Md Majid Jahangir | SBU ID - 112077145
AMS 560 - Big Data Systems, Algorithms and Networks

Assignment 2 (Paper Reading)
Spark SQL: Relational Data Processing in Spark

Motivation : Any big data applications require a data pipeline that could be expressed efficiently using a combination of both procedural code and relational queries. However, the earliest systems designed for such pipeline such as MapReduce had just procedural programming interface, whereas later, new systems such as Pig, Hive, Shark, etc, provided new relational interfaces to big data, and would take advantage of declarative queries to provide better optimizations. However, the goal of any big data application cannot be achieved by either using relational or procedural code. There are applications which requires transactions of data between several data sources and can be best achieved using relational declarative queries whereas, the same relational queries cannot help in advanced analytics such as machine learning or graph processing, and they can be achieved best using procedural model. Hence, this paper tries to combine both the models and propose a component in Apache Spark, Spark SQL, that can seamlessly intermix both the procedural and relational declarative query model.

Prior Solutions: Prior solution was a relational interface built on Spark called Shark. It modified the Apache Hive system to run on Spark and implemented columnar processing over Spark engine. Even though it performed well on Spark, it had several limitations such as it could only be used to query external data stored in Hive and hence was not useful for data processing inside Spark program to make data pipelines. Also, since it was built on top of Hive, it was just tailored for MapReduce and hence it became difficult to extend it to implement new data types that could also be used for advanced analytics like machine learning or graph processing.
Besides, there were other solutions not specific to Spark, such as Pig, Hive, Dremel which provided relational interfaces to big data, however just the relational declarative queries would not help much in advanced analytics as much as procedural model would, hence even those systems did not serve fully to real big data applications.

Key Ideas of Proposed Solution / How does Spark SQL work (Key Questions) : Spark SQL closes the gap between relational and procedural model by intermixing both and by running as a library on top of Spark. It provides SQL interfaces which can be accessed using JDBC/ODBC or Dataframe API that performs relational operations on external data sources and Spark's built-in distributed collections. Dataframe is the main abstraction in Spark SQL and is a distributed collection of rows with same schema which can be manipulated using Spark's procedural API or relational APIs that allow optimizations. They can be constructed from external tables/data sources or from existing native Java/Python Spark's objects enabling relational operations such as where, groupby, join, etc, or procedural Spark APIs such as map. Such dataframes evaluates operations lazily only on invoking output operations, thus allowing to perform optimizations in intermediate stages.
Spark SQL uses a nested data model based on Hive for tables and dataframes supporting all major SQL data types as well complex data types such as arrays, structs, maps, unions, etc. Infact even such complex data types could be nested together to create more powerful data types. It is only because of this that the Spark SQL has been able to model data from several sources such as relational schema, Hive, JSON, and Java/Python/Scala native objects. This makes Spark SQL interoperable with procedural Spark code since dataframes can be constructed directly against objects of Python/Scala/Java and Spark SQL can infer its schema using reflection.
Besides, Spark SQL allows user defined functions (UDF) to be defined and registered inline enabling users to implement UDFs that not only work on scalar values but also on complex data types such as database tables. These UDFs can use several Spark's APIs within themselves, thus exposing advanced analytics functions to users.

Also, an extensible query optimizer called Catalyst has been designed based on functional programming constructs in Scala. It contains a library for representing trees (to represent expressions or SQL queries) and applying rules (using pattern matching functions of Scala - functions converting tree to another tree) to manipulate the trees. These trees are used to perform analysis, planning and runtime code generation enabling optimizations.
In Spark SQL, catalyst is used in four phases, namely -

1. Analysis - Spark SQL takes a syntax tree from a SQL parser or a dataframe object, which may contain unresolved attributes or relations. Spark SQL uses catalyst rules and a Catalog object that maintains all the tables of all data sources and resolves these unresolved attributes by looking them up in the catalog and mapping the attributes.
2. Logical Optimization - Once the attributes and relations have been resolved in the previous stage to result into a logical plan, standard rules-based optimizations such as predicate pushdown, null propagation, boolean expression simplification, etc, are applied to it to optimise the plan logically.
3. Physical Planning - In this phase, the logically optimized plan is used by Spark SQL to generate one or more physical plans using operators that match Spark execution engine. Using a cost model, one of the physical plan is selected. This phase can also perform some rule-based physical optimizations such as pipelining filters into one Spark map operation.
4. Code generation - This is the final phase of query optimization which generates Java bytecode of the query to be run on each machine

Finally, some of the other key ideas of Spark SQL has been discussed under the strengths section for brevity.

Performance : The performance of Spark SQL was measured on two aspects - SQL query processing and Spark program performance. The performance was evaluated and compared against Shark and Impala using AMPLab big data benchmark which generates a workload comprising four categories of queries performing jobs like scan, aggregation, joins and a UDF based map-reduce. The above mentioned queries except for UDF based map-reduce had varying parameters which grouped those 3 queries into 3 sub-categories, where first sub-category (numbered ‘a’) was the most selective query and the third sub-category (numbered ‘c’) was the least selective and processed more data. It was observed that Spark SQL was substantially faster than Shark in all of those queries because of code generation in Catalyst that reduces overhead; whereas Spark SQL was competitive with Impala, where for some queries Impala would perform marginally better and for other queries Spark SQL would perform better but marginally.
Also, program written using Dataframe API would outperform the same program written using native Spark code in Scala by 2 times and the same program written in native Python by 12 times because of optimizations achieved in Dataframe API through code generation. The code written using Dataframe API would be more concise as well.
Also, Dataframe API improved performance by 2 times in applications that needs both relational and procedural processing since all the operations could be written in a single program and pipelined across relational and procedural code. Unlike prior solutions, Dataframe API would avoid saving the result of SQL query to a file as an intermediate result to be passed to the procedural Spark job as a result of pipelining procedural and relational jobs.

