# The Hadoop Ecosystem and Data Warehousing and Manipulation with Hive

## Advanced Analytics with Machine Learning and Hadoop: Week One Lecture

Welcome to *Advanced Analytics with Machine Learning and Hadoop*! This course is designed as a continuation of the material introduced in the *Introduction to Analytics using Hadoop* course, which provides an introduction to the Hadoop core components and MapReduce paradigm. In this course, we will dive further into some of the higher-level programming languages and projects within the Hadoop ecosystem and learn how these tools help us take advantage of the Hadoop architecture to more easily perform big-data processing.

This course assumes that you have completed the *Introduction to Analytics using Hadoop* course, or, have an equivalent familiarity with Hadoop and its core components, including HDFS and the MapReduce API. Additionally, an intermediate-level familiarity with Java is recommended, since many of the tools we will be covering utilize a Java-based API. However, the focus of this course will be on the analytic capabilities that these tools provide and less on the code implementation. By the end of this course, you will have been introduced to several key open-source data tools within the Hadoop ecosystem that allow you load data into Hadoop, analyze data using higher-level languages that compile into MapReduce jobs, create data processing workflows, and apply machine learning algorithms atop Hadoop.

This first week's lesson provides a quick overview of the specific projects within the Hadoop ecosystem that we will focus on in this course, and reviews how to setup a local, pseudo-distributed Hadop environment. We will then discuss the Apache Hive project in detail, learning how to load data into Hive tables and perform structured query analysis on our Hadoop data sets. At the end of this lecture, you should have a broad

sense of the topics and Hadoop projects we'll cover in this course, and understand how to perform basic data analysis using Hive.

## Table of Contents



 

## Chapter 11: The Hadoop Ecosystem

While HDFS and MapReduce form the core foundation of the Hadoop framework[1], other projects have now grown around these components that allow developers to more easily leverage the distributed data storage and processing capabilities of Hadoop. In fact, numerous open-source and proprietary bundled Hadoop distributions have since been released which extend Hadoop into a general-purpose platform for big data analysis. Many of the projects contained in these distributions are also hosted open-source projects by the Apache Software Foundation.

These associated projects form what is often called the "Hadoop ecosystem", comprised of additional tools and languages that enable common data processing and analysis patterns by abstracting the native Hadoop libraries behind a higher-level interface. These tools thus enable developers to build complex data processing, workflows, and machine learning patterns that leverage Hadoop's massive-parallelization without requiring the direct implementation of its native MapReduce API.

It is important to note that the Hadoop ecosystem is still constantly growing and evolving, and many of the projects that exist under the umbrella of Hadoop seek to address the same or similar problems in different ways. Hence why we use the term "ecosystem" rather than "suite", as these tools are not neatly parcelled into distinct functional categories. Hadoop programmers should thus evaluate these tools and libraries in the context of their specific use case and programming preference. For the scope of this book, we will explore some actively supported Hadoop tools and languages that are frequently applied in the context of big data processing and analytics.

[1] Apache Hadoop Project: http://hadoop.apache.org/

## Data Warehousing with Hadoop

### Hive

While Hadoop is very efficient at storing and processing massive amounts of unstructured, file-based data, it falls short in the capabilities we would typically expect from a data warehousing application. This problem actually stems from the inherently unstructured and flexible nature of HDFS and MapReduce, since common data warehousing capabilities like dimensional data modeling or schema design, extract/transform/load operations (ETL) and online analytical processing (OLAP) often assume an imposed data structure in order to perform structured query operations.

This is an area where relational database management systems (RDBMS) and Structured Query Language (SQL) excel, and while SQL isn't ideal for every data problem, it is a standard among data analysts and a common bridge between other data applications for integration to software platforms not just people. However, organizations are increasingly finding that the volume and velocity of their data is quickly surpassing the capabilities of an RDBMS-based data warehousing solution[2].

Hive is an abstraction of MapReduce that provides a framework for data warehousing on top of Hadoop, developed by the Facebook team. In particular it provides a HiveQL language that is very similar to SQL and allows analysts with strong SQL skills to run queries on high volumes of data residing in HDFS or HBase. The Hive Interpreter simply runs on a client machine and compiles HiveQL queries into MapReduce jobs that are then submitted to the cluster. We will discuss Hive in detail in this lesson.

