Spark SQL: Relational Data Processing in Spark

ACM SIGMOD Conference **May 2015**
Open source project: http://spark.apache.org./
Summary By: OUMOUSS EL MEHDI (M2 Data & Knowledge)

I. Introduction

Big data problems are very challenging, especially when we take a look to the different processing techniques, data sources and storage formats that we have to deal with. Yet, to face these challenges, users until now were always forced to choose between two options: either use powerful procedural APIs such as MapReduce, but that offers a low-level interface and requires manual optimization; or use relational APIs, that are more user friendly like Pig, Hive and Dremel, however not sufficient solutions for big data applications.

These two classes of systems - relational and procedural - have until now remained largely disjoint. However, it is observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. This paper describes the effort to combine both models in Spark SQL, a major new component in Apache Spark.

II. Spark SQL

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on the experience from Shark, a first effort to build a relational interface on Spark. Figure 1. shows where Spark SQL as component fits in the Spark framework. 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). The gap between these two models is bridged through two contributions:

I. A declarative **DataFrame API**, that is able to perform relational operations on both external data sources and Spark's built-in distributed collections.

II. **Catalyst**: a highly extensible optimizer, to support the wide range of data sources and algorithms in big data.

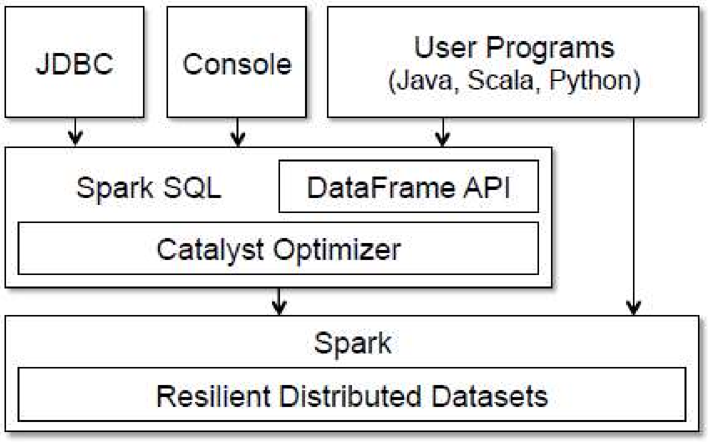

Figure 1: **Interfaces to Spark SQL, and interaction with Spark.**

III. DataFrames

A DataFrame is a distributed collection of rows with a homogeneous schema. It is the equivalent of a table in a relational database. Spark SQL's DataFrame API provide a tighter integration between relational and procedural processing. DataFrames offers several benefits that make them more convenient and more efficient than Spark's procedural API, to just state a few:

- DataFrames store data in a columnar format that is significantly more compact than Java/Python objects. This makes it easy to compute multiple aggregates in one pass using a SQL statement, something that is difficult to express in traditional functional APIs.

- DataFrames can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. In this sense, Dataframes can be viewed as RDDs of Row objects. Yet oppositely to RDDs, DataFrames keep track of their schema and can be manipulated with various relational operations (e.g. where, groupBy) that lead to more optimized execution and enables constructing complex pipelines that mix different analytics.

- Spark DataFrames evaluates operations lazily. This means that a DataFrame object represents a logical plan to compute a dataset, but no execution occurs until a special "output operation" is called, this enables rich optimizations.

- The ubiquitous use of DataFrames made it much easier to expose all algorithms in all languages, namely for MLlib's algorithms in SQL.

Finally, DataFrame operations in Spark SQL go through a relational optimizer, Catalyst.

IV. Catalyst

To implement Spark SQL, a new extensible optimizer was designed. Catalyst, was built using Scala's features, this made it easy to add composable rules, control code generation, and define extension points. Catalyst's extensible design had two purposes:

- Make it easy to add new optimization techniques and features to Spark SQL, in response to the various problems that can be encountered in the big data world.

- Enable external extension of the optimizer, which allows further support for new data types or new rules specific to a data source.

At its core, Catalyst contains a general library for representing trees and applying rules to manipulate them. Catalyst transforms a tree representing an expression in SQL into an AST (abstract syntax tree) for Scala code to evaluate that expression, and then compile and run the generated code. This transformation follows a framework of four phases, as shown in **Figure 2.** :

