Using Hive and Hivemall scalable library over Hadoop for Data Analysis and Machine Learning

Supervisor: Professor Elias Savvas

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ ΓΙΑ ΤΟ ΠΡΟΓΡΑΜΜΑΜΕΤΑΠΤΥΧΙΑΚΩΝ
ΣΠΟΥΔΩΝ (ΠΜΣ) “ΜΗΧΑΝΙΚΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΔΙΑΔΙΚΤΥΑΚΕΣ
& ΦΟΡΗΤΕΣ ΕΦΑΡΜΟΓΕΣ” ΤΟΥ ΠΑΝΕΠΙΣΤΗΜΙΟΥ ΘΕΣΣΑΛΙΑΣ
ΓΕΝΙΚΟ ΤΜΗΜΑ ΛΑΡΙΣΑΣ

ΚΥΡΙΑΚΟΣ ΣΚΟΥΛΑΡΙΚΗΣ | ΑΜ 7419017

Abstract

Organizations are flooded with data. The volume and the variety of data that is available now to organizations is incredible: companies are storing more data from more sources in more formats than ever before. And they have realized the data they gather is a valuable resource for understanding their customers, the performance of their business in the marketplace, and the effectiveness of their infrastacture. The challenge here is that traditional tools are poorly equipped to deal with the scale and complexity of much of this data.

That's where Hadoop comes in.

The Hadoop ecosystem emerged as a cost-effective way of working with such large data sets. It imposes a particular programming model, called MapReduce, for breaking up computation tasks into units that can be distributed around a cluster. Underneath this computation model is a distributed file system called the Hadoop Distributed Filesystem (HDFS).

However, a challenge remains; how do you move an existing data infrastructure to Hadoop, when that infrastructure is based on traditional relational databases and the SQL?

This is where Hive comes in.

Hive is a standard for SQL queries over petabytes of data in Hadoop. It provides SQL-like access to data in HDFS, enabling Hadoop to be used as a data warehouse. The Hive Query Language (HiveQL) has similar semantics and functions as a standard SQL in the relational database, so that experienced database analysts can easily get their hands on it. Hive's query language can run on different computing engines, such as MapReduce, Tez and Spark. In this work although, for all the examples we will use MapReduce as the computing engine.

Ok, with Hive we can easily access and manipulate our data. But this data are valuable not being a "data packrat" but rather for building products, features, and intelligence predicated on knowing more about the world (where the world can be users, searches, machine logs, or whatever is relevant to an organization).

So, one of the vital components of Data Analytics is Machine learning.

Machine learning (ML) is the study of computer algorithms that improve automatically through experience. It is seen as a subset of artificial intelligence. Machine learning algorithms build a mathematical model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so.

Machine learning algorithms learn from data. It is critical that you feed them the right data for the problem you want to solve. Even if you have good data, you need to make sure that it is in a useful scale, format and even that meaningful features are included.

So, the logical question is, can we use SQL over Hive for Machine Learning?

And the answer is positive since 2016. We can use for Machine Learning theApache Hivemall, which is a collection of ML algorithms and versatile data analytics functions. It provides a number of ease of use ML functionalities through the Apache Hive UDF/UDAF/UDTF interface.

Apache Hivemall offers a variety of functionalities: regression, classification, recommendation, anomaly detection, k-nearest neighbor, and feature engineering. It also supports state-of-the-art Machine Learning algorithms such as Soft Confidence Weighted, Adaptive Regularization of Weight Vectors, Factorization Machines, and AdaDelta.

Key-words: *Hadoop, Big Data, Hive, Hivemall, MapReduce, SQL, HiveQL, Machine Learning, HDFS, Data Warehouse*

INDEX

CHAPTER 1: OVERVIEW OF BIG DATA: HADOOP

Hadoop and the term Big Data are like synonymous. But like many buzzwords, what people mean when they say "big data" is not always clear. This lack of clarity is made worse by IT people trying to attract attention to their own projects by labeling them as "big data", even though there's nothing big about them.

At its core we can say that big data is simply a way of describing data problems that are unsolvable using traditional tools. A well-known saying in this domain is to describe big data with the help of three words starting with the letter V: *volume, velocity and variety*. These different Vs are explained as follows [1]:

Volume: High volume of data ranging from dozens of terabytes and even petabytes.

Variety: Data that is organized in multiple structures, ranging from raw text (which, from a computer's perspective, has little or no discernible structure (unstructured data) to log files (semi structured) to data ordered 2 in strongly typed rows and columns (structured data). To make things even more confusing, some data sets even include portions of all three kinds of data (multi structured data).

Velocity: Data that enters an organization and has some kind of value for a limited window time –a window that usually shuts well before the data has been transformed and loaded into a data warehouse for deeper analysis. The higher the volumes of data entering an organization per second, the bigger the velocity challenge. Each of these criteria clearly poses its own, distinct challenge to someone wanting analyze the information.

The commonly held rule of thumb is that if your data storage and analysis work exhibits any of these three characteristics, chances are that you've got yourself a big data challenge. In summary, big data is not just about lots of data, it is a practice to discover new insight from existing data and guide the analysis of new data. A big-data-driven business will be more agile and competitive to overcome challenges and solve problems.

Apache Hadoop is a platform that provides pragmatic, cost-effective, scalable infrastructure for building applications based on big data. Made up of a distributed filesystem called Hadoop Distributed Filesystem (HDFS) and a computation layer that implements (and least when it was introduced) a processing paradigm called MapReduce, Hadoop is an open source, batch data processing[2] system for enormous amounts of data.

[3]Hadoop uses a cluster of plain old commodity servers with no specialized hardware or network infrastructure to form a single, logical, storage and compute platform, or cluster that can be shared by multiple individuals or groups.

For the history books, Hadoop was modeled after two papers produced by Google. The first was published in 2003, the now famous paper "The Google Filesystem[1]", describes a [3] pragmatic, scalable, distributed filesystem which was based on these four salient points [2]:

- Failures are the norm
- Files are large
- Files are changed by appending, not by updating
- Closely coupled application and filesystem APIs

The following year, another paper, titled "MapReduce: Simplified Data Processing on Large Clusters" was presented, defining a programming model and accompanying framework that provided automatic parallelization, fault tolerance and the scale to process hundreds of terabyres of data in a single job over thousands machines.

These papers directly inspired the development of HDFS and Hadoop MapReduce, respectively.

The general idea when an application is running on Hadoop is the work to be divided among the nodes (machines) in the cluster, HDFS stores the data that will be processed. A Hadoop cluster can span thousands of machines, where HDFS stores data, and MapReduce jobs do their processing near the data, which keeps I/O costs low.

HDFS

The first half of Apache Hadoop is the filesystem HDFS (Hadoop Distributed Filesystem). HDFS was built to support high throughput, streaming reads and writes of extremely large files.

There are a number of specific goals for HDFS [3]:

- Store millions of large files, each greater than tens of gigabytes, and filesystems sizes reaching tens of petabytes.
- Use scale-out model based on inexpensive commodity servers. Accomplish availability and high throughput through application-level replication of data.
- Optimize for large, streaming reads and writes rather than low-latency access to many small files. Batch performance is more important than interactive response times.
- Deal with component failures of machines and disks.
- Support the functionality and scale requirements of MapReduce processing.

HDFS does something unique. You take 30 computers for example and install an OS on each of them (like Ubuntu). After networking them together you install HDFS on all them and declare one of them as a master node and all the others computers (29 in our example) as worker nodes. This makes up your HDFS cluster. Now, when you copy files to a directory, HDFS stores parts of your file on multiple nodes in the cluster. In that way, HDFS becomes a virtual filesystem on top of the Linux filesystem. And the most peculiar is that HDFS abstracts the fact you're storing data on multiple nodes in a cluster. HDFS also creates and stores on the master node metadata of every file where it is stored on the cluster, acting as a directory, and providing a global picture of the filesystem's state.

Figure 1-1 shows a view of how HDFS works [2].

