Spark SQL : Relational Data Processing in Spark

Pushkar Sinha

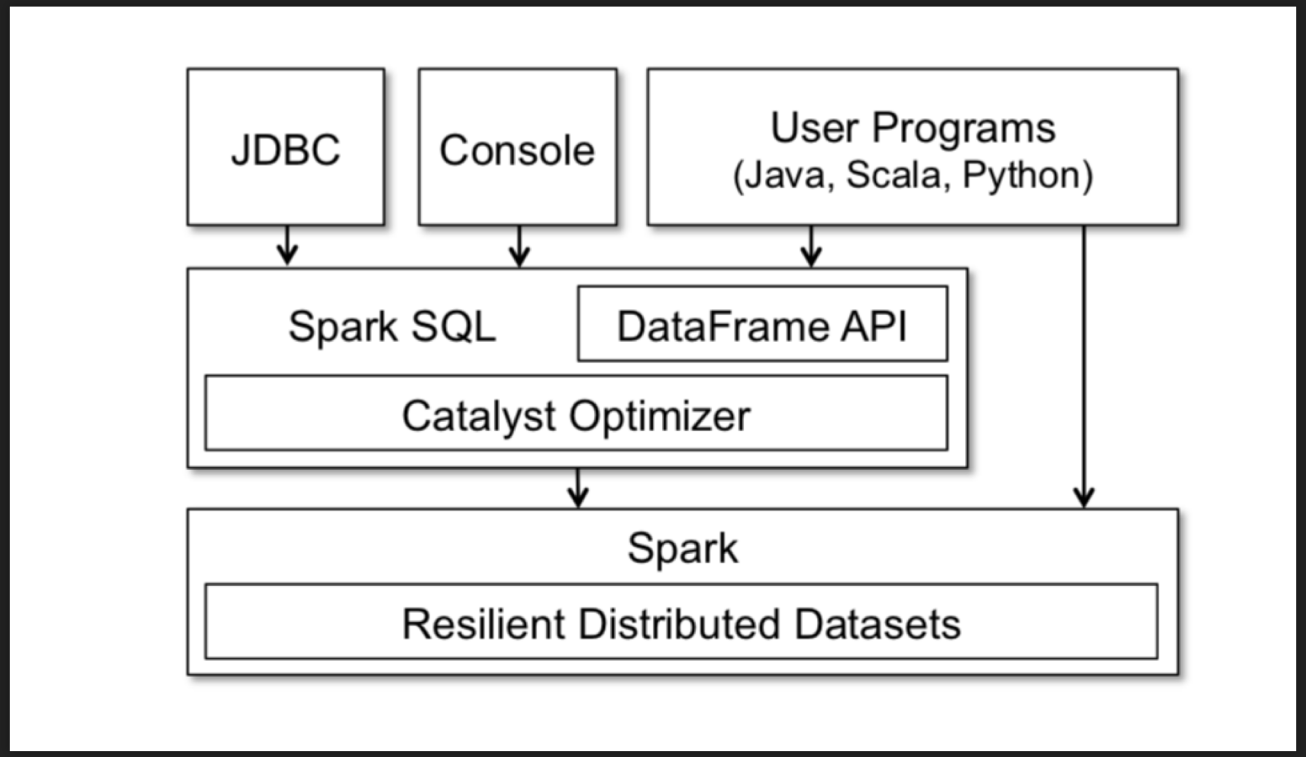
Why Spark SQL

- To give a relational based approach to data processing programs.
- Have a data structure anytime to contain a full table to be manipulated along the code.

Two Pillars of Spark SQL

- DataFrame API : All RDD manipulations are in the format ”map”, “filter”, “reduce” but with dataframe, we don’t need them for SQL like operations.
- Catalyst Optimizer : Adding new optimization techniques to spark SQL and add data source specific rules for filtering and aggregation over the data source with the support of new data types.

Two Pillars of Spark SQL



DataFrame API

- You feel as if the whole table is a data structure and one can manipulate and save it in an SQL way while programming:
  - employees
    .join(dept, employees("deptId") === dept("id"))
    .where(employees("gender") === "female")
    .groupBy(dept("id"), dept("name")) .agg(count("name"))
- Easy conversion from RDD to dataframe and vice versa while inferring schema.

DataFrame API contd…

- In memory caching : Save any computed dataframe (columns-wise compressed) in the memory such that it can be retrieved much faster if needed again.
- UDFs : Not only with scalar values but after being registered it can be used taking a whole table as input (also via JDBC/ODBC) , e.g. MADLib

