to use, operate and integrate into large applications[4], it is very impressive how efficient and powerful Spark is in different applications, in particular Spark Streaming. By learning more and more about the Spark system, I think there must be a lot more complicated tasks that can be better done using Spark and I will continue my exploration in Spark.

ACKNOWLEDGMENTS

We would like to thank Professor Kristin Tufte, the instructor for this course, for providing advice, ideas, and resources for this project.

REFERENCES

[1] Apache Spark Under The Hood https://tanthiamhuat.files.wordpress.com/2019/01/apache-spark-under-the-hood.pdf

[2] Getting Started with Apache Spark from Inception to Production https://mapr.com/ebook/getting-started-with-apache-spark-v2/assets/Spark2018eBook.pdf

[3] Large-scale text processing pipeline with Apache Spark https://arxiv.org/pdf/1912.00547.pdf

[4] Structured Streaming: https://databricks.com/wp-content/uploads/2018/12/sigmod_structured_streaming.pdf