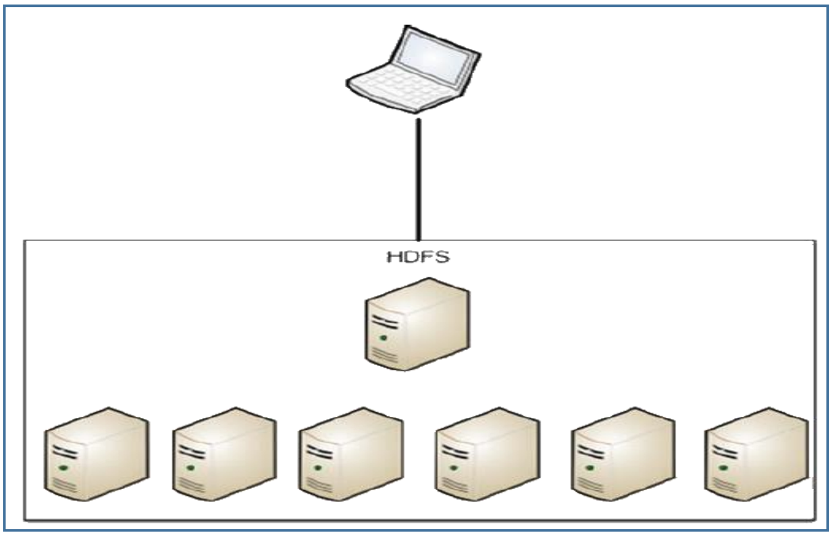


Figure 1-1

The salient point to take away is the ability to grow is now horizontal instead of vertical. Instead of adding CPU or RAM to a single machine, you simple need to add a new machine, i.e. a node. Linear scalability allows you to quickly expand your cluster capabilities based on your expanding resource needs.

HDFS replicates each block of a single large file to multiple machines in the cluster. By default, each block in a file is replicated three times. Because files in HDFS are write once, once a replica is written, it is not possible for it to change. This obviates the need for complex reasoning about the consistency between replicas and as a result, applications can read any of the available replicas when accessing a file. Having multiple replicas means multiple machine failures are easily tolerated.

MAPREDUCE

Storage is only part of the equation. Data is practically useless if we cannot process or analyze them.

MapReduce involves the processing of a sequence of operations on distributed data sets. The data consists of key-value pairs, and the computattions have only two phases: a map phase and a reduce phase. User-defined MapReduce jobs run on the compute nodes in the cluster.

The general workflow of a MapReduce job runs as follows [1]:

1. During the Map phase, input data is split into a large number of fragments, each of which is assigned to a map task.
2. These map tasks are distributed across the cluster.
3. Each map task processes the key-value pairs from its assigned fragment and produces a set of intermediate key-value pairs.
4. The intermediate data set is sorted by key, and the sorted data is portioned into a number of fragments that matches the number of reduce tasks.
5. During the Reduce phase, each reduce task processes the data fragment that was assigned to it and produces an output key-value pair.
6. These reduce tasks are also distributed across the cluster and write their output to HDFS when finished.

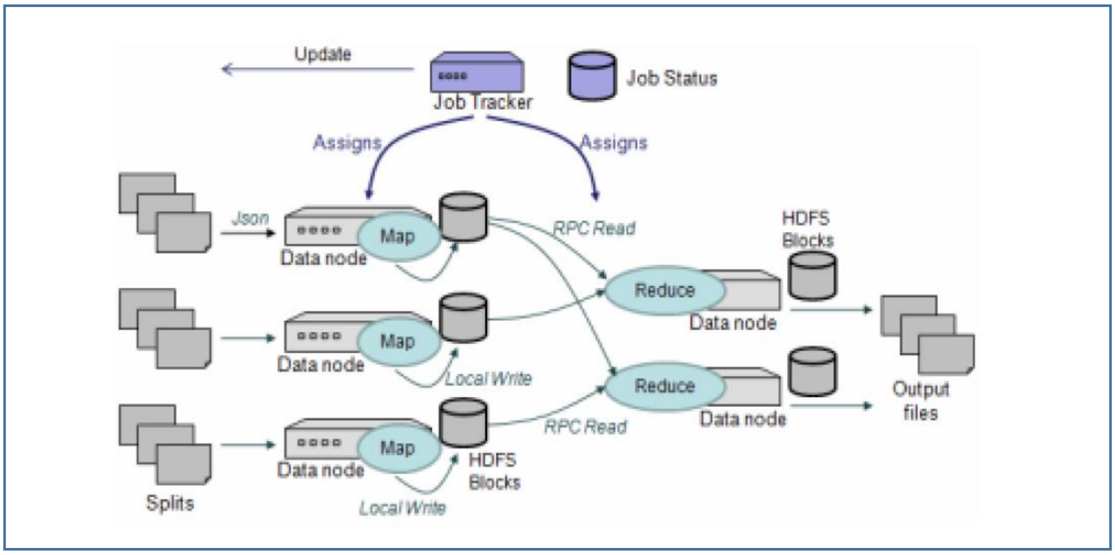


Figure 1-2

Do not forget Hadoop processes each of these in parallel. There is no need for communication between nodes during the Map phase. [2] This is critical when dealing with large data sets because you do not want inter-system communication or data transfer occurring between nodes. But not all the problems can easily or even be parallelized. MapReduce is not a silver bullet for every class of problems. [3] For example the act of training a model in ML cannot be parallelized for many types of models. This is true for many algorithms where there is shared state or dependent variables that must be maintained and updated centrally.

[5] The Hadoop MapReduce framework in earlier (pre-version 2) Hadoop releases has a single master service called a JobTracker and several worker services called TaskTrackers, one per node in the cluster. When you submit a MapReduce job to the JobTracker, the job is placed into a queue and then runs according to the scheduling

rules defined by an administrator. The JobTracker manages thes assignment of map and reduce tasks to the TaskTrackers.

[1] With Hadoop 2, a new resource management system is in place called YARN (Yet Another Recourse Manager). YARN provides generic scheduling and resource management services so that you can run more than just MapReduce applications on your Hadoop cluster. But the JobTracker/TaskTracker architecture run only on MapReduce.

Hadoop offers great promise to organizations looking to gain a competitive advantage from data science. Hadoop lets organizations collect a massive amount of data that can later be used to extract insights of immense business value for use cases that include fraud detection, sentiment analysis, risk assessment, predictive maintenance, churn analysis, user segmentation and many more. But deploying Hadoop can be extraordinarily complex and time consuming, making it difficult to gain the insights.

The key point for Hadoop to truly have a broad impact on the IT industry and live up to its true potential, is the compatibility with the older technologies: It had to support SQL; integrate with and extend the RDBMS; and enable IT professionals who lack skills in using Java Map/Reduce to take advantage of its features. To overcome this disadvantage, Hive, one of the Hadoop ecosystem tools can be used. Apache Hive is an ETL (Extract, Transform, and Load) and data warehouse solution under the Hadoop ecosystem. Hive is an open-source data warehousing solution built on top of Hadoop, which facilitates easy data summarization, ad-hoc queries, and the analysis of large datasets stored in various databases and file systems that integrate with Hadoop.

1: The original GFS (Google Filesystem) is not the same as what has become Hadoop. GFS was a framework while Hadoop become the translation of the framework put into action.

2: Batch processing is used to process data in batches. It reads data from the input, processes it, and writes it to the output. Apache Hadoop is the most well-known and popular open source implementation of the distributed batch processing system using the MapReduce paradigm.

CHAPTER 2: INTRODUCING HIVE

As much as the Hadoop ecosystem evolves and provides exceptional means to access new types of data and structures, we cannot deny the influence and purpose of traditional relational systems. Relational systems and especially the data access methods employed by these systems have served as a valuable tool for over 30 years. The SQL query language brought data access to the masses by abstracting away concepts such as data location and instead allowed developers to focus on how the data will be presented. SQL excels as a declarative language in which you clearly specify what you want to do in simple English language syntax. You SELECT, JOIN SUM, data FROM a source WHERE the values equals, or does not equal, something. The developer does not have to worry about where the data resides on disk, and the structure of the data is already predefined in a relational format consisting of tables with rows and columns.

The attraction of SQL to the Hadoop world was not in its ability to consume data schematized as rows and columns or its efficient use of indexes and statistics but instead, SQL's popularity as a data query tool. Simply put – a lot of people who accessed data knew how to write SQL statements. Java is the language of MapReduce so early in the Hadoop adoption, if you needed to perform computation and access data in Hadoop, you had to write Java code, specifically MapReduce programs. Large companies like Facebook came to realize that you could not hire enough Java developers to write the amount of MapReduce code needed to take full advantage of the quantity of data stored in HDFS. In order to increase adoption and ease of use, developers needed to abstract away MapReduce complexity in favor of a more demotic programming language.

The answer is Apache Hive and the HiveQL.

[6] Apache Hive is a data warehouse for Apache Hadoop. It was created at Facebook by the Data Infrastructure Team, led by Jeff Hammerbacher and is being used to run thousands of jobs on the cluster with hundreds of users, for a wide variety of applications. Hive was originally designed as a translation layer on top of Hadoop Map/Reduce. Hive takes large amount of unstructured data and place it into a structured view, which can be used by business analysts. Hive supports use cases such as Ad-hoc queries, summarization, and data analysis. HiveQL can also be exchange with custom scalar functions means user defined functions (UDF's), aggregations (UDFA's) and table functions (UDTF's).

