COMPSCI 590S Systems for Data Science **Fall 2016**

Lecture 23: Hybrid Systems (SQL + Big Data)

Lecturer: Emery Berger *Scribe(s): Molly McMahon, Parth Gandhi*

23.1 Issues with MapReduce Systems

- It is complicated to express many problems with MapReduce, leading to confusing and hard-to-write code.
- MapReduce requires manual optimization.

23.2 Previous Attempts to Mitigate These Issues

- FlumeJava - attempts to solve the problem of manual optimization by using lazy evaluation. It is a library solution which optimizes MapReduce plans and minimizes the number of operations before execution.
- DryadLinq - Language-level solution which allows users to "jam" SQL statements into other programming languages. This is meant to make operations easier to express.
- ORM (Object Relational Mapper) - allows users to operate on tables in a native language (e.g. Java) by mapping tables to object representations and vice-versa. This approach requires SQL queries to be represented as strings, which makes it vulnerable to attacks like SQL injection, and also makes it more difficult and prone to type errors. Eg. Java Hibernate.

23.3 Spark

Spark is a cluster computing framework for Scala that provides MapReduce functionality and distributed data storage. High performance and fault tolerance are provided using RDDs (resilient distributed datasets), which are kept in memory (to the extent possible) during computation. RDDs operations are evaluated lazily, and fault tolerance ensured by logging operations to disk. To recover a lost RDD, Spark only needs to replay the operations recorded in the log on the original RDD.

23.4 Spark SQL

Spark SQL is similar to Hive, in that it provides a SQL interface on top of a MapReduce system. However, while Hive's API focuses only on providing SQL functionality, Spark SQL provides other functionality in addition to SQL. The main features and advantages of Spark SQL are as follows:

23.4.1 Data Types

Spark can handle structured, unstructured, and semi-structured data, and does not require a schema (these types of data are defined below).

- Structured Data - requires a set schema
- Semi-Structured Data - usually does not have a set schema, can consist of a mixture of structured and schema-less data
- Unstructured Data - No schema, e.g. key-value stores

Example: JSON(Javascript Object Notation)

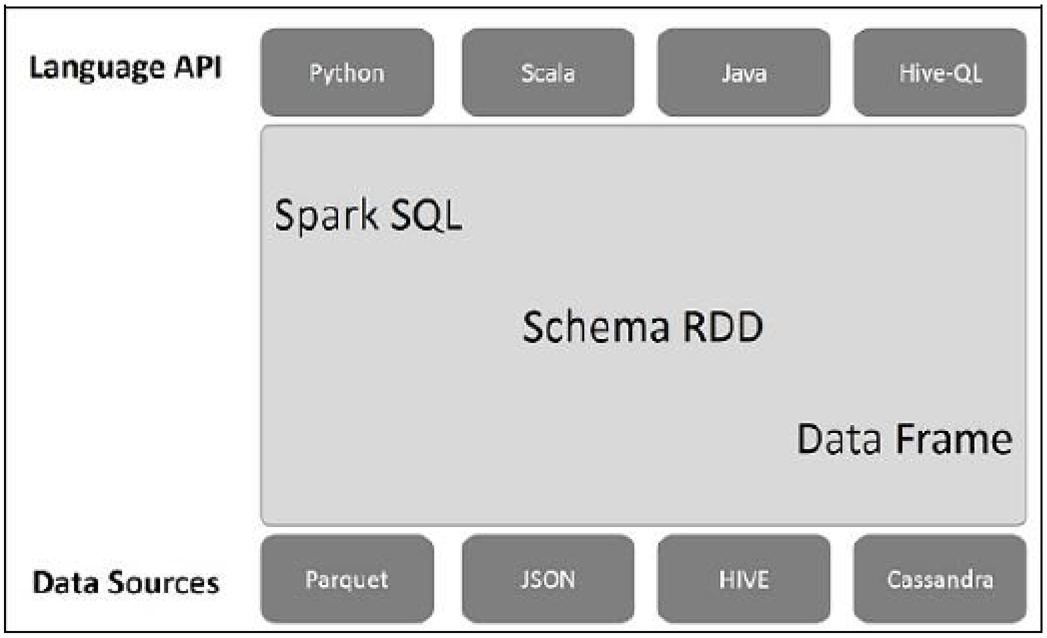
- Like a Javascript object, but allows nesting
- More flexible as a data export format than CSVs
- Can ”serialize” Javascript (or other types) of objects by converting to JSON text
- Can have ”schematas,” or tree representations of nested data

Spark SQL provides automatic schema inference - it can ”guess” the schema of data stored as JSON objects, CSVs, or collections of Java/Python/etc. objects by inspecting the types of the entries' attributes.

