COMPSCI 590S Systems for Data Science **Fall 2017**

Lecture 18: SparkSQL

Lecturer: Emery Berger *Scribe(s): Shamanth Kumar, Yichi Zhang*

In this lecture, we first discussed about SparkSQL which is a new module in Apache Spark that integrates relational processing with Sparks functional programming API and its advantages. Then we talked about some basic concepts of complier.

18.1 Disadvantages of using HiveQL

1) Hive creates UDF(User Defined Functions) and UDAF(User Defined Aggregation Functions) as Java. This operation is a big ask for Java as Java is mostly static.
2) UDF is a black box and it cannot be optimized. 3) HiveQL creates a bridge between SQL and MapReduce. It doesnot integrate them.
4) It has more disk I/O.

18.2 SparkSQL

Spark SQL is similar to Hive, in that it provides a SQL interface on top of a MapReduce system. However, while Hives API focuses only on providing SQL functionality, Spark SQL provides other functionality in addition to SQL.

18.3 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. The main advantage of using Spark as compared to Hive is that it reduces the amount of I/O and hence is faster compared to Hive.

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 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.

Spark is written in a programming language called Scala. Scala runs on JVM(Java Virtual Machine). Examples of other JVM languages are Clojur and Kotlin. Scala is more advanced as compared to Java. Scala builds DSL's(Domain Specific Language).

18.3.1 Macros in Scala

Macros in Scala are turned into Abstract Syntax Trees(AST) which can generate code. There are no macros in Java.

18.4 Complier

We discussed basic concepts of complier. Compiler is used to transform source code (usually high level programming language e.g. Java, C) to target code (e.g. assembly language).

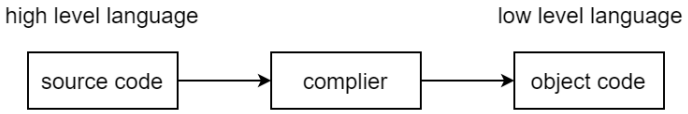

Figure 18.1: Compiler

Compiling has four basic stages: lexical analysis, parsing, complier logic, code generation.

1. Lexical analysis : Compiler scans the source code and converts characters to meaningful tokens. Common tokens include reserved word in certain language, such as "new", "if", "while" etc. in Java. For example, it takes 'w', 'h', 'i', 'l', 'e' into integer 15 which represents keyword "while".

2. Parsing : Also called syntax analysis, converts sequences of tokens into formal grammar, abstract syntax tree (AST). A statement can be parsed as followings:

   statement ::= varname assign expr
   expr ::= number|expr binary_op expr|unary_op expr|expr|varname

   For example, given a statement $x = 9 + (y * 6)$:

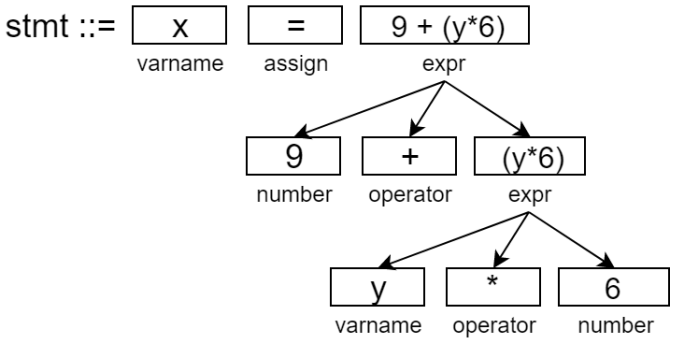

Figure 18.2: Example of an AST

3. Complier logic and code generation: Compiler then optimizes AST, generates intermediate representation(IR) and finally genrerates assembly code. Several optimization methods:

   (a) Constant folding in tree. For example, const operator const $\rightarrow$ const.

(b) Inlining. Make small function inlined.

(c) Dead code elimination.

(d) Liveness analysis.

(e) Strength reduction. Replacing expensive operations by cheaper operations. For example, replacing division 2 with right shift.

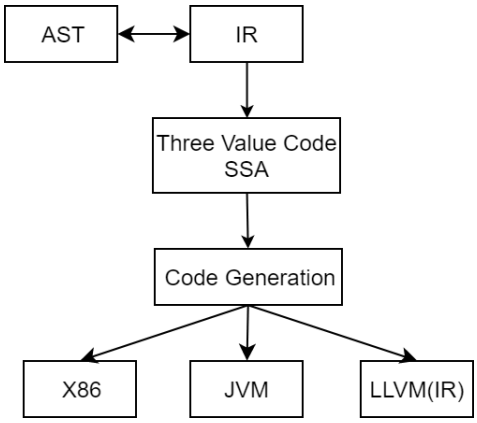

Figure 18.3: CPU frequency scaling over the years

18.4.1 Strength Reduction

Replacing expensive operations by cheaper operations. Example: Division is an expensive operation. So, compiler replaces division operation in say, "/2" with right shift operator which is similar to division but cheaper compared to it.

18.5 SparkSQL Optimizations

Optimizer in SparkSQL is called a 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. Consider your program, pick an optimization, it gives you new AST, keep continuing the optimization process and stop it when converges i.e., when the AST's are equivalent. A real compiler does not do this but Spark SQL does. Spark SQL optimizes SQL queries and optimizes UDF's using limited expression language as optimizing UDF's with Java is a high level job. All the SQL queries and all the expressions are compiled to a giant AST.

Several optimization techniques:

1. 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.

2. Column 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.

3. 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.