Hive, a now top-level Apache project and a vital component within the Apache Hadoop ecosystem, drives several leading big-data use cases and has brought Hadoop into data centers across the globe. [1] It is the entry point into an exceedingly complex data storage environment. Hive is the bridge to the traditional RDBS world and provides an SQL dialect known as Hive Query Language (Hive QL), which can be used to perform SQL-like tasks. As of this, it is being used and developed by a number of companies like Amazon, IBM, Yahoo, Netflix, Financial Industry Regulatory Authority (FINRA) and many others.

[7] The Hive Query Language (HiveQL) has similar semantics and functions as standard SQL in the relational database, so that experienced database analysts can easily get their hands on it. [8]Traditional SQL queries must be implemented in the MapReduce Java API to execute SQL applications and queries over distributed data. Hive provides the necessary SQL abstraction to integrate SQL-like queries (HiveQL) into the underlying Java without the need to implement queries in the low-level Java API. Since most data warehousing applications work with SQL-based querying languages, Hive aids portability of SQL-based applications to Hadoop. Hive's query language can run on different computing engines, such as MapReduce, Tez and Spark.

[1]Hive also makes possible the concept known as enterprise data warehouse (EDW) augmentation, a leading use case for Apache Hadoop, where data warehouses are set up as RDBMSs built specifically for data analysis and reporting.

[9]The versatility and power of Hadoop lies in its ability to store and process any kind of unstructured, semi-structured or structured data. Hive allows the users to create a metadata layer on top of this data and access it using a SQL interface. As much as it is familiar to the end user for its interface, it is different in terms of how it handles the underlying data. Hive does not take control of how data is persisted to disk or its lifecycle. Users can first store any kind of data in HDFS, in its inherent format, and then define metadata to read it independently of the data. Hive makes it easier to manage and process data with a variety of tools with this flexibility. It is important the user should remember that Hive is not a database; it is a human friendly, familiar interface to query the underlying data files that are stored on HDFS.

CHAPTER 3: SETTING UP THE HIVE ENVIRONMENT

This chapter will introduce how to install and set up the Hive environment in the cluster. It also covers the usage of basic Hive commands.

INSTALLING APACHE HIVE ON A MULTINODE CLUSTER

To introduce the Hive installation, we will use Hive version 3.1.2 as our client. We can download it from the Apache mirror (http://apachemirror.wuchna.com/hive/hive-3.1.2/) or via the terminal writing the following code:

```
wget https://downloads.apache.org/hive/hive-3.1.2/apache-hive-3.1.2-bin.tar.gz
```

The pre-installation requirements for the installation are as follows:

- JDK 1.8
- Existing up and running Hadoop cluster where Hadoop-3.2.1 deployed and configured with 3 DataNodes.
- Ubuntu 16.04

Setup Installation Guide

1. Download and untar Apache Hive
2. Edit .bashrc file
3. Edit hive-config.sh file
4. Create Hive directories in HDFS
5. Initiate Derby database
6. Start hiveserver2 and beeline interface

----

1. Download and untar Apache Hive

Select a healthy DataNode with high hardware resource configuration in the cluster to install Hive. Extract the previously downloaded apache-hive-3.1.2-bin.tar.gz from the terminal and rename as a hive.

2. Edit .bashrc file

The file .bashrc is the one where we have stored the variables for Hadoop.

```
sudo gedit .bashrc
```

Insert the following lines:

```
export HIVE_HOME = /home/hadoopuser/hive (the path where hive is installed)

export PATH=$PATH:$HIVE_HOME/bin

source ~/.bashrc
```

3. Edit hive-config.sh file

Hive-config.sh file is in the bin directory within your Hive installation dir

```
sudo gedit $HIVE_HOME/bin/hive-config.sh
export HADOOP_HOME=/home/hadoopuser/hadoop
```

4. Create Hive directories in HDFS

We need to create two folders, named warehouse and tmp, inside HDFS and give them permissions like following:

```
hdfs dfs –mkdir /tmp
hdfs dfs –chmod g+w /tmp
hdfs dfs –mkdir -p /user/hive/warehouse
hdfs dfs –chmod g+w / user/hive/warehouse
hdfs dfs –ls /user/hive
```

5. Initiate Derby database

By default, Hive uses the Derby (http://db.apache.org.derby/) database as the metadata store. Is the one we have use in this installation. Hive can also use other relational databases, such as Oracle, PostgreSQL or MySQL as the metastore.

```
$HIVE_HOME/bin/schematool –initSchema –db Type derby
```

If the initiation of the Derby database fails, is probably the guava jar file incompatibility between the Hadoop version and the Hive version.

- Locate the guava jar file in the Hive lib directory: *ls $HIVE_HOME/lib*
- Locate the guava jar file in the Hadoop lib directory: *ls $HADOOP_HOME/share/Hadoop/hdfs/lib*
- We will see that the version are different. We have to remove the guava version inside the lib folder of Hive: *rm $HIVE_HOME/lib/guava-19.0.jar*
- Copy the guava version from the lib folder of Hadoop inside the lib folder of Hive: *cp $HADOOP_HOME/share/Hadoop/hdfs/lib/guava-27.0-jve.jar $HIVE_HOME/lib/*

6. Start hiveserver2 and beeline interface

There are multiple options available to connect with Hive to execute HQL queries, data loading etc. Hive CLI can be used by default but Hive should be installed on the same machine or the DataNode in the cluster. It connects directly to the Hive Driver. Hive CLI won't be used in real-time/ production environment. since it's depreciated from Hive 2.0 onwards. HiveServer2 (HS2) is a service that enables clients to execute queries against the Hive. HS2 supports multi-client concurrency and authentication. We don't need any separate configuration for HiverServer2 . If we can access Hive CLI from the terminal without any issue, HiverServer2 service can be started by executing following command in a separate terminal.

$HIVE_HOME/bin/hiveserver2

And access the web ui of HiverServer2 from browser at default port 10002.

Beeline is another thin client CLI to execute queries via HiveServer2 which support concurrent client connection and authentication. Beeline can be leveraged by multiple user from multiple node in the cluster to execute queries. And it will be the preferred interface for this essay. We can access the beeline interface:

$HIVE_HOME/bin/beeline -n myjio -u jdbc:hive2://localhost:10000

[7] Hive first started with hiveserver1. However, this version of Hive server was not very stable and sometimes suspended or blocked the client's connection quietly. Since v0.11.0, Hive has included a new thrift server called hiveserver2 to replace the first server. hiveserver2 has an enhanced server, designed for multiple client concurrency and improved authentication. It also recommends using beeline as the major Hive command-line interface instead of the old hive command. The primary difference between the two versions of servers is how the clients connect to them. Hive is an Apache-Thrift-based client and beeline is a JDBC client. The hive command directly connects to the Hive drivers, so we need to install Hive libraries on the client. However, beeline connects to hiveserver2 through JDBC connections without installing Hive libraries on the client. That means we can run beeline remotely from outside the cluster.

WHAT IS INSIDE HIVE?

[4][10] The core of a Hive library distribution contains three parts. The main part is the Java code itself. Multiple JAR files such as hive-exe.jar and hive-metastore.jar can be found under the $HIVE_HOME/lib directory.

The $HIVE_HOME/bin directory contains executable scripts that launch various Hive services, including the hive command-line interface (CLI). Although CLI is the most popular way for someone to use Hive, in this essay we will use beeline (also a command line) as our interface. Hive also has other components. A Thrift service provides remote access from other processes. Access in this case is provided by using JDBC and ODBC drivers. They are implemented on top of the Thrift service.

All Hive installations finally require a metastore service, which Hive uses to store table schemas and other metadata. It is typically implemented using tables in a relational database. By default, Hive uses a built-in SQL server, Derby, which provides limited, single process storage. But it the perfect fit for educational and academicals purposes, so this is the option we will use in this essay too.

Finally the conf directory contains the files that configure Hive. Hive has a number of configurations properties, some of them we have already mentioned on the installation section. These properties control features such as the metastore, various optimazations and safety controls.