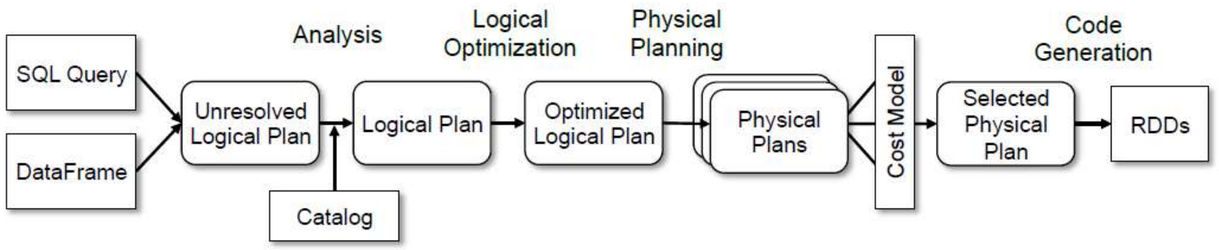

Figure 2. Phases of query planning in Spark SQL

(1) Analyzing a logical plan to resolve references
Spark SQL begins with a relation that is computed either from a dataframe or an abstract syntax tree (AST) returned by a SQL parser. This relation may contain unresolved attribute references or relations. Spark SQL uses Catalyst rules and a Catalog object that tracks the tables in all data sources. To resolve these attributes, an "unresolved logical plan" tree is built with unbound attributes and data types, then several rules are applied.

(2) Logical plan optimization
In this phase several standard rule-based optimizations are applied to the logical plan. Among these optimization, we have: constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules.

(3) Physical planning
Giving a logical plan, Catalyst generate multiple physical plans and compare them using a cost model. At the moment, cost-based optimization is only used to select join algorithms. Richer cost-based optimization are intended to be implemented in the future.

(4) Code generation to compile parts of the query to Java bytecode.
The final phase of query optimization consists of Java bytecode generation to run on each machine. Catalyst relies on a special feature of the Scala language, quasiquotes. Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. This made code generation much simpler and resulted in interesting execution speed ups.

Using Catalyst, a variety of features tailored for the complex needs of modern data analysis have been built:

- **Schema inference algorithm (for semi-structured data ex. JSON)**

In big data environments, Semi-structured data is very common thanks to its various characteristics. Yet, it is cumbersome to work with in a procedural environment. So In order to save users the pain of resorting to solutions like ORM libraries, Spark SQL added a JSON data source that automatically infers a schema from a set of records in one pass. The JSON file is treated as a table where fields are accessed by their path. Similar inference, is intended in future works for CSV and XML files.

- **Machine learning (data) types**

Another Spark utility, MLlib, Spark's machine learning library, is based on dataframes as a way to exchange data between different stages of a pipeline. A useful abstraction, as machine learning workflows consist of several transformations on data, such as feature extraction, normalization, dimensionality reduction, and model training. This representation make it easy to change parts of the pipeline or to search for tuning parameters at the level of the whole workflow.

- **Query federation to external databases**

Data pipelines often combine data from heterogeneous sources that can reside in different machines or locations, as a result, it is necessarily to query them in a smart way that isn't costly and that reduce the amount of data transferred. . In this perspective, Spark SQL data sources leverage Catalyst to push predicates down into the data sources whenever possible. Future works are looking to add predicate pushdown also for key-value stores such as HBase and Cassandra.

V. Evaluation

Spark SQL's performance have been evaluated based on two dimensions:

1. **SQL query processing performance**

Spark SQL was compared against Shark and Impala, using the AMPLab big data benchmark. The benchmark contains four types of queries with different parameters performing scans, aggregation, joins and a UDF-based MapReduce job. The results show that Spark SQL is substantially faster than Shark thanks to code generation in Catalyst and generally competitive with Impala.

2. **Spark program performance**

First Experiment:
Two implementations of a Spark program that does a distributed aggregation were compared based on computation time. The first one is hand written using map reduce function in the Python API, while the second one is written using the DataFrame API. Besides being much more concise, the DataFrame version of the code outperforms the hand written Python version by 12 times. This is due to the DataFrame API that performs efficiently in matter of execution (code generation) as well as the Catalyst's optimization at both logical and physical levels.

Second Experiment:
The DataFrame API can also improve performance in applications that combine relational and procedural processing, by letting developers write all operations in a single program and pipelining computation across relational and procedural code. In this experiment, done over a synthetic dataset, two stages are pipelined. The first consist of data filtering and the second on a count operation. Two approaches are being compared here. The first, where the pipeline is implemented using a separate SQL query followed by a Scala-based Spark job. And a second approach where the pipeline is combined using the DataFrame API. The DataFrame-based pipeline does not only avoids costs of saving intermediate results, but it is easier to understand and operate, also improves performance by 2 times.

VI. Conclusion

Spark SQL an evolution of both **SQL-on-Spark** and of Spark it-self, offers rich APIs and optimizations while keeping the benefits of the Spark programming model. Spark SQL has managed to reach more or less the goals set for it, namely:

- Support relational processing both on Spark's native objects and external data sources.
- Offer high performance.
- Support different data sources (i.e. semi-structured data), with extension points for new data types.
- Enable extension with advanced analytics algorithms such as graph processing and machine learning.

Depending on evaluations as well as feedbacks from users, Spark SQL has succeeded to mix relational and procedural processing through simple and efficient pipelines without losing in matter of computation speed.