Student Name: Nikhil Doifode
Student ID#: 112714121

Review of "Spark SQL: Relational Data Processing in Spark"

Big Data Applications and Systems have evolved in recent times and require combination of different processing techniques, process input data from different sources like stream or storage. Application developers want the flexibility provided by the procedural programming languages and run some analytics on the data for machine learning purposes and also do some relational data processing like Shark but with more efficiently. Previous solutions like Shark forced users to choose between relational processing and procedural programming interface because it needed incoming data to be expressed with the combination above two which made it more complex and remained disjoint. So Spark SQL library was created in Spark stack to fix these issues to meet following goals:

1. Support relational data processing on RDD's and on external data sources
2. Provide High Performance
3. Provide easy method to support new data sources and semi-structured data like JSON
4. Extend it's application to support advance applications like Machine Learning and Graph Processing

Spark SQL solves this problem by providing following key ideas:

1. Spark SQL provides Dataframe API's that can perform operation on relational data from external sources and Spark's in built distributed data store called Datasets.
2. Spark SQL also introduced extensible optimizer called Catalyst to make it easy to add data sources, optimization rules and support data from different sources and use it for applications like Machine Learning.

Beside the extensibility and flexibility, Spark SQL shows performance improvement over other previous solutions like Shark and Impala in various query processing like Scan, Aggregation, Join and UDF. Also Dataframe API's allowed Spark SQL to outperform handwritten Python versions by 12 times and Scala version by 2 times as shown in the following figure.

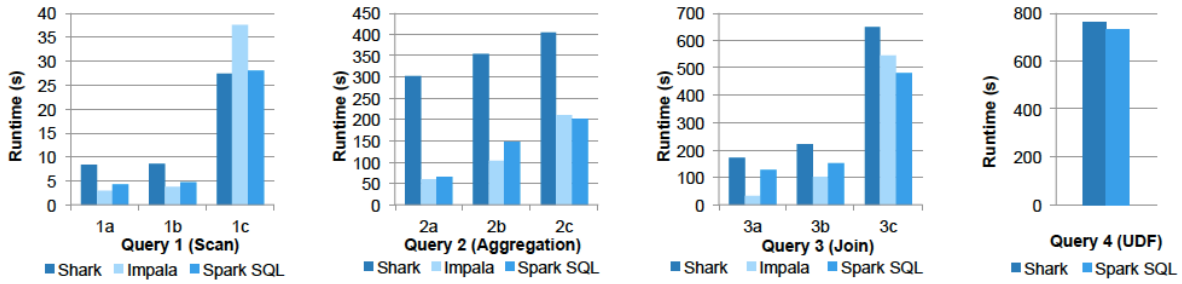

Figure 8: Performance of Shark, Impala and Spark SQL on the big data benchmark queries [31].

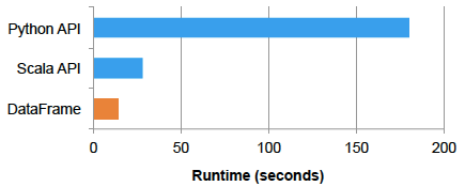

Figure 9: Performance of an aggregation written using the native Spark Python and Scala APIs versus the DataFrame API.

Strengths:

1. Support new Data sources easily like external DB and semi-structured data.
2. High Performance for DBMS queries
3. Spark SQL's extension to Advance algorithms for analytics in Machine Learning and Graph Processing
4. Make Developer's job easier with mixture of procedural (E.g.: MapReduce) and SQL applications.