USING HIVE IN THE CLOUD

[7] All major cloud service providers, such as Amazon, Microsoft and Google, offer matured Hadoop and Hive as services in the cloud. Using the cloud version of Hive is very convenient. It requires almost no installation and setup. Amazon EMR is the earliest Hadoop service in the cloud. However, it is not a pure open source version since it is customized to run only on AWS. Hadoop enterprise service and distribution providers, such as Cloudera and Hortonworks also provide tools to easily deploy their own distributions on different public or private clouds. Although Hadoop was first built on Linux, Hortonworks and Microsoft have already partnered to bring Hadoop to the Windows based platform and cloud successfully. The consensus among all the Hadoop cloud service providers here is to allow enterprises to provision highly available, flexible, highly secure, easily manageable and governable Hadoop clusters with less effort and little cost.

CHAPTER 4: HIVE ARCHITECTURE

In contrast to its popularity and widespread use, Hive's performance was not ideal, reason being the data was not efficiently stored or query processing required multiple MapReduce jobs. Business users felt that Hive is incapable of handling interactive data warehouse workloads, for instance queries would take several hours to return a result and often time users would think the cluster has gone down. The fact that the query would potentially run against petabytes of data was little consolation to end users who just wanted interactive data analytics. Of course, you can horizontally scale your cluster for additional compute resources and speed up the processing, but that would not be a long-term approach and strategy for increasing Hadoop adoption.

The need of a low-latency SQL engine was urgent.

[9] The original Hive on MapReduce open sourced by Facebook required a much needed reengineering to provide competitive functionality to the users. As a result, as of Hive 1.2.1 you have a choice to run as batch using MapReduce v2, or as interactive using Tez or leverage in-memory processing using Spark. Although Tez is now the default execution engine, in this essay we will use the MapReduce engine for our case study.

It is true that Hive is a little complicated but its complexity is surmountable and will be familiar to those who make a living accessing data. Also, like any software development project, Hive is constantly changing and changing fast. We will try to cover in this chapter the core elements of Hive as for the 3.1.2 release.

HIVE OVERVIEW

The following diagram is the architecture view of Hive in the Hadoop ecosystem. The Hive metadata store (metastore) can use either embedded, local or remote databases. The thrift server is built on Apache Thrift Server technology. With its latest version, hiveserver2 is able to handle multiple concurrent clients, support Kerberos, LDAP and provide better options for JDBC and ODBC clients, especially for metadata access.

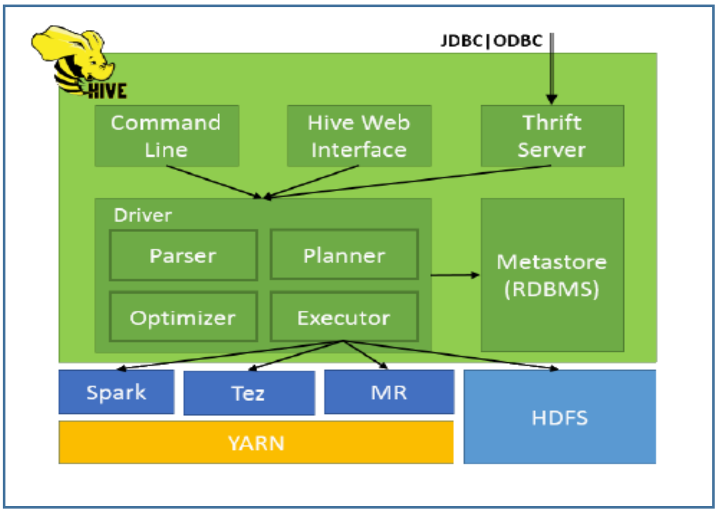


Figure 4-1. Hive Architecture

As we have already mentioned, at the bottom of Hive architecture lies the Hadoop Distributed File System (HDFS) and above this, on the earlier versions of Hive, the MapReduce system, [11] which is the processing layer of Hadoop. MapReduce programming model is designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. It is the heart of Hadoop. Hive queries are converted to MapReduce code and executed using the MapReduce infrastructure, the JobTracker and TaskTracker.

[12]In a cluster architecture, Apache Hadoop YARN (Yet Another Resource Negotiator) sits between HDFS and the processing engines being used to run applications. It combines a central resource manager with containers, application coordinators and node-level agents that monitor processing operations in individual cluster nodes. YARN can dynamically allocate resources to applications as needed, a capability designed to improve resource utilization and application performance compared with MapReduce's more static allocation approach. The addition of YARN significantly expanded Hadoop's potential uses. The original incarnation of Hadoop closely paired the Hadoop Distributed File System (HDFS) with the batch-oriented MapReduce programming framework and processing engine, which also functioned as the big data platform's resource manager and job scheduler. As a result, Hadoop 1.0 systems could only run MapReduce applications -a limitation that Hadoop YARN eliminated.

We can think of the HDFS and MapReduce systems as being parts of the Apache Hadoop operating system, with the Hive as higher-level functions or applications.

To understand the differrent componets of Hive, besides the metastore, the other componets of Hive could be broadly classified as Hive clients and Hive servers. Hive servers provide interfaces to make the metastore available to external applications and check for users’s authorization and authentication, and Hive clients are various applications used to access and execute Hive queries on the Hadoop cluster.

With this in mind and moving up the diagram, [11] we find the Hive Driver, part of Hive Services which is responsible for receiving the queries submitted by Thrift, JDBC, ODBC, CLI, Web UL interface, which are the part of Hive client. It includes compiler, optimizer, parser and executor used to break down the Hive query language statements. [13] The Hive Driver may choose to execute HiveQL statements and commands locally or spawn a MapReduce job, depending on the task at hand. The Hive Driver stores table metadata in the metastore and its database.

[14] Hive Parser parses the query. It performs semantic analysis and type-checking on the different query blocks and query expressions by using the metadata stored in metastore and the Planner generates an execution plan.

The execution plan created by the Planner is the DAG (Directed Acyclic Graph), where each stage is a map/reduce job, operation on HDFS, a metadata operation.

[14] Optimizer performs the transformation operations on the execution plan and splits the task to improve efficiency and scalability.

[15] The Hive Query executor connects to Hive (or Impala) and performs one or more user-defined Hive (or Impala queries) each time it receives an event record. The Hive Query executor waits for each query to complete before continuing with the next query for the same event record. It also waits for all queries to complete before starting the queries for the next event record. Depending on the speed of the pipeline and the complexity of the queries, the wait for query completion can slow pipeline performance.

[9] As beneficial as Hive was at providing a SQL abstraction layer for running MapReduce, there were still some major limitations. One of them was the ability for clients to connect to the metastore using standard ODBC and JDBC connections. A feature we take for granted in traditional relational database systems. The open source community addressed this limitation by creating the Hiveserver1. Hiveserver1 allowed clients to access the metastore using ODBC connections.

[9] But there were still limitations with hiveserver1. Primarily, the limitations included user concurrency restrictions as well as security integration with LDAP. Each of these components were solved with the implementation of Hiveserver2. The Hiveserver2 architecture is based on a Thrift Service and any number of sessions comprised of a driver, compiler (parser and planner) and executor. The metastore is also a key component of Hiveserver2. Figure 4-2 shows Hiveserver2 basic architecture.

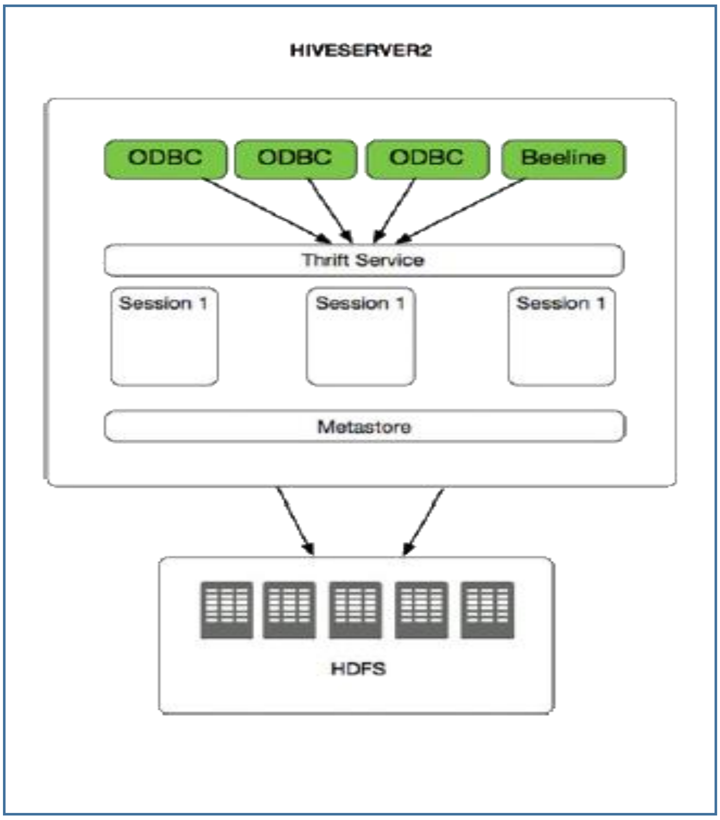


Figure 4-2

[9] Hiveserver2 supports Kerberos custom authentication, as well as pass-through LDAP authentication. All connection componets –JDBC, ODBC and Beeline, have the ability to use any one of these authentications methods.