### HBase

HDFS is optimized for batch processing, where the data is written once and processed by MapReduce in a streaming, sequential fashion[3]. Unfortunately, this makes HDFS ill suited for random, quick read/write access to individual records within a large dataset.

---

[2] IBM Developer Works "Build a data warehouse with Hive": http://www.ibm.com/developerworks/library/bd-hivewarehouse/

[3] The Apache HBase Reference Guide: http://hbase.apache.org/book.html

HBase is a database that can run on top of HDFS, enabling more traditional, real-time data access while also providing the benefits of linear scalability. HBase is a distributed, non-relational, column-oriented database, modeled after Google's *BigTable*[4]. In Week 2, we will review the data storage characteristics of HBase and how to utilize HBase to provide low-latency, random access over large data sets.

### Sqoop

Hive provides structured querying capabilities in Hadoop, but often before we get to that step, we need to first migrate our data from a relational database into HDFS. Additionally, after we perform our data processing in Hadoop, we may need to migrate that data back into a relational store for consumption in other applications.

Sqoop is a tool that enables us to both import and export data between relational databases, including MySQL, PostgreSQL, Oracle, SQL Server, DB2 and other JDBC-protocol databases, and HDFS, HBase, or directly into a Hive table.

## Higher-Order Hadoop Programming

MapReduce provides us with flexible, fine-grained control on how we process our data, regardless of the native format and structure of that data. However, in many cases we would like to abstract standard data processing patterns from low-level map and reduce operations, while still leveraging the large-scale capabilities of Hadoop. We discuss two libraries that provide a higher-level abstraction of the MapReduce API.

### Pig

Pig is a high-level MapReduce platform that includes an expressive language, called Pig Latin, which enables programmers to write what would otherwise be complex MapReduce transformations through a simple scripting language. Like Hive, Pig scripts are compiled into MapReduce code that executes over HDFS. However, Pig raises the

[4] Bigtable: A Distributed Storage System for Structured Data: F. Chang, J. Dean, S. Ghemawat, et al.

level of abstraction by describing data analysis problems as data flows[5]. In Week 3, we will revisit some data analytics problems using Pig.

### Cascading

Cascading is similar to Pig in that it enables users to define and execute data-processing flows in Hadoop. However, rather than provide its own native scripting language, the Cascading library is based on Java. Though written in Java, the Cascading API still hides many of the complexities of MapReduce programming by abstracting standard data processing operations (split, joins, etc.) away from underlying mapper and reducer tasks.

## Machine-Learning and Data Mining

### The 3 C's of Machine Learning with Mahout

In *Introduction to Analytics using Hadoop,* we introduced three common Machine Learning paradigms: Clustering, Classification and Collaborative Filtering. These statistical techniques typically involve a range of machine learning and data-mining algorithms that are commonly applied on text-based data sets; for example, K-means clustering, Bayesian techniques for classification, and using the Pearson correlation as a similarity measurement in collaborative filtering.

Apache Mahout is a library that enables developers to implement these core machine-learning patterns and algorithms on top of Hadoop[6]. Mahout specifically focuses on recommendation mining of text-based data sets, by supporting the three areas of clustering, classification, and collaborative filtering to support the necessary infrastructure for such applications. In Week 4, we will build a simple recommender by using Mahout to implement some common patterns across these key use cases.

---

[5] Hortonworks "Apache Pig": http://hortonworks.com/hadoop/pig/
[6] Apache Mahout: https://mahout.apache.org/

## Chapter 12: Structured Data Queries with Hive

Apache Hive is a data warehousing framework built on top of Hadoop, which allows developers who are fluent in SQL to leverage the Hadoop platform while using familiar querying and data manipulation patterns. Hive was initially developed at Facebook to utilize the availability, scalability, and resilience of the Hadoop infrastructure, without sacrificing their well-known structured data schemas or requiring their developers to adopt an unfamiliar programming paradigm in MapReduce[7].

Hadoop's distributed infrastructure enables Hive to provide both scalability and efficiency in querying massive amounts of structured data, which would typically stretch the limits of traditional RDBMS solutions. The Hive Query Language (HQL) provides a SQL abstraction over MapReduce, enabling developers to define, query and manipulate their data using familiar and intuitive data access patterns.

