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

Lecture 19: Hive & HiveQL

Lecturer: Emery Berger *Scribe(s): Sarthak Nandi, Mohit Surana*

19.1 Precursors to Hive

19.1.1 Why MapReduce/Hadoop?

1. Abstractions
   (a) Fault tolerance is ensured and a developer need not clutter their core application logic
   (b) "Implicit" parallelism obtained if you are able to follow the MapReduce paradigm while framing your solution
   (c) Scaling up (# of processors) and out (# of machines) to support more clients and to handle large amounts of data
2. Heterogeneous machine configurations* (different RAM, CPU, Disk I/O) are supported and hence you don't need to upgrade all systems at one go
3. Schemaless nature allows you to store any form of data (for later processing)

* Each unique configuration is referred to as a SKU (Stock Keeping Unit), The number of SKUs are minimized for the purpose of quality control. This way, they can "blame the right people" when things go wrong.

19.1.2 Comparing different systems

MapReduce provided narrow abstraction but supplanted traditional db. It allows iterative queries which are harder in SQL along with ease of scaling out.

Graph of performance scalability vs abstraction level vs generality summarized as follows:

Data processing system	Performance Scalability	Abstraction Level	Generality
SQL (without UDFs)	low	high	low
SQL (with UDFs)	low	high	high
MapReduce	high	low	low
Redis	high	medium	low
MongoDB	high	high	low
BigTable	high	low	low
Hive	high	high	high

Table 19.1: Comparison of different data processing systems

UDFs greatly increase generality for SQL, as you can even process blobs in UDF. MapReduce has a narrow API, and requires multiple stages for several kinds of operations.

MapReduce focused on high scalability. After MapReduce, newer frameworks are focusing on increasing abstraction level and generality while increasing/preserving scalability.

19.2 Hive

19.2.1 Introduction

More like SQL, but scalable as it is on top of MapReduce (Hadoop). HiveQL supports DDL to create tables and specify how we want the data to be stored (serialization, partitioning schemes) and DML to support a **subset of** data manipulation queries that are supported by **SQL**.

Using Java in UDFs provides stronger integration and coupling with Hadoop and eliminates the need to run external programs on Hadoop using JNI. Similarly, Microsoft earlier had UDF's in C++, now in C# which is very similar to Java.

19.2.2 Code example

HiveQL allows a user who may not be familiar with the low level details of MapReduce to run queries via a language similar to SQL. The following snippet demonstrates that how a simple SELECT - FROM - WHERE translates to a couple of lines of code:

```
ctx = new HiveContext();
users = ctx.table("user");
young = users.where(users("age") < 21);
System.out.println(young.count());
```

19.2.3 Implementation

Data in tables is stored in the form of files on HDFS. Tables and keys are represented by directories, partitions are stored as subfolders and the individual rows are stored as buckets in files inside these directories.

/wh/table/ds=20090101/ctry=US

Keeping keys in directory names is a way of getting back locality, and enables distribution. In the example above, all rows for the same dates would be on the same machine, hence providing increased locality. The increased locality would be for some specific queries along certain columns, not along all.

Hive compiles queries in a way similar to FlumeJava using dataflow graphs equivalent to DAGs of MapReduce jobs. Hive also performs optimizations like:

1. Predicate pushdown (as far as possible) - try to filter first to reduce data size transmitted over the network (eg. query where WHERE section happens and filters data)
2. Column pruning - only use columns that we need (eg. query where we use column name instead of *)

19.2.4 Support for sampling

Hive supports sampling queries which is widely used in big data when approximate answers are acceptable. Concept: look at x% of total records and multiply the answer by x to get a rough approximation of the actual answer. The data must be random. 95% of times sampling results in confidence interval with correct number of samples.

E.g. instead of taking average salary of all employees, take salary from every 100th.

Sampling does not work when the actual number of examples satisfying our conditions are too small i.e. the issue of needle in a hay stack. E.g. age 81. Sample needs to have some representation at sampling rate. Sampling also cannot be used when very high accuracy is needed.

19.2.5 Advantages

1. High level declarative language that allows people unfamiliar with the MapReduce abstraction to leverage the scalability of Hadoop
2. UDFs (single row) & UDAFs (over multiple rows) in Java (because it is closely tied to Hadoop which is also in Java)
3. Hive supports incoming stream of rows

19.2.6 Disadvantages

1. No support for update / delete rows in existing tables
2. Is slow as it needs disk I/O as the data is stored on Hadoop
3. Only supports a subset of the functionality exposed by SQL
4. UDFs and UDAFs are treated as black boxes and hence HiveSQL is unable to optimize them

Some of these issues are the primary focus of SparkSQL (to be discussed in next class).

Side notes

Discussion regarding one of the causes of stragglers

Bad blocks - this can be handled by bad block remapping (adding one level of indirection), extra seek latency for items that we presumed locality (slow decay or death)

Link to Prof. Berger's notes for this class