Beeline it's a JDBC client that is based on the SQLLine CLI. The Beeline shell works in both embedded mode as well as remote mode. In the embedded mode, it runs an embedded Hive (similar to Hive CLI) whereas remote mode is for connecting to a separate HiveServer2 process over Thrift.

Start the Hiveserver2:

```
$HIVE_HOME/bin/hiveserver2
```

Start the Beeline interface:

```
$HIVE_HOME/bin/beeline -n myjio -u jdbc:hive2://localhost:10000
```

Running Hive queries in Beeline interface:

Metastore is the central repository of Apache Hive metadata. It stores metadata for Hive tables (like their schema and location) and partitions in a relational database. It provides client access to this information by using metastore service API. [7] Hive's metadata structure provides a high-level, table-like structure on top of HDFS. It supports three main data structures, tables, partitions and buckets. The tables correspond to HDFS directories and can be divided into partitions, where data files can be divided into buckets. Hive's metadata structure is usually the Schema of the Schema-on-Read concept on Hadoop, which means we do not define the schema in Hive before we store data in HDFS. Applying Hive metadata after storing data brings more flexibility and efficiency to our data work.

[14] Hive metastore consists of two fundamental units:

1. A service that provides metastore access to other Apache Hive services.
2. Disk storage for the Hive metadata which is separate from HDFS storage.

By default, Hive includes the Apache Derby RDBMS (stored on the local file system) configured with the metastore in what's called embedded mode. Embedded mode means that the Hive Driver, the metastore service and Apache

Derby are all running in one Java Virtual Machine (JVM). [13] The embedded mode of Hive has the limitation that only one session can be opened at a time from the same location on a machine as only embedded Derby database can get lock and access the database files on disk.

[13] To solve this limitation, a separate RDBMS database runs on same node. The metastore service and Hive service still run in the same JVM. This configuration mode is named local metastore. Here, local means the same environment of the JVM as well as the service in the same node.

There is one more configuration where one or more metastore servers run in a separate JVM process to the Hive service connecting to a database on a remote machine. This configuration is named remote metastore.

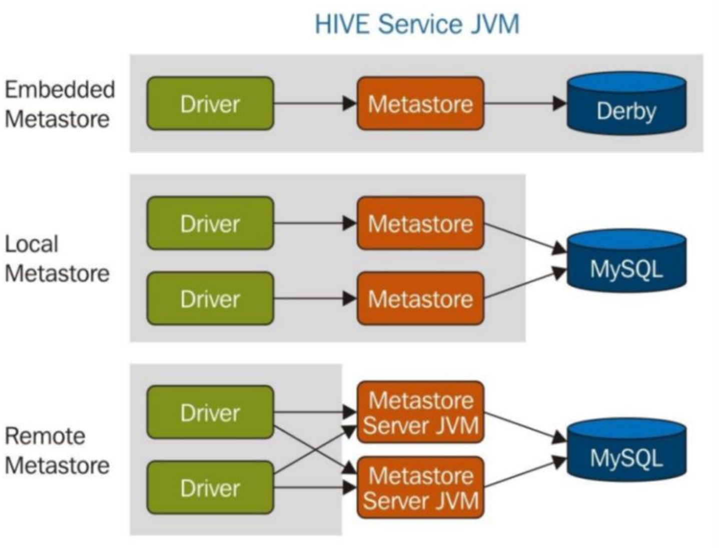


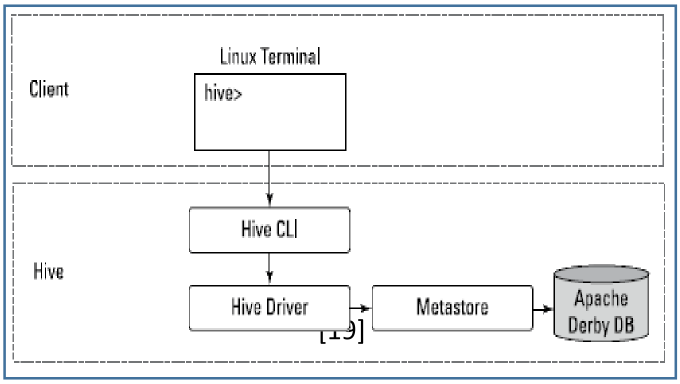
Figure 4- 3

If the metastore service is down or unavailable, then clients would not be able to run any HiveQL as metastore data is not accessible.

Hive supports 5 backend databases which are as follows: Derby, MySQL, MS SQL Server, Oracle, Postgres.

The key to application support is the Hive Thrift Server which enables a rich set of remote clients to submit requests to Hive, using a variety of programming languages other than Java (PHP or Python, for example), and retrieve results. [16] Hive Thrift Server is built on Apache Thrift (http://thrift.apache.org/), therefore it is sometimes called the Thrift server although this can lead to confusion because of the newer service mentioned earlier, Hiveserver2, which is also built on Thrift. Since the introduction of HiveServer2, Hive Thrift Server has also been called Hiveserver1.

[4]To continue with the Hive architecture, note that Hive includes a Command Line Interface (CLI), where we can use a Linux terminal window to issue queries in either interactive or batch mode and administrative commands directly to the Hive Driver. [6]CLI is the command line interface acts as Hive service for DDL (Data definition Language) operations. All drivers communicate with Hive server and to the main driver in Hive services as shown in above architecture diagram.



[19]

Figure 4-4

Figure 4-4 illustrates the componets of Hive that are needed when running the CLI on a Hadoop cluster.

To run the Hive CLI, you execute the hive command and specify the CLI as the service you want to run.

```
$HIVE_HOME/bin hive –service cli
```

Navigating within the CLI is straight-forward, especially if you are used to other database systems. Keep in mind that Hive was developed based on MySQL so syntax and data types between the two are quite similar.

```
hive> CREATE DATABASE SoccerData;
```

Be sure to end all commands with a semicolon (the same applies to Beeline).

```
hive> use SoccerData;
hive> CREATE TABLE Events;
```

Note that Hive is not case sensitive, but we prefer to use same syntax as SQL.

Finally, the Hive Web Interface is an alternative to using the Hive command line interface. Using the web interface is a great way to get started with Hive. [16] The Hive Web Interface, abbreviated as HWI, is a simple graphical user interface (GUI). HWI is only available in Hive releases prior to 2.2.0. Any user with a web browser can work with Hive. This has the usual web interface benefits. In particular, a user wishing to interact with Hadoop or Hive requires access to many ports. A remote or VPN user would only require access to the Hive Web Interface running by default on 0.0.0.0 tcp/9999.

[11]Finally, Hive can operate in two modes depending on the size of data nodes in Hadoop.

These modes are:

- Local mode
- Map reduce mode

Local mode: if the Hadoop installed under pseudo mode with having one data node we use Hive in this mode. If the data size is smaller in term of limited to single local machine, we can use this mode. Processing will be very fast on smaller data sets present in the local machine

MapReduce mode: if Hadoop is having multiple data nodes and data is distributed across different node we use Hive in this mode. It will perform on large amount of data sets and query going to execute in parallel way. Processing of large data sets with better performance can be achieved through this mode.

JOB EXECUTION FLOW IN HIVE

[11] The following diagram demonstrates step by step the job execution flow in Hive with Hadoop.