Hive commands and queries are compiled, optimized, and executed by the Hive Driver, usually in the form of MapReduce jobs which are sent to the JobTracker to be run across the Hadoop cluster[8]. Thus, Hive has inherited certain limitations from HDFS and MapReduce that constrain it from providing key online transaction processing (OLTP) features that one might expect from a traditional database management system. In particular, because HDFS is a write-once, read-many filesystem, Hive does not provide row-level inserts, updates, or deletes. Additionally, Hive queries entail a higher-latency due to the overhead required to generate and launch the MapReduce jobs on the cluster; even small queries that would complete within a few seconds on a traditional RDBMS will take upwards of ten seconds to finish with Hive.

On the plus side, Hive provides the high-scalability and high-throughput that you would expect from any Hadoop-based application, and as a result, is very well-suited to batch-

[7] What is HIVE? http://www.slideshare.net/nzhang/hive-training-motivations-and-real-world-use-cases
[8] Programming Hive: Capriolo, Edward; Wampler, Dean; Rutherglen, Jason. (p. 7). O'Reilly Media.

level workloads for online analytical processing (OLAP) of very large data sets at the terabyte and petabyte scale.

## Setting Up Hive

### Installation

Hive requires Hadoop and Java 1.6+. For an overview on the various mechanisms to install Hadoop, refer to guide in the Resources section titled “Installing a Development Environment”. For the purposes of this chapter, we will assume that you are working with Hadoop 1.2.1 under Ubuntu as a "single node cluster", or pseudo-distributed mode, and using the Oracle Java platform with JDK 1.7.

We will similarly install Hive in a local, pseudo-distributed mode, but for a complete overview of the available Hive installation and configuration options, we recommend reading the excellent O'Reilly book *Programming Hive*.

The installation process for Hive is similar to Hadoop's. We'll start by downloading and extracting the latest stable Hive release tarball from the Apache Hive Download Mirrors, which is as of this writing, currently at version 0.11.0:

```
~$ wget http://archive.apache.org/dist/hive/hive-0.11.0/hive-
0.11.0.tar.gz
~$ sudo mv hive-0.11.0.tar.gz /srv/
~$ cd /srv
/srv$ sudo tar -xzf hive-0.11.0.tar.gz
/srv$ sudo chown -R analyst:analyst hive-0.11.0
/srv$ sudo ln -s $(pwd)/hive-0.11.0 $(pwd)/hive
```

We will also add the following to our environment variables to our bash profile (~/.profile) for convenience:

```
export HIVE_HOME=/srv/hive
export PATH=$PATH:$HIVE_HOME/bin
```

Also, if you don't already have the HADOOP_PREFIX environment variable set, make sure to include that as well, as Hive will reference HADOOP_PREFIX to find Hadoop's libraries and configuration files:

```
export HADOOP_PREFIX=/srv/hadoop
```

And while we're checking our Hadoop settings, let's verify that Hadoop is running by restarting our Hadoop services:

```
/srv$ $HADOOP_PREFIX/bin/stop-all.sh
/srv$ $HADOOP_PREFIX/bin/start-all.sh
```

## Hive Warehouse Directory

By default, Hive data is stored in HDFS, in a warehouse directory located under /user/hive/warehouse. Make sure this location exists and is writable by all Hive users. If you want to change this location, you can modify the value for the hive.metastore.warehouse.dir property in hive-site.xml.

For our local configuration, we'll just create the directory within the local filesystem and set the permissions for all users:

```
/srv$ sudo mkdir -p /user/hive/warehouse
/srv$ sudo chmod a+rwx /user/hive/warehouse
```

## Hive Metastore Database

Hive requires a *metastore* service backend, which Hive uses to store table schema definitions, partitions, and related metadata. The Hive metastore service also provides clients (including Hive) with access to the metastore info via the *metastore service API*.

The metastore can be configured in a few different ways, with the default Hive configuration using an embedded metastore called the *Derby SQL Server* which provides single-process storage where the Hive driver, metastore interface, and Derby database all share the same JVM. This is a convenient configuration for development and unit testing, but will not support true cluster-configurations since only a single user can connect to the Derby database at any given time.