What's the role of Catalyst? (Key Question): Catalyst is an extensible query optimizer whose role is to optimise the queries or relations. It contains a library to represent queries or relations as trees and applies several rule-based optimization techniques in 4 different phases to optimize the query. The four phases has been explained clearly in the “How does Spark SQL work” section. It also enables the framework to be extended with new data sources, including the support for semi-structured data such as JSON; user defined functions and user defined

types for domains such as machine learning and to add new optimizations techniques or rules that pertains to big data.

What's the difference between Spark SQL and existing database (relational and non-relational) [Key Question]:

1. Unlike many traditional DBMSes, Spark SQL provides a support for complex data types in the query language and the API and it also supports user defined types.
2. Unlike many traditional DBMSes, Dataframes in Spark SQL enables not only the same relational operations like SQL but also allows a developer to implement them as domain-specific language (DSL) because of their integration in a full programming language. It is because of this, developers can also use control structures such as if statements and loops to structure their code.
3. Spark SQL made API analyze logical plans eagerly so as to check whether the data types in queries are appropriate or whether the column names used in expressions exists in tables or not. This enabled Spark SQL to report error as the developer types an invalid line or invalid expression, unlike traditional DBMSes, which performs this check only on execution.

Strength :

1. To handle some of the challenges in big data environments, some additional features added to Spark SQL are -
    a. Semi-structured data such as JSON is quite popular in large scale environments and it is very cumbersome to parse such data in procedural model, hence a JSON data source has been added to automatically infer a schema from records.
    b. Integration with Spark's Machine Learning library - MLlib, Spark's machine learning library introduced a new API that will use Dataframe of Spark SQL which resembles machine learning pipelines, an abstraction in ML libraries such as scikit-learn. Dataframes provide a compact and flexible format which allows multiple type of fields to be stored for each record. This makes it easy for developers to build complex pipelines and helped Spark SQL to be adopted even more among developers.
2. It allows developers to leverage benefits of relational processing and procedural models by tightly integrating them through a declarative DataFrame API that integrates with procedural code.
3. It allows support for SQL data types such as boolean, integer, timestamp, etc, and also complex data types such as structs, arrays, maps,unions, etc. It also supports user-defined type with the help of Catalyst.
4. It provides SQL interfaces which can be accessed using JDBC/ODBC or Dataframe API that performs relational operations on external data sources. It also supports various external data sources such as JSON, Hive table, native Java/Python/Scala objects, etc.
5. The dataframes can not only be operated on using relational domain-specific language (DSL), but also can be registered as temporary tables in the system catalog and queried using SQL.
6. Unlike traditional DBMSes, it is easier to construct a concise and declarative statements using control structures like if statements or loops.
7. Error reporting while developing is easier because of eager analysing of logical plans. This enabled Spark SQL to report error as the developer types an invalid line or invalid expression, unlike traditional DBMSes, which performs this check only on execution.
8. It allows interoperability with procedural Spark code, as it allows user to construct DataFrames directly against RDDs of objects native to the programming language such as Python/Java/Scala.

9. Catalyst has been designed as an extensible query optimizer which enables the framework to be extended with new data sources, including the support for semi-structured data such as JSON; user defined functions and user defined types for domains such as machine learning. Besides, it also enables us to add new optimizations techniques or rules that helps in optimizing SQL queries.

Weakness :

1. No support for timestamps fields in AVRO file format.
2. There is no support for Hive transactions.
3. It does not support fixed length character type data and hence tables containing character type field cannot be accessed by Spark SQL.
4. Optimizations of queries in Spark SQL are limited because Spark engine does not understand the structure of data in RDD or the semantics of user functions

Follow Up Works -

1. Optimizing Big-Data Queries Using Program Synthesis [3]: Matthias Schlaipfer, Kaushik Rajan, Akash Lal, Malavika Samak - This paper proposes a system called Blitz that can synthesize efficient query specific operators using automated program reasoning. The system uses static analysis to identify sub-queries as potential targets for optimizations.
2. Stargate: a data source connector based on spark SQL [4]: Yuzheng Tao, Gang Wu, Yi Kang - This paper presents a data source connector called Stargate which provides a set of framework for different storage engines to connect to Spark SQL engine. The proposed system helps in coonecting computing engines to data sources on different storage engines and help the engine in understanding and adapting the storage engine to improve computational efficiency.
3. CloudMdsQL: querying heterogeneous cloud data stores with a common language [5]: Boyan Kolev, Patrick Valduriez, Carlyna Bondiombouy, Ricardo Jimenez-Peris, Raquel Pau, Jose Pereira - This paper presents a cloud multi-datastore query language (CloudMdsQL) and its query engine. It is a functional SQL like query language capable of querying multiple heterogeneous data store, both relational and NoSQL.

References -

1. Spark SQL: Relational Data Processing in Spark - Michael Armbrusty, Reynold S. Xiny, Cheng Liany, Yin Huaiy, Davies Liuy, Joseph K. Bradleyy, Xiangrui Mengy, Tomer Kaftanz, Michael J. Franklinyz, Ali Ghodsiy, Matei Zahariay
2. Tutorialspoint, https://www.tutorialspoint.com/apache_spark_online_training/spark_sql_advantages_and_disadvantages.asp
3. ACM Digital Library, https://dl.acm.org/citation.cfm?id=3132773
4. ACM Digital Library, https://dl.acm.org/citation.cfm?id=3184088
5. ACM Digital Library, https://dl.acm.org/citation.cfm?id=2989976