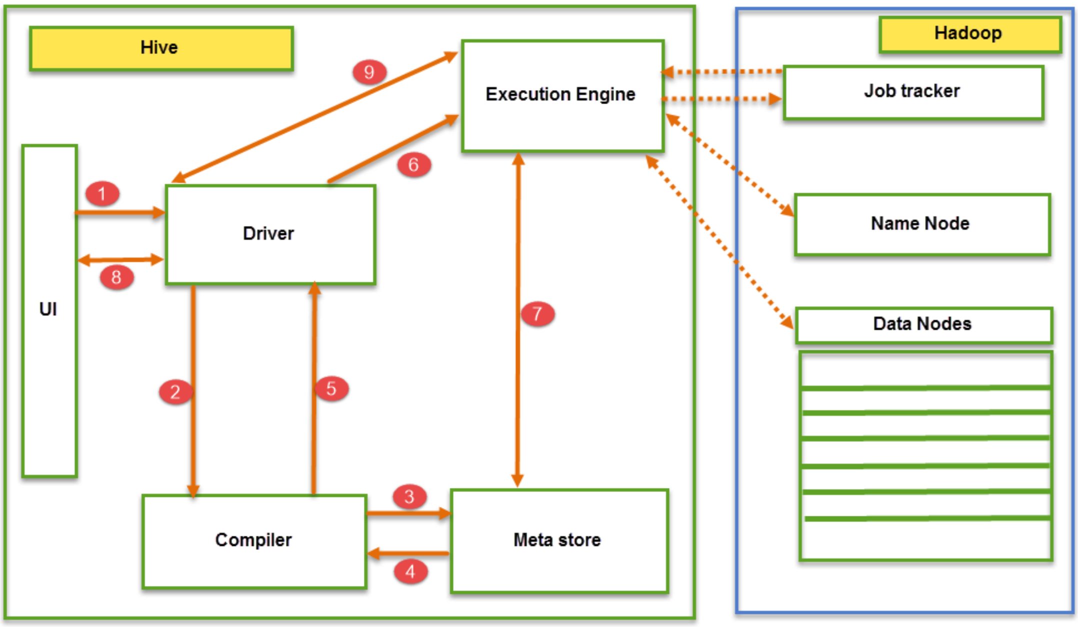


Figure 4-5

[17] **Step-1: Execute Query –**

Interface of the Hive such as Command Line or Web user interface delivers query to the driver to execute. In this, UI calls the execute interface to the driver such as ODBC or JDBC.

Step-2: Get Plan –

Driver designs a session handle for the query and transfer the query to the compiler to make execution plan. In other words, driver interacts with the compiler.

Step-3 & 4: Get Metadata –

In this, the compiler transfers the metadata request to any database and the compiler gets the necessary metadata from the metastore.

Step-5: Send Metadata –

Metastore transfers metadata as an acknowledgement to the compiler.

Step-6: Send Plan –

Compiler communicating with driver with the execution plan made by the compiler to execute the query.

Step-7: Execute Plan –

Execute plan is sent to the execution engine by the driver.

- Execute Job
- Job Done
- Dfs operation (Metadata Operation)

Step-8: Fetch Results –

Fetching results from the driver to the user interface (UI).

Step-9: Send Results –

Result is transferred to the execution engine from the driver. Sending results to Execution engine. When the result is retrieved from data nodes to the execution engine, it returns the result to the driver and to user interface (UI).

TEZ EXECUTION ENGINE

What we have described so far are the core elements of Hive architecture as it was first introduced by Facebook. But the limitations of Hive, as we have discuss so far, led to the introduction of two new engines (Hive 1.2.1), one interactive named Tez and the other a leverage in-memory processing, named Spark.

We will cover in this essay the core elements of the execution engine Tez.

[9] When Hadoop was conceived, only MapReduce engine was available and it was a batch operation. It meant that MapReduce had a unique ability to crunch massive amounts of data but processing that data was a monumental task, which not only took up most of your cluster resources, but was also note expected to finish quickly. MapReduce is Java so to access data on Hadoop you had to know Java, specifically how to write Java MapReduce code. As we have already mentioned, Facebook solved this problem by creating Hive and HiveQL. Although this was a huge step in providing access to Hadoop, did nothing for the problem that MapReduce being a batch operation. Users wrote similar SQL code on Hadoop (via Hive) but the performance was nothing like the one they were used to with RDBM's.

Some early Hadoop distributors solved this problem by creating data access architectures, which accessed data within Hadoop, but processed the data outside of Hadoop. Some of these early SQL-in-Hadoop solutions (Apache Impala and Apache Hawq to name the first ones) were based on popular MPP architectures, which utilized parallel processing to gather and execute the data. The goal is to execute in-memory processing for fastest results and avoid costly disk IO operations.

These early SQL-in-Hadoop solutions solved many limitations present with Hive on MapReduce, most importantly performance and ANSI SQL compliance. The proble with early solutions was that they failed to execute on larger data sets. This is because once memory capacity is full, the data needs to spill to disk resulting to IO bottlenecks. The other problem was that they were not Hive or open source, so these solutions included additional costs. But early Hadoop adopters had been using Hive for quite a time and, instead of looking for a new SQL environment, they would prefer to make Hive better.

So the open source community launched what was marketed the Stinger Initiative. This initiative aimed to provide interactive SQL-in-Hadoop natively in Hive. In order to accomplish this a new engine was required. This new engine was Tez.

Tez became the new platform for Hive execution. MapReduce is still supported for Hive execution but Tez is now the default engine when running Hive queries in Hadoop.

So Tez is a distributed execution framework and alternative of MapReduce to process data applications on Hadoop. It is built on Hadoop YARN, and can execute complex Directed Acyclic Graphs, short form DAG, of general data processing tasks. In simpler words, it can be thought of as a more advanced, optimized, flexible, and powerful successor of our conventional MapReduce framework.

Now what Tez can provide us?

Tez provides us performance gains over MapReduce execution. It can give us optimal resource management and can perform more complex tasks in less time than MapReduce. It can also provide us flexible input processor output runtime model i.e it can construct runtime executors dynamically by connecting different inputs, processors, and outputs. Overall, Tez engine is better than MapReduce in almost all fields. It may be in end user empowerment or execution performance, Tez will always surpass MapReduce in all aspects.

And more importantly, Tez can run any MapReduce job without any modification.

So this allows us easy migration of projects also. Now that being said, the question arises, how Tez is better than MapReduce? What exactly Tez does? And how Tez works backstage, which makes it better than MapReduce?

Actually, there are a number of factors involved in this. First is Tez execution plan. Tez execution plan is similar like of Spark's, that is, it creates DAG of task from the job. Tez represents data processing as a data flow graph, with the graph vertices representing application logic and the edges presenting the actual movement of data. This graph allows users to involuntarily express complex query logic. Tez will read the full job, and out of that job, create a DAG of task setting the dependencies and parallelism of various tasks, and then execute the plan to produce output. This enables the YARN framework to allocate resources more intelligently, and Tez has a simple Java API to express this DAG of data processing.

The second factor is number of reduced stages. To understand this, let us take an example of it.

Suppose we have a query in which there is a need to have multiple reduce tasks. In that case, what our MapReduce framework will do is, it will break the plan apart and create one MapReduce job per reduce task. It means the number of MapReduce jobs will be equal number of reduce tasks. And undoubtedly, the more the number of MR jobs, the more will be disk reads and writes, because all the MR jobs will then run in chain fashion and needs to be scheduled one after another.

So the output of first MR job will be written to disk in HDFS. Then the next job will read the data from disk. And after completion, it again writes the data to disk. And these rewrite processes will go on till all the jobs in chain complete, and no developer in the world would want these many reads and writes in his job. It will take a boat load of time to complete this full job.

But Tez has addressed this issue by not creating a separate MR job per reduce task. Rather, it has linked all those reduce tasks directly in a single job, and data can now be pipelined from a reducer to next reducer directly, without the need of any temporary HDFS storage. So apparently, less reads and writes in HDFS means less execution time.

Moving towards the third factor, third factor is extension of the previous one, that is, disk writes of MapReduce versus in-memory competition in Tez. Conventionally, in MapReduce, the data is shuffled across nodes regardless of the data size. Even if the data is small and can be processed entirely in memory, even then also it has to get shuffled across nodes. But Tez allows for small datasets to be handled fully in memory. So small datasets can be entirely computed in memory itself without having to shuffle it across the nodes.

Summarizing, these were some important factors which makes Tez better than MapReduce. Although it is not this essay scope to discuss the Tez engine, it is important to know that now few SQL jobs still utilize MapReduce engine.

CHAPTER 5: DATA TYPES IN HIVE

DATA TYPES

Hive data types are categorized into two types: primitive and complex. Hive supports many primitive data types that are similar to relational databases, such as INT, SMALLINT, TINYINT, BIGINIT, BOOLEAN, FLOAT and DOUBLE. In addition to primitive data type, Hive also supports few complex data types, such as ARRAY, MAP, STRUCT and UNION.

Primitive data types

Hive supports large number of primitive data types, which are divided into the four following categories:

- Numeric data types: Store positive and negative exaxt and floating-point numbers
- String data types: Store alphanumeric data in strings
- Date/Time data types: Store temporal values
- Miscellaneous data types: Boolean (True or false) and Binary (Variable length array of binary data)

A complete list of primitive data types is very well documented on the Apache web site (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types)

[4] It is useful to remember that each of these types is implemented in Java, so the particular behavior details will be exactly what you would expect from the corresponding Java type. For example, STRING is implemented by the Java String, FLOAT is implemented by Java float, etc.