By default, Derby will create a *metastore_db* subdirectory under the current working directory from which you started your Hive session, and if you change your working directory, Derby will fail to find the previous metastore and will thus recreate it. To avoid this behavior, we need to configure a permanent location for the metastore database by updating the metastore configuration:

```
/srv$ cd $HIVE_HOME/conf
/srv/hive/conf$ cp hive-default.xml.template hive-site.xml.template
/srv/hive/conf$ vim hive-site.xml.template
```

Find the property with the name *javax.jdo.option.ConnectionURL* and update it an absolute path:

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>

<value>jdbc:derby:;databaseName=/srv/hive/metastore_db;create=true</va
lue>
    <description>JDBC connect string for a JDBC
metastore</description>
</property>
```

For the purposes of this chapter, we will use the embedded Derby server as our metastore service. But we encourage you to refer to the Apache Hive manual for installing a local or remote metastore server for production-level configurations.

## The Hive Command-Line Interface (CLI)

Hive's installation comes packaged with a handy command-line interface (CLI), which we will use to interact with Hive and run our HQL statements. To start the Hive CLI from the $HIVE_HOME:

```
/srv$ cd $HIVE_HOME
/srv/hive$ bin/hive
```

This will initiate the CLI and bootstrap the logger (if configured) and Hive history file, and finally display a Hive CLI prompt:

```
Logging initialized using configuration in jar:file:/srv/hive-
0.11.0/lib/hive-common-0.11.0.jar!/hive-log4j.properties
Hive history file=/tmp/analyst/hive_job_log_analyst_4367@analyst-
virtualbox_201404060214_1144745181.txt
hive>
```

At any time, you can exit the Hive CLI by entering *exit* followed by a semi-colon and . Hive can also run in non-interactive mode directly from the command line by passing the filename option, -f, followed by the path to the script to execute:

```
/srv/hive$ hive -f /home/analyst/script.sql
```

Additionally, the quoted-query-string option, -e, allows you to run inline commands from the command line:

```
/srv/hive$ hive -e 'SHOW TABLES;'
```

You can view the full list of Hive options for the CLI by using the -H flag:

```
/srv/hive$ hive -H
usage: hive
 -d,--define <key=value>          Variable subsitution to apply to
hive
                                  commands. e.g. -d A=B or --define
A=B
    --database <databasename>     Specify the database to use
 -e <quoted-query-string>         SQL from command line
 -f <filename>                    SQL from files
 -H,--help                        Print help information
 -h <hostname>                    connecting to Hive Server on remote
host
    --hiveconf <property=value>   Use value for given property
    --hivevar <key=value>         Variable subsitution to apply to
hive
                                  commands. e.g. --hivevar A=B
 -i <filename>                    Initialization SQL file
 -p <port>                        connecting to Hive Server on port
number
 -S,--silent                      Silent mode in interactive shell
 -v,--verbose                     Verbose mode (echo executed SQL to
the
                                  console)
```

## Hive Querying Language (HQL)

### Creating a Database

Creating a database in Hive is very similar to creating a database in a SQL-based RDBMS, by using the CREATE DATABASE or CREATE SCHEMA statement:

```
hive> CREATE DATABASE log_data;
```

Hive will raise an error if the database already exists in the metastore; we can check for the existence of the database by using IF NOT EXISTS:

```
hive> CREATE DATABASE IF NOT EXISTS log_data;
```

We can then run SHOW DATABASES to verify that our database has been created. Hive will return all databases found in the metastore, along with the default Hive database:

```
hive> SHOW DATABASES;
OK
default
log_data
Time taken: 0.085 seconds, Fetched: 2 row(s)
```

Additionally, we can set our working database with the USE command:

```
hive> USE log_data;
```

### Creating Tables

Hive also provides a SQL-like CREATE TABLE statement which in its simplest form, takes a table name and column definitions:

```
CREATE TABLE apache_log (
    host STRING,
    identity STRING,
    user STRING,
    time STRING,
    request STRING,
    status STRING,
    size STRING,
    referer STRING,
    agent STRING
);
```

However, because Hive data is stored in the filesystem, either in HDFS or the local filesystem, the CREATE TABLE command also takes optional clauses to specify the row format with the ROW FORMAT clause that tells Hive how to read each row in the file and map to our columns. For example, we could indicate that the data is in a delimited file with fields delimited by the tab-character:

```
hive> CREATE TABLE shakespeare (
    lineno STRING,
    linetext STRING
)
```