23.4.2 Integration of Spark and SQL

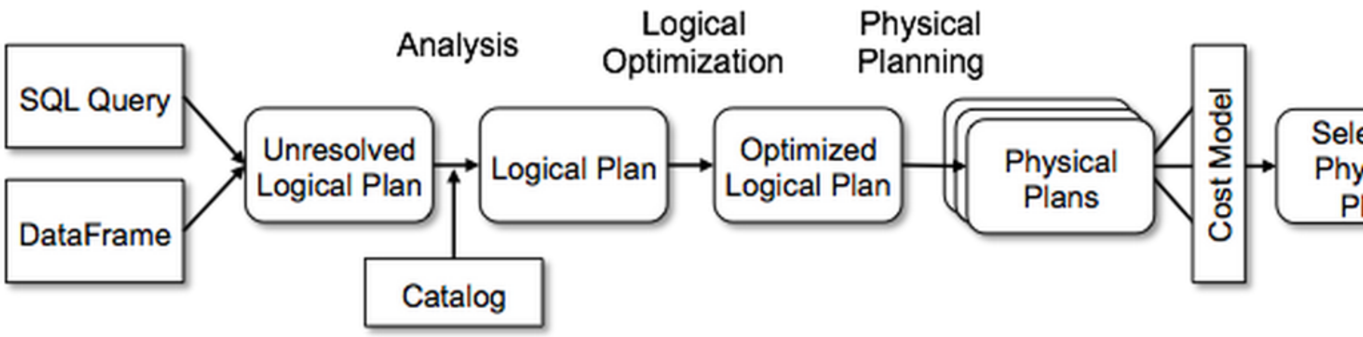
Instead of replacing Spark code with a SQL API (the way Hive replaces Hadoop code with SQL), Spark allows users to intermix SQL-like operations with Spark code. All parts of the resulting plan (both Spark and SQL code) are optimized. Spark SQL allows users to define User Defined Functions (UDFs), which can be written inline in Java, Scala, or Python. These can be registered with the SQL engine and optimized along with the rest of the plan.

Spark SQL provides the DataFrame API, which basically corresponds to a SQL table. These DataFrames can be created from Spark RDDs or from external sources, and can be passed to and from Spark code/libraries.



23.4.3 Spark SQL Optimizations(Catalyst)

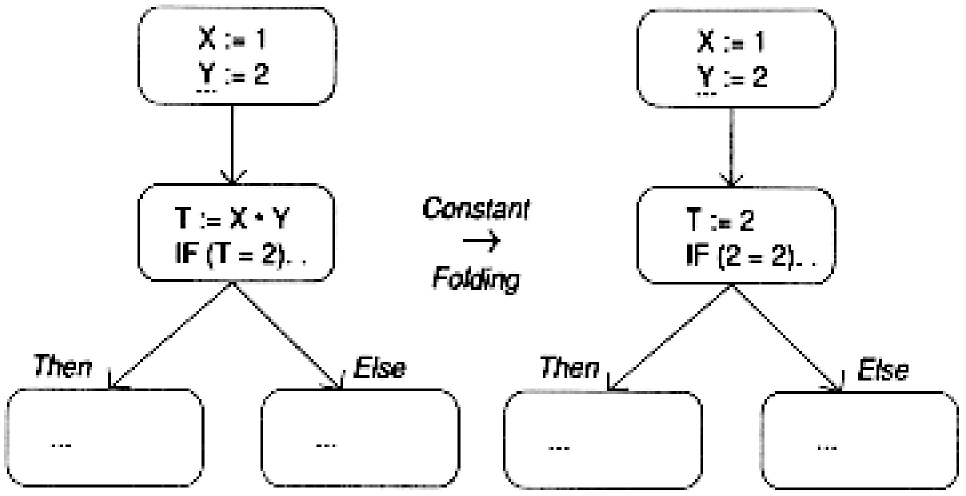
Spark SQL operations on DataFrames are represented as abstract syntax trees (ASTs) of operations. Operations can be expressed with Scala quasiquotes, which Scala automatically parses into ASTs. These trees can then be optimzed via ”tree surgery.”



23.4.3.1 Constant Folding

Constant Folding combines multiple operations into one operation. For example, if we the following two add operations, constant folding would combine them into one add operation.

EXAMPLE 13 Global constant propagation



This can be optimized still further with an algebraic substitution and a dead code elimination. The result is shown in Example 14.

23.4.3.2 Pattern Matching

Given a recursive definition of a tree, Scala pattern matching is used to replace certain subtrees with new subtrees. The optimizer iterates over the tree, applying predefined pattern matching optimization rules until the tree reaches a "fixed point" and stops changing.

23.4.3.3 Predicate Pushdown

This query optimization takes predicate statements (e.g. "WHERE" clauses) and pushes them closer to the data source, so that less irrelevant data is retrieved as part of the SQL query.

23.4.3.4 Projection Pruning

This query optimization moves the attributes selected closer to the data source, again to avoid retrieving unecessary data that won't actually be used.

23.4.4 Quasiquotes

Quasiquotes are a neat notation that lets you manipulate Scala syntax trees with ease:

scala> val tree = q”i am a quasiquote ”

tree: universe.Tree = i.am(a.quasiquote)

Every time you wrap a snippet of code into q”...” quotation it would become a tree that represents given snippet.