Note that Hive does not support “character arrays” (strings) with maximum-allowed lengths, as is common on other SQL dialects. Relational databases offer this feature as a performance optimization; fixed-length records are easier to index, scan etc. In the “looser” world in which Hive lives, where it may not own the data files and has to be flexible on file format, Hive relies on the presence of delimiters to separate fields. Also, Hadoop and Hive emphasize optimizing disk reading and writing performance, where fixing the lengths of column values is relatively unimportant.

Complex data types

Complex types are also known as collection types. [2] These consist of more than one element of primitive data types and are internally implemented using native serializers and deserializers. They allow you to store the data in collection format without having to break it into further individual fields, as you would do in a normalized schema in a relational database. The complex data types are quite useful to map real-world data to a schema laver. The following are the complex types supported by Hive:

- STRUCT: The struct type in Hive is analogous to the STRUCT in C programming language. It is a record type that holds a set of named fields that can be of any primitive data types. Fields in the STRUCT type are accessed using the dot (.) notation, i.e.: .col_name1.

Syntax: STRUCT<col_name1 : data_type, col_name2 : data_type,… >

- MAP: The map data type Contains key-value pairs. In Map, elements are accessed using keys. For example, if a column name is of type Map: ‘firstname’ -> ‘john’ and ‘lastname’ -> ‘roy’ then the last name can be accessed using the name [‘lastname’]
- ARRAY: This is an ordered sequence of similar elements. It maintains an index in order to access the elements; for example, an array day, containing a list of elements [‘Sunday’, ‘Monday’, ‘Tuesday’]. In this, the first element ‘Sunday’ can be accessed using day[0] and similarly the third element can be accessed using day[2].
- UNIONTYPE: This data type enables us to store different types in the same memory location. It is an efficient way of using the same memory location for multipurpose. It is similar to Unions in the C programming language. You can define a union type with many data types, but at a time, only one data type can be hold by it.

Syntax: UNIONTYPE<data_type, data_type, …>

For MAP, the type of keys and values are unified. However, STRUCT is more flexible. STRUCT is more like a table, whereas MAP is more like an ARRAY with a customized index.

An example of how to use complex data types:

We have to create a customer table with name as the struct data type, the following command can be used:

CREATE TABLE customer (id INT, name

STRUCT<first_name:STRING, last_name:STRING>);

Here, the column name is of the type STRUCT with two fields –first_name and last_name- then the first_name field can be referenced using name.first_name. Similarly the last_name field can be referenced using name.last_name

Most RDBMs does not support such data types, because using them tends to break normal form. For example, in traditional data models, structs might be captured in separate tables, with foreign key relations between the tables, as appropriate.

[4] A practical problem with breaking normal form is the greater risk of data duplication, leading to unnecessary disk space consumption and potential data inconsistencies, as duplicate copies can grow out of sync as changes are made.

However, in Big Data systems, a benefit of sacrificing normal form is higher processing throughput. Scanning data of hard disks with minimal “head seeks” is essential when processing terabytes to petabytes of data. Embedding collections in records makes retrieval faster with minimal seeks. Navigating each foreign key relationship requires seeking across the disk, with significant performance overhead.

Similar to SQL, HiveQL supports both implicit and explicit type conversion. Primitive type conversion from a narrow to a wider type is known as implicit conversion. However, the reverse conversion is not allowed. All the integral numeric types, FLOAT and STRING can be implicitly converted to DOUBLE and TINYINT, SMALLINT and INT can all be converted to FLOAT. There is a data type cross-table describing the allowed implicit conversion between every two types, which can be found at:

https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types

Hive also supports all the well-known Relational Operators, Arithmetic Operators, Logical Operators and also String Operators and Operators on Complex Types.

HIVE DATA DEFINITION LANGUAGE (DDL)

Hive’s Data Definition Language (DDL) is a subset of HQL statements that describe the Hive data structure by creating, deleting or altering schema objects as databases, tables, partitions and buckets. Most DDL statements start with the CREATE, DROP or ALTER keywords. The syntax of HiveQL is very similar to SQL DDL and is case-insensitive.

DDL Command	Use With
CREATE	Database, Table
SHOW	Databases, Tables, Table Properties, Partitions, Functions, Index
DESCRIBE	Database, Table, view
USE	Database
DROP	Database, Table
ALTER	Database, Table
TRUNCATE	Table

[4] When you write data to a traditional database, the database has a total control over the storage. The database is the “gatekeeper”. An important implication of this control is that the database can enforce the schema as data is written. This is called schema on write.

But Hive has no such control over the underlying storage. There are many ways to create, modify and even damage the data that Hive will query. Therefore, Hive can only enforce queries on read. This is called schema on read.

Now that we are familiar with various data types in the world of Hive, we can look at how these can be used to read data. A Hive data model contains of: Databases, Tables, Partitions and Buckets.

Database

[18] The database in Hive describes a collection of tables that are used for a similar purpose or belong to the same groups. A database in Hive is a namespace that holds metadata information for a set of tables. [7] In addition, DATABASE has a name alias, SCHEMA, meaning they are the same thing in HiveQL. At the filesystem level, it is a directory under which all internal tables that belong to that namespace are stored. The syntax for this statement is as follows:

CREATE DATABASE|SCHEMA [IF NOT EXISTS] human_resources

The above query is executed to create a database named human_resources

Hive will create a directory for each database. Tables in that database will be stored in subdirectories of the database directory. The exception is tables in the default database, which doesn't have its own directory.

Table

[6] Apache Hive tables are the same as the tables present in a Relational Database. The table in Hive is logically made up of the data being stored. And the associated metadata describes the layout of the data in the table. We can perform filter, project, join and union operations on tables. In Hadoop data typically resides in HDFS, although it may reside in any Hadoop filesystem, including the local filesystem or S3. But Hive stores the metadata in a relational database and not in HDFS.

Each table maps to a directory and all the data in the table is stored in this hive user-manageable directory (full permission). This kind of table is called internal, or managed table. It means that Hive moves the data into its warehouse directory. [7] When data is already stored in HDFS, an external table can be created to describe the data. It is called external because the data in the external table is at an existing location outside the warehouse directory. [1] When an internal table is dropped, its data is deleted together. However, when an external table is dropped, the data is not deleted. It is quite common to use external tables for source read-only data or sharing the processed data to data consumers giving customized HDFS locations. On the other hand, the internal table is often used as an intermediate table during data processing, since it is quite powerful and flexible when supported by Hive QL.

```
CREATE TABLE managed_table (dummy STRING);

LOAD DATA INPATH '/user/kyriskou/data.txt' INTO table managed_table;
```

This will move the file hdfs://user/ kyriskou/data.txt into Hive's warehouse directory for the managed_table table, which is hdfs://user/hive/warehouse/managed_table.

Further, if we drop the table using:

```
DROP TABLE managed_table
```

Then this will delete the table including its data and metadata. The data no longer exists anywhere. This is what it means for HIVE to manage the data.

The external table in Hive behaves differently. We can control the creation and deletion of the data. The location of the external data is specified at the table creation time:

```
CREATE EXTERNAL TABLE external_table (dummy STRING);

LOCATION '/user/ kyrisko /external_table';

LOAD DATA INPATH '/user/ kyrisko /data.txt' INTO TABLE external_table;
```

Now, with the EXTERNAL keyword, Apache Hive knows that it is not managing the data. So it does not move data to its warehouse directory. It does not even check whether the external location exists at the time it is defined.

CHAPTER 6: PARTITIONING AND BUCKETING IN HIVE

[1] By default, a simple HiveQL query scans the whole table. This slows down the performance when querying a big table.

So in real time, we would have GBs, TBs, even petabytes of data. For example, let's suppose we have a file of 10 GB, which contains data about some departments in a company. And without the concept of partitioning used, our file will be in a directory like this: *"home/user/department_table".*

The whole 10 GB file with all the accounts in a single directory. Now on this directory, when you fire a search query to select only data of accounts department, Hive has to scan the full file and retrieve only those rows where department_column is equals to accounts. So in this case, we have made Hive unnecessarily scan other departments when it has to fetch only accounts department. So why not we, at pre-hand create a separate directory for accounts department, and save only accounts department data in it.