```
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t';
```

In the case of the Apache access log, we will instead use the Hive serializer-deserializer row format option, SERDE, and the contributed RegexSerDe library to specify a regex with which to deserialize and map the fields into columns for our table. We'll need to manually add the hive-contrib jar from the lib folder to the current hive session in order to use the RegexSerDe package:

```
hive> ADD JAR /srv/hive/lib/hive-contrib-0.11.0.jar;
hive> CREATE TABLE apache_log(
   host STRING,
   identity STRING,
   user STRING,
   time STRING,
   request STRING,
   status STRING,
   size STRING,
   referer STRING,
   agent STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
   WITH SERDEPROPERTIES (
 "input.regex" = "([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^
\"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\".*\") ([^
\"]*|\".*\"))?",
 "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s
%9$s"
)
STORED AS TEXTFILE;
```

Once we've created the table, we can use DESCRIBE to verify our table definition:

```
hive> DESCRIBE apache_log;
OK
host                    string                  from deserializer
identity                string                  from deserializer
user                    string                  from deserializer
time                    string                  from deserializer
request                 string                  from deserializer
status                  string                  from deserializer
size                    string                  from deserializer
referer                 string                  from deserializer
agent                   string                  from deserializer
Time taken: 0.553 seconds, Fetched: 9 row(s)
```

Note that in this particular table, all columns are defined with the Hive primitive data type, string. Hive supports many other primitive data types that will be familiar to SQL users and generally correspond to the primitive types supported by Java including[9]:

| Type | Description | Example |
|---|---|---|
| TINYINT | 8-bit signed integer, from -128 to 127 | 127 |
| SMALLINT | 16-bit signed integer, from -32,768 to 32,767 | 32,767 |
| INT | 32-bit signed integer | 2,147,483,647 |
| BIGINT | 64-bit signed integer | 9,223,372,036,854,775,807 |
| FLOAT | 32-bit single-precision float | 1.99 |
| DOUBLE | 64-bit double-precision float | 3.14159265359 |
| BOOLEAN | true/false | true |
| STRING | 2GB max character string | 'a'<br>"hello world" |
| TIMESTAMP | Nanosecond precision | 1400561325 |

In addition to the primitive data types, Hive also supports complex data types that can store a collection of values:

| Type | Description | Example |
|---|---|---|
| ARRAY | Ordered collection of elements. The elements in the array must be of the same type. | recipients<br>ARRAY<email:STRING> |

[9] Hive Language Manual and DDL:
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL

| | | |
|---|---|---|
| MAP | Unordered collection of key-value pairs. Keys must be of primitive types and values can be of any type. | files MAP<filename:STRING, size:INT> |
| STRUCT | Collection of elements of any type. | address STRUCT<street:STRING, city:STRING, state:STRING, zip:INT> |

This may seem awkward at first, since relational databases generally don't support collection types, but instead store associated collections in separate tables to maintain *first normal form* and minimize data duplication and the risk of data inconsistencies. However, in a big data system like Hive where we are processing large volumes of unstructured data by sequentially scanning off disk, the ability to read embedded collections provides a huge benefit in retrieval performance[10].
For a complete reference of Hive's supported table and data type options, refer to the documentation in the Apache Hive Language Manual.

### Loading Data

With our table created and schema defined, we are ready to load the data into Hive. It's important to note one important distinction between Hive and traditional RDBMSs with regards to schema enforcement--Hive does not perform any verification of the data for compliance with the table schema nor does it perform any transformations when loading the data into a table.

Traditional relational databases enforce the schema on *writes* by rejecting any data that does not conform to the schema as defined; Hive can only enforce queries on schema *reads*. If in reading the data file, the file structure does not match the defined schema, Hive will generally return null values for missing fields or type mismatches and attempt to recover from errors. Schema on read enables a very fast initial load, since the data is not read, parsed, and serialized to disk in the database's internal format. Load

[10] Programming Hive: Capriolo, Edward; Wampler, Dean; Rutherglen, Jason. (pp. 43-44). O'Reilly Media.

operations are purely copy/move operations that move datafiles into locations corresponding to Hive tables[11].

Data loading in Hive is done in batch-oriented fashion using a bulk LOAD DATA command or by inserting results from another query with the INSERT command. To start, let's pull down the Apache log data file[12] and load it into the table we created earlier:

```
~$ wget -O ~/apache.log.gz
ftp://ita.ee.lbl.gov/traces/calgary_access_log.gz
~$ gunzip ~/apache.log.gz
~$ $HIVE_HOME/bin/hive

hive> use log_data;
OK
Time taken: 0.221 seconds

hive> LOAD DATA LOCAL INPATH '${env:HOME}/apache.log' OVERWRITE INTO
TABLE apache_log;
Copying data from file:/home/analyst/apache.log
Copying file: file:/home/analyst/apache.log
Loading data to table log_data.apache_log
Deleted
hdfs://localhost:9000/user/hive/warehouse/log_data.db/apache_log
Table log_data.apache_log stats: [num_partitions: 0, num_files: 1,
num_rows: 0, total_size: 52276884, raw_data_size: 0]
OK
Time taken: 5.895 seconds
```

LOAD DATA is Hive's bulk loading command. The LOCAL option indicates that the filepath is on the local filesystem, so that Hive will *copy* it to the target location for the table. By default, Hive will assume the file is already on HDFS and will proceed to *move* the file into the table location. If the OVERWRITE keyword is used, then any existing data in the target table will be deleted and replaced by the data file input; otherwise the new data will be added to the table. Once the data has been copied and loaded, Hive will output some statistics on the loaded data; although the num_rows reported is 0, you can verify the actual count of rows by running a SELECT COUNT (output truncated):

---

[11] Apache Hive Language Manual DML: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML

[12] University of Calgary: http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html

```
hive> SELECT COUNT(1) FROM apache_log;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
726739
Time taken: 34.666 seconds, Fetched: 1 row(s)
```

After the MapReduce jobs have executed, you should see that the apache_log table now contains 726739 rows.

## Data Analysis with Hive

### Grouping

Given an Apache access log file, with rows consisting of web log data in the Apache Common Log Format[13]:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0"
200 2326
```

Consider a MapReduce program that computes the number of hits per calendar month; although this is a fairly simple group-count problem, implementing the MapReduce program still requires a decent level of effort to write the Mapper, Reducer, and main function to configure the job, in addition to the effort of compiling and creating the JAR file. However with Hive, this problem is as simple and intuitive as running a SQL GROUP BY query:

```
hive> SELECT month, count(1) AS count FROM (SELECT split(time, '/')[1]
AS month FROM apache_log) l GROUP BY month ORDER BY count DESC;
OK
Mar 99717
Sep 89083
Feb 72088
Aug 66058
Apr 64984
May 63753
Jul 54920
Jun 53682
Oct 45892
Jan 43635
```

[13] http://httpd.apache.org/docs/2.2/logs.html#accesslog

```
Nov 41235
Dec 29789
NULL    1903
Time taken: 84.77 seconds, Fetched: 13 row(s)
```

Both the Hive query and the MapReduce program perform the work of tokenizing the input and extracting the month token as the aggregate field. However, not only does Hive provide a succinct and natural query interface to perform the grouping, but since our data is structured as a Hive table, we can easily perform other ad-hoc queries on any of the other fields:

```
hive> SELECT host, count(1) AS count FROM apache_log GROUP BY host
ORDER BY count;
```

In addition to count, Hive also supports other aggregate functions to compute the sum, average, min, max as well as statistical aggregations for variance, standard deviation, and covariance of numeric columns. When using these built-in aggregate functions, you can improve the performance of the aggregation query by setting the following property to true:

```
hive> SET hive.map.aggr = true;
```

This setting tells Hive to perform "top-level" aggregation in the map phase, as opposed to aggregation after performing a GROUP BY. However, be aware that this setting will require more memory[14]. A full list of built-in aggregate functions can be found here: Hive Operators and User-Defined Functions (UDFs)

Using Hive also provides us with the convenience of easily storing our computations. We can create new tables to store the results returned by these queries for later record-keeping and analysis.

```
hive> CREATE TABLE remote_hits_by_month AS SELECT month, count(1) AS
count FROM (SELECT split(time, '/')[1] AS month FROM apache_log WHERE
host == 'remote') l GROUP BY month ORDER BY count DESC;
```

[14] Programming Hive: Capriolo, Edward; Wampler, Dean; Rutherglen, Jason. (p. 86). O'Reilly Media.

### Joins

We've covered some of the conveniences that Hive offers in querying and aggregating data from a single, structured data set, but Hive really shines when performing more complex aggregations across multiple data sets.