Weakness:

1. Cost-based optimization in Catalyst is only used to select join algorithms: for relations that are known to be small, Spark SQL uses a broadcast join, using a peer-to-peer broadcast facility available in Spark. However, costs can be estimated recursively for a whole tree using a rule. Thus, richer cost-based optimization should have been implemented
2. Spark SQL, uses Schema Inference algorithms like used in Twitter with JSON data type of RDDs of Python objects, as Python is not statically typed so an RDD can contain multiple object types. Hence, similar inference for CSV files and XML is missing
3. Predicate pushdown for key-value stores such as HBase and Cassandra, which supports limited forms of filtering is missing
4. No support for transactional table Type [1]

Questions:

1. Difference between Spark SQL and existing Databases (Relational & Non-Relational)?

Ans:

Spark SQL vs MySQL: Support for Distributed Data processing which is essential in big data applications.

Spark SQL vs Shark: Better Performance and flexibility to choose data source and interface

Spark SQL vs Postgres SQL: Spark SQL has SQL like Data Definition Language (DDL) and Data Manipulation Language (DML) statements and support for user defined functions

Spark SQL vs Non-Relational Databases: Support for Relational Data queries on large data sets in distributed manner.

2. How does Spark SQL work?

Ans: Spark SQL is built on top of Apache Spark, where data is distributed over different machines and organized into collections called Resilient Distributed Datasets (RDD's) with support of MapReduce like functions. Spark provides Dataframe API, which allows relational operations on both external data sources and RDD's. They provide support for developer to call relational operations such as projection, filter, join, and aggregation.

Additionally, Spark SQL provides in-memory cache to reduce memory footprint, and offers user-defined functions.
Along with this Spark SQL comes with extensible query optimizer called Catalyst. As can be seen from the Spark SQL API architecture diagram below, the Logical Plan corresponding to the Dataframe will be optimized and converted by Catalyst, and finally become the corresponding RDD Physical Plan, and then submitted to Spark for calculation.
Catalyst supports both rule-based and cost-based optimization. The main data type of Catalyst is a tree made up of node objects, where each node has a node type and zero or more children.

3. What is the role of catalyst?

Ans: Catalyst is an extensible query optimizer. As can be seen from the Spark SQL API architecture diagram below.

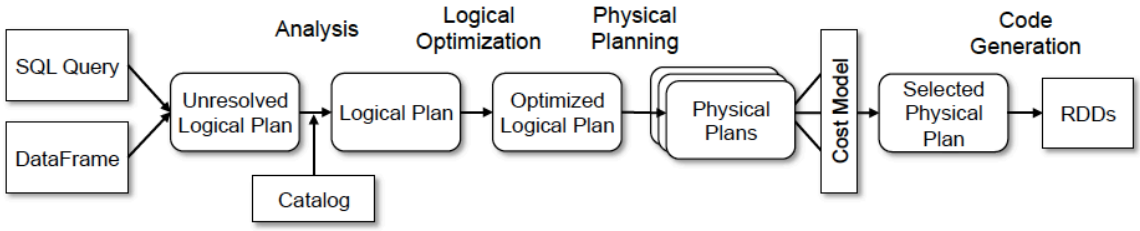

Figure 3: Phases of query planning in Spark SQL. Rounded rectangles represent Catalyst trees.

Catalyst's query optimization for Dataframe can be divided into the following phases:

a. Analyzing Logical Plan to resolve references: Information in the Dataframe object constructed by the user through the Dataframe API layer used to construct an AST (Abstract Syntax Tree) that computes the operation of the corresponding data.
b. Logical Plan Optimization: The logical optimization phase applies standard rule-based optimizations to the logical plan. These include constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules.
a. Physical Planning: Catalyst takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine and selects a plan using a cost model.
b. Code generation to compile parts of the query to Java Bytecode: Catalyst uses the Quasi quotes function provided by Scala to allow the programmatic construction of abstract syntax trees (AST's) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. Catalyst is used to transform a tree representing an expression in SQL to an AST for Scala code to evaluate that expression, and then compile and run the generated code.

References:

[1] https://forums.databricks.com/questions/3093/does-spark-sql-support-hive-transactions-yet.html
Spark SQL: Relational Data Processing in Spark – Michael Armbrust, ….