Hive tables can be broken in further logical chunks for ease of management and improving performance. There are few ways we can further abstract data in Hive. [2]:

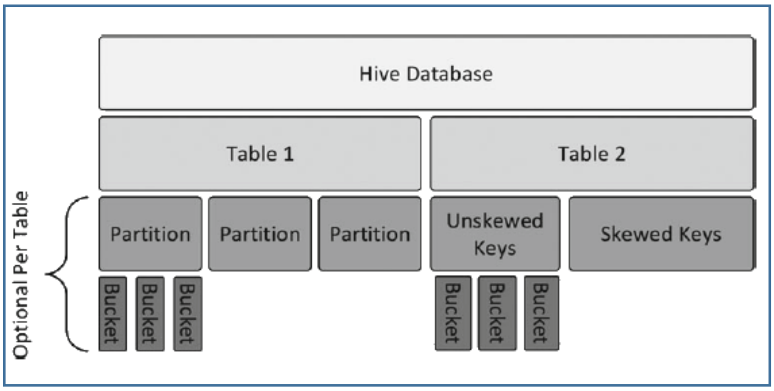


Figure 6-1

PARTITIONING

Partitioning is often used in the relational database world to enhance performance and for better management of the data. The concept of partitioning in Hive is no different.

[6] Apache Hive organizes tables into partitions for grouping same type of data together based on a column or partition key. In Hive, each partition corresponds to predefined partition column(s), which maps to subdirectories in the table's directory in HDFS. When the table gets queried, only the required partitions (directory) of data in the table are being read, so the I/O and time of the query is greatly reduced. Using partition is a very easy and effective way to improve performance in Hive. [13] Partitions are like horizontal slices of data that allows the large sets of data as more manageable chunks.

Partitioning in Hive is done using the PARTITIONED BY clause in the create table statement of table. Table can have one or more partitions. A table can be partitioned on the basis of one or more columns. The columns on which partitioning is done cannot be included in the data of table. For example, you have the four fields id, name, age and city and you want to partition the data on the basis of the city field, then the city field will not be included in the columns of create table statement and will only be used in the PARTITIONED BY clause. You can still query the data in a normal way using: where city='Larissa'. The result will be retrieved from the respective partition because data is stored in a different directory with the city name for each city

CREATE EXTERNAL TABLE IF NOT EXISTS employees (

id INT, name STRING, age INT)

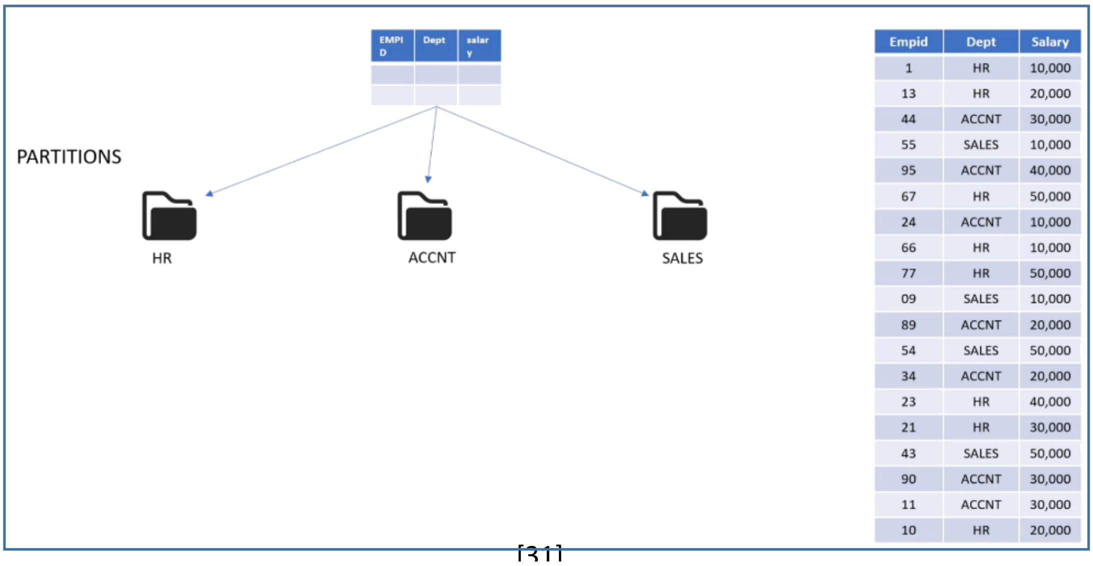
PARTITIONED BY (city STRING)

BUCKETING

Partitioning is used to increase the performance of queries, but the partitioning technique is efficient only if there is a limited number of partitions. To overcome the problem of partitioning, Hive provides the concept of bucketing.

Bucketing is another data organizing technique in Hive. It is a technique for decomposing larger datasets into more manageable ports. The basic fundamental of bucketing is that, when a table is bucketed on a column, then all the records with same column value will go to same bucket. We can use bucketing directly on a table, but it gives us best performance result when we do bucketing and partitioning side by side. We can assume it as, first we will create a partition, and inside the partition the data will be stored in buckets.

Let's understand it by an example. We have a table of employee salary of a company X. My table is containing three columns, employee ID, employee department, and employee salary. First of all, to get a good performance and a better organization of data, what we can do is, we can create partitions based on department. So each department is a partition now.

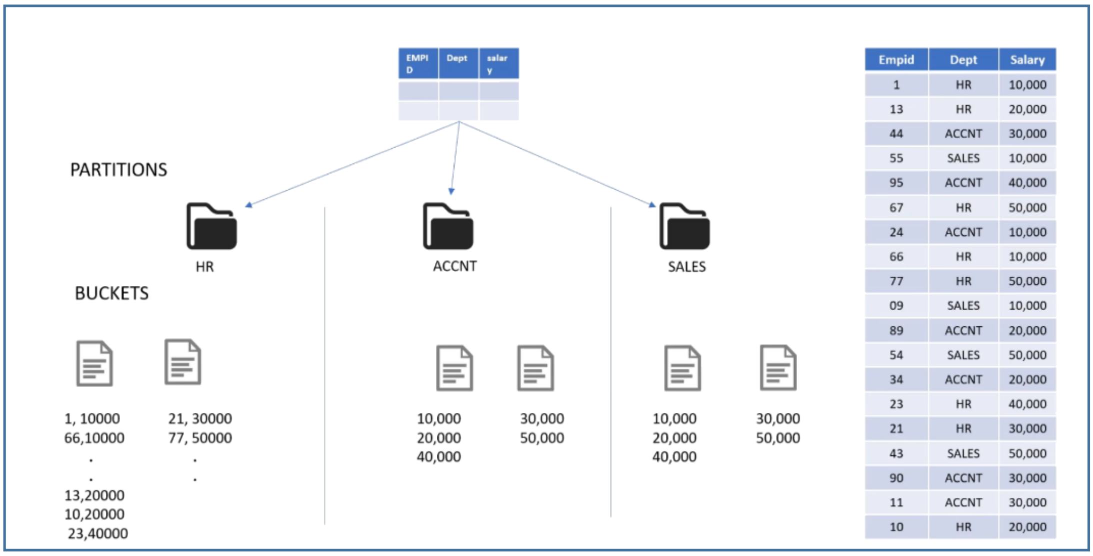


Empid	Dept	Salary
1	HR	10,000
13	HR	20,000
44	ACCNT	30,000
55	SALES	10,000
95	ACCNT	40,000
67	HR	50,000
24	ACCNT	10,000
66	HR	10,000
77	HR	50,000
09	SALES	10,000
89	ACCNT	20,000
54	SALES	50,000
34	ACCNT	20,000
23	HR	40,000
21	HR	30,000
43	SALES	50,000
90	ACCNT	30,000
11	ACCNT	30,000
10	HR	20,000

Figure 6-2

But what if still a partition has so many records on it? What if even after creating a partition, the latency is high? To achieve better organization of data, we'll do bucketing in this table. So within a partition, if I'm creating buckets to break down the data, I'll be achieving the best optimization.

Now, if I'm doing bucketing on salary column and creating two buckets, then inside each partition, all the employees with 10,000 salary will be present in a single bucket, and nowhere else. Please don't get confused that there will be different buckets for different salaries. There can be any number of salaries in a single bucket, but those salary records will be present in that bucket only. So 10,000, 20,000, and 40,000 salaried employees are in this bucket. 30,00 and 50,000 salaried employees are in second bucket, and same for every partition.



Empid	Dept	Salary
1	HR	10,000
13	HR	20,000
44	ACCNT	30,000
55	SALES	10,000
95	ACCNT	40,000
67	HR	50,000
24	ACCNT	10,000
66	HR	10,000
77	HR	50,000
09	SALES	10,000
89	ACCNT	20,000
54	SALES	50,000
34	ACCNT	20,000
23	HR	40,000
21	HR	30,000
43	SALES	50,000
90	ACCNT	30,000
11	ACCNT	30,000
10	HR	20,000

Figure 6-3

Please note that unlike partition, a bucket is