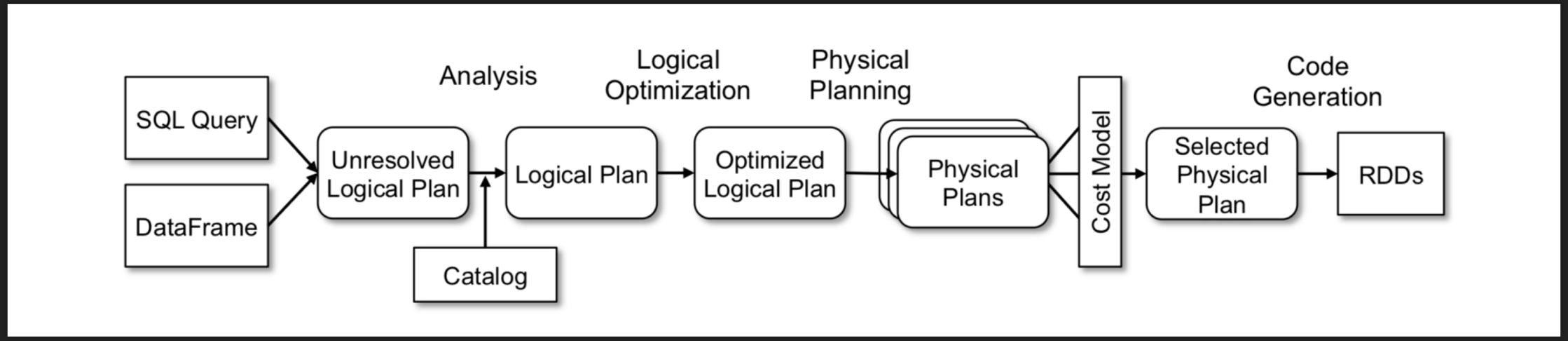
Catalyst Optimizer

- A framework for optimized execution of SQL operations and storage of data.
- Starts with a tree (AST) and a set of rules applied over 4 different phases (analysis, logical optimization, physical optimization, code generation).

Catalyst Optimizer

- Lets consider the rules in the form (LHS:RHS).
- And everytime we compare the LHS in the tree nodes and replace them with the RHS of rules. (matching takes place here recursively with replacements)

Catalyst Optimizer



Catalyst Optimizer (Analysis)

- After the SQL query has been given, it gets converted to an “unresolved logical plan” tree.
- In analysis phase the resolving of tables, columns, giving unique id to same value columns.

Catalyst Optimizer (Logical Optimization)

- Different types of logical operations like constant folding, predicate pushdown, projection pruning, null propagation, etc.
- We can add and keep on adding rules to be applied for the logical optimization of the operations on respective nodes to be optimized.
- As an example, there is an attempt to optimize an aggregate with small precision decimals where the decimal is converted to long and after aggregation converted back to decimals.

Catalyst Optimizer (Physical Planning)

- Cost based : suppose you ask for a join and one of the tables is really small , a rule be picked up to do a brodcast join. Further we can embed our own rules.
- Rule based : pipelining projections or filters.

Catalyst Optimizer (Code Generation)

- Code generation helps in figuring out the final code. With all the rules to convert into direct code based on pattern matching.

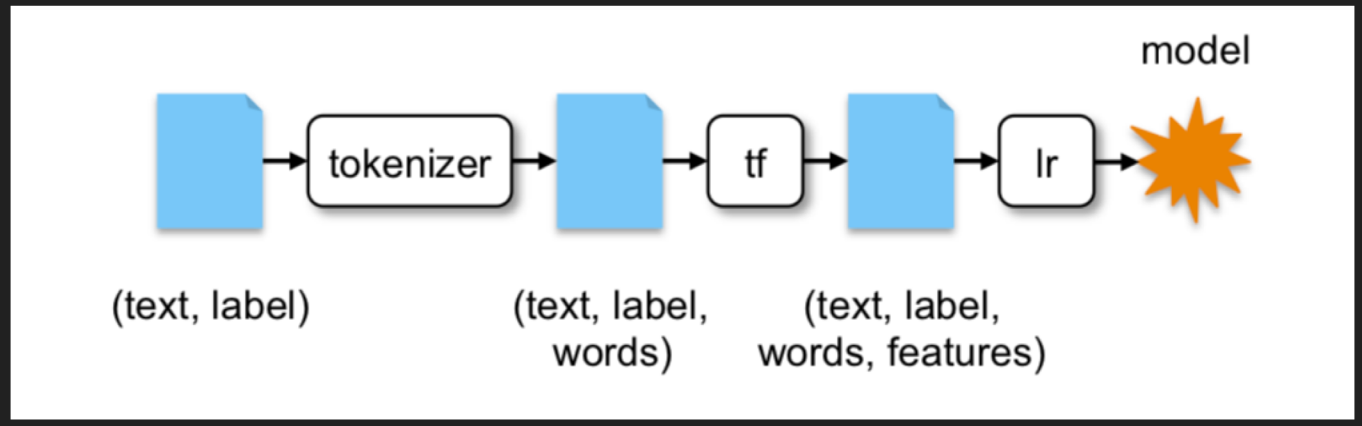
Catalyst Optimizer (Extension Points)

- Data sources : User has the freedom to define different data sources but has to implement methods like createRelation, tableScan etc. Some already implemented data sources are CSV, AVRO, Parquet, JDBC ds.
- UDT : User can create new datatype to be accepted by spark by extending UserDefinedType class.

MLLib

- Easy conversion from JSON to Dataframe and RDD.
- Even DDL with DML can be executed using SparkSQL and get RDD or dataframe as the return value.

MLLib (A text mining example)



- data = <DataFrame of (text, label) records>
- tokenizer = Tokenizer() .setInputCol("text").setOutputCol("words")
- tf = HashingTF() .setInputCol("words").setOutputCol("features")
- lr = LogisticRegression() .setInputCol("features")
- pipeline = Pipeline().setStages([tokenizer, tf, lr])
- model = pipeline.fit(data)