Reading Assignment 2

October 5, 2018

Anna Hedstrom annaheds@kth.se
Sandra Pico Oristrell sandrapo@kth.se

1 Motivation

1.1 Spark

The paper *"Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing"* by Zaharia et al., (2012) [2], proposes a new abstraction (read: data structure) called resilient distributed datasets (RDDs). The RDDs are designed to empower programmers with the ability to perform parallelisable, fault-tolerant, in-memory computations on clusters. The authors demonstrate clear intent of making the RDDs general-purpose - with a broad set of use cases in mind - they encapsulate the abstractions into a larger, collective computing system - named Spark.

Interestingly, in the paper, the authors point out that no alternative computing framework at the time would suffice for a general application of important emerging tasks such as iterative- and graph algorithms (i.e., data reuse) and more complex (interactive) data mining. Recognising that widely adopted computing systems (think: MapReduce) could not live up to the increased requirements (not to mention that they also carried with it substantial overheads, excessive writes to disk and even a neglect to leverage distributed memory), the authors convincingly motivated the creation of Spark.

1.2 Spark SQL

The paper *"Spark SQL: Relational Data Processing in Spark"* by Armbrust et al., (2015) [1], declares an important contribution to the Spark library, namely the integration with relational processing e.g., SQL. In the authors own words, they present the approach as an "evolution" of Spark it-self - by offering richer APIs and optimiser(s); e.g., Catalyst, while at the same time leveraging the benefits that comes with Spark such as distributed memory.

In particular, the motivation for building Spark SQL arose from the need to combine two seemingly disjoint systems, that is relational- and procedural systems. The authors recognised a challenge many users face - that they increasingly want to perform advanced analytics (think: machine learning) but that the preferred language to do so, i.e., declarative queries, fundamentally lack expressiveness. Interestingly, based on experience with Shark (SQL-on Spark), they went on to build an API where users could benefit from both worlds, namely Spark SQL.

2 Contribution

2.1 Spark

We see Spark as both an advancement and extension of the prevailing cluster computing frameworks at the time. Not only does Spark accommodate both complex query processing as well as smart optimisations for data reuse, it also includes a clear reduction of the number of read-write to disk. We find that Spark provides many contributions and have listed the most important ones below.

- **Many operators**. The RDD abstractions (that make up Spark) can be manipulated using a relatively rich set of operators. These covers two categories: lazy transformations such as *map, filter* and *join* that builds the logical plan (chain of objects capturing the lineage of a RDD) for defining a new RDD as well as actions which triggers the computations to return a value or write to external storage such as *collect*, *take* and *reduce*.
- **Fault-tolerance.** Efficient data recovery is provided by so-called lineage. Rather than storing actual data, it logs (its lineage), which essentially is the transformations used to build a data set. Upon failure, only the lost partitions of the RDDs need to be recomputed (no need for full rollback).
- **Low-level control.** Since RDD is a lower-level API, it also comes with a lot of power.

This includes *custom partitioning of data*, which enable optimisation of data placement, as well as *control of persistence* (i.e., the choice of in-memory or out-of-core processing) that lets users keep intermediate results in memory by caching data. By this we minimise costly efforts to fetch data from disk.

- **Supports advanced data flows.** It further supports advanced data flow graphs, which implies any composition/ mixture of data sources, operators and/or data sinks.

2.2 Spark SQL

As said before, Spark SQL lets its users enjoy the benefits (and simplicity) of relational processing, while simultaneously accomodating for complex analytics. In particular, we find that Spark SQL makes two main value propositions:

- **DataFrame API**. Via the declarative DataFrame API, one can perform relational "SQL-like" operations on both external-data sources and built-in RDD abstractions. Given that the operations are lazily evaluated, one can perform relational optimisations.
- **Catalyst**. Spark SQL includes an general framework optimiser called Catalyst, which is an extensible query optimiser that use features of Scala programming to make it easy to complement with new data -types and sources such as semi-structured data (e.g., JSON) as well as optimisation rules and other user defined functions. In this way, Catalyst help to support the wide range of algorithms and data sources, espeically useful in the context of machine learning.