In the previous course, we developed a MapReduce program to analyze the on-time performance of U.S. airlines based on flight data collected by the Research and Innovative Technology Administration (RITA) Bureau of Transportation Studies[15] (this exercise can be found in the Resources section of this course). The on-time data set was normalized in that chapter to include all required data within a single data file, however in reality the data as downloaded from RITA's website actually includes codes that must be cross-referenced against separate lookup data sets for the Airline and Carrier Codes.

Thus each row of the on-time flight data includes an integer value that represents the code for AIRLINE_ID (i.e. - 19805) and a string value that represents the code for CARRIER (i.e. - "AA"). AIRLINE_ID codes can be joined with the corresponding code in the airline_code_lookup.csv file in which each row contains the code and corresponding description:

```
"19805","American Airlines Inc.: AA"
```

Accordingly, CARRIER codes can be joined with the corresponding code in carrier_code_lookup.csv which contains the code and corresponding airline name and effective dates:

```
"AA","American Airlines Inc. (1960 - )"
```

[15] RITA TransStats: http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time

Implementing these joins in a MapReduce program would require either a map-side join to load the lookups in memory, or reduce-side join in which we'd perform the join in the reducer. Both methods require a decent level of effort to write the MapReduce code to configure the job, but with Hive, we can simply load these additional lookup data sets into separate tables and perform the join in a SQL query.

Assuming that we've uploaded our data files to HDFS or local filesystem, let's start by creating a new database for our flight data:

```
hive> CREATE DATABASE flight_data;
OK
Time taken: 0.741 seconds
```

And then define schemas and load data for the on-time data and lookup tables (output omitted and newlines added for readability):

```
hive> CREATE TABLE ontime_data (
    flight_date TIMESTAMP,
    airline_id INT,
    carrier_id STRING,
    origin STRING,
    dest STRING,
    depart_time STRING,
    depart_delay INT,
    arr_time STRING,
    arr_delay FLOAT,
    cancelled FLOAT,
    elapsed_time FLOAT,
    air_time FLOAT,
    distance FLOAT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

hive> CREATE TABLE airlines (code INT, description STRING) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

hive> CREATE TABLE carriers (code STRING, description STRING) ROW
FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;

hive> LOAD DATA LOCAL INPATH '${env:HOME}/flights_jan_2014.csv'
OVERWRITE INTO TABLE ontime_data;
```

```
hive> LOAD DATA LOCAL INPATH '${env:HOME}/airline_code_lookup.csv'
OVERWRITE INTO TABLE airlines;

hive> LOAD DATA LOCAL INPATH '${env:HOME}/carrier_code_lookup.csv'
OVERWRITE INTO TABLE carriers;
```

To get a list of airlines and their respective average departure delays, we can simply perform a SQL JOIN on ontime_data and airlines on the airline code and then use the aggregate function AVG() to compute the average depart_delay grouped by the airline description:

```
hive> SELECT a.description, AVG(f.depart_delay) FROM airlines a JOIN
ontime_data f ON a.code = f.airline_id GROUP BY a.description;

"AirTran Airways Corporation: FL"   13.729980745271265
"Alaska Airlines Inc.: AS"  0.7249449563728289
"American Airlines Inc.: AA"    11.214197756088662
"American Eagle Airlines Inc.: MQ"  17.11047971455002
"Delta Air Lines Inc.: DL"  17.412184334427167
"ExpressJet Airlines Inc.: EV"  22.330995848950145
"Frontier Airlines Inc.: F9"    20.058729877940916
"Hawaiian Airlines Inc.: HA"    -1.479647782023592
"JetBlue Airways: B6"   26.939031811894882
"SkyWest Airlines Inc.: OO" 9.526145563809365
"Southwest Airlines Co.: WN"    22.316265302142014
"US Airways Inc.: US (Merged with America West 9/05. Reporting for
both starting 10/07.)"   6.403216729362282
"United Air Lines Inc.: UA" 16.68654160087778
"Virgin America: VX"    6.461069023569023
Time taken: 55.305 seconds, Fetched: 14 row(s)
```