3 Solution

3.1 Spark

Very briefly, the **RDDs** are immutable collections of records ("objects") which are spread across clusters in forms of partitions (atomic pieces of information). RDDs are all created via deterministic operations on either data storage or other RDD and always comes in one of two types; either *generic*- or *key-value* type.

Moreover, we have learned that the programming model of Spark is the **language-integrated API** that exposes the RDDs (and their transformations invoked using methods). Naturally there is more to it. More broadly, Spark applications consist of a driver process and a set of executor processes.

- **The driver process** controls the Spark application and can similarly be seen as the "heart" of the Spark application. It sits on a node in the cluster and runs the `main()` function. The driver process is responsible for three main tasks, namely i) the maintenance of meta information, ii) response to input or a users program and iii) analysis/ distribution and scheduling of work across executors.
- **The executor processes** are responsible for two main tasks, namely i) the execution of assiged code and ii) reporting of the computation's state back to to the driver.

3.2 Spark SQL

As mentioned before, Spark SQL runs on top of Spark. The interfaces of SQL can be accessed through command-line, using JDBC/ODBC or through the DataFrame API which is the main abstraction in Spark SQL. Similar to RDD, it is a distributed collection, yet contains rows with a homogeneous schema, making it equivalent to a table in a relational database. Unlike RDDs, the schemas of the Dataframes are kept track of for optimisation purposes. Then, using the mentioned implementation, the solution of Spark SQL gives the following:

- **Integration.** Query structured data inside Spark programs, using DataFrame API or SQL.
- **Uniform data access.** One can connect to different data sources in the same way.
- **Standard connectivity.** Connecting through JDBC or OCBC based on the server mode, providing the possibility to connect to other intelligence tools.

At the same time, as part of the solution, we have the Catalyst optimiser, that works mainly from a tree data type which in turn is composed of node objects. Here, each node contains the node type and specific number of children (which can be zero), where rules can be applied to manipulate the objects.

Very briefly, we find that Spark SQL uses the Catalyst's general tree transformation in four different phases (see figure page 6):

- Analysing logical plan to resolve references,
- Logical plan optimisation,
- Physical planning,
- Code generation to compile parts of the query.

4 Discussion

The authors of Spark (and Spark SQL) claim wide applicability, and at most instances they provide reasoning or evidence for such statements. However, as with all computing frameworks, there are always drawbacks. We discuss a chosen few below.

- **No support for asynchronous updates.** First, we note the Spark is less suitable for applications requesting asynchronous updates (that are often web-related). Although the authors themselves point out that RDDs (i.e., disregarding Spark Streaming extension) is better qualified for batch analytics, we still consider this a limitation given how it reduces applicability.
- **Real-time processing is not "real".** Second, another drawback is the fact that there is no "true" support for real time processing of live data. Given that the RDDs are processed and returned in batches of predefined interval, Spark will only be as close to "near" real-time processing (i.e., "micro-batch processing").
- **Optimisations are manual (i.e., error-prone).** Third, while we already listed the benefits for RDDs of being a low-level language, we also note the manual control that is required to partition and cache data sets. For inexperienced programmers, it is therefore fairly easy to get it wrong and so, "sub-optimise" in efforts to "optimise".
- **Dependence on other file systems.** Fourth, another issue is that Spark relies on other file management systems and thus requires additional integration with other platforms/ applications, e.g., HDFS. This process could be error prone and costly.
- **Limited understanding of data structure.** While we know that lazy evaluation of RDDs can be useful from a query optimisation perspective, it also has its drawbacks. We learn that it implies that "the engine does not understand the structure of the data" in the RDD (i.e., Java/Python objects) or even the semantics of the user functions (i.e., arbitrary code).
- **No support for predicate pushdown for key-value stores**. Finally, to conclude, in the paper, the author states that they also are planning to add predicate pushdown for key-value stores, as this is current a limitation that other tools like, `Cassandra` or `HBase`, are already supporting.

References

[1] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion 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*, pages 2–2. USENIX Association, 2012.