As you can see, performing joins in Hive versus MapReduce provides pretty significant savings in coding effort (also, Hawaiian Airlines seems to be doing pretty well in 2014 so far!). More importantly, the structured Hive data schema that we've defined gives us the ability to add or change queries with ease; let's update our query to instead return the average departure delay grouped by carrier:

```
hive> SELECT c.description, AVG(f.depart_delay) FROM carriers c JOIN
ontime_data f ON c.code = f.carrier_id GROUP BY c.description;

"Aces Airlines (1992 - 2003)"   6.461069023569023
"AirTran Airways Corporation (1994 - )" 13.729980745271265
"Alaska Airlines Inc. (1960 - )"    0.7249449563728289
"American Airlines Inc. (1960 - )"  11.214197756088662
"American Eagle Airlines Inc. (1998 - )"    17.11047971455002
```

```
"Atlantic Southeast Airlines (1993 - 2011)" 22.330995848950145
"Delta Air Lines Inc. (1960 - )"    17.412184334427167
"ExpressJet Airlines Inc. (2012 - )"    22.330995848950145
"Frontier Airlines Inc. (1960 - 1986)"  13.729980745271265
"Frontier Airlines Inc. (1994 - )"  20.058729877940916
"Hawaiian Airlines Inc. (1960 - )"  -1.479647782023592
"JetBlue Airways (2000 - )" 26.939031811894882
"Simmons Airlines (1991 - 1998)"    17.11047971455002
"SkyWest Airlines Inc. (2003 - )"   9.526145563809365
"Southwest Airlines Co. (1979 - )"  22.316265302142014
"US Airways Inc. (1997 - )" 6.403216729362282
"USAir (1988 - 1997)"   6.403216729362282
"United Air Lines Inc. (1960 - )"   16.68654160087778
"Virgin America (2007 - )"  6.461069023569023
Time taken: 100.97 seconds, Fetched: 19 row(s)
```

Hive can be a good fit for use cases such as these, where we are working with datasets that lend themselves to a structured table-based format, and the computations that we are interested in are batch-oriented, OLAP queries, rather than real-time, row-oriented, OLTP transactions. For more information and further reading on using and optimizing Hive, we recommend the excellent example-driven O'Reilly book "Programming Hive".

## Conclusion

In this lecture, we introduced some of the projects within the Hadoop ecosystem, and set up a local development environment with Hadoop. We also learned how Hive enables us to utilize familiar data analysis patterns with HQL, and more easily perform batch-level data analysis on our data within Hadoop by providing an approachable SQL-like MapReduce abstraction.

Next week, we will explore Hadoop's NoSQL database, HBase, which offers real-time data access and row-level update capabilities while leveraging the large-scale capabilities of Hadoop. We will also learn how to use Sqoop to import data from existing relational database systems into a Hadoop data warehousing tool such as Hive or HBase. For the rest of the week, take a look at the reading, share your thoughts on the discussion questions in the forum, and set up your development environment before diving into the assignment.

## Discussion Questions

1. Share your past experience with Hadoop, either previous coursework, applications that you've built, or any other self-directed learning with Hadoop or Hadoop-related projects. What are you most hoping to learn from this course?
2. Describe the data warehousing technologies that you are using currently (or have used in the past), and the use cases, performance, data magnitude/scale, and data access requirements you have for your data warehouse. Does your data warehouse serve both OLTP and OLAP use cases?
3. What kinds of data warehouse applications are suitable for Hive? Why is Hive not a replacement for a relational database management system?

## Assignment

The Resources section contains a set of CSV data containing batting and pitching statistics from 2012, plus fielding statistics, standings, team stats, managerial records, post-season data, and more[16]. Included in the ZIP file is a README (readme 2012.txt) that describes the data contained in each of the CSVs.

Create a database called 'baseball_stats' and in that database create Hive tables to hold the data in Masters.csv, which contains a master list of players and their information, Teams.csv (Teams data), Batting.csv (batting statistics) and Salaries.csv (salary statistics). Load the CSV data into the tables you created, and implement the following queries:

1. List of top 50 players by highest number of home runs in a season. List the players by Player ID, First Name and Last Name, number of home runs and season year.
2. List average salaries for each team in 2012. List the teams by Team ID, League ID, Team Name, and average salary amount in descending order.

Submit your CREATE TABLE statements, query statements, and results for both queries.

---

[16] Lahman's Baseball Statistics: http://seanlahman.com/baseball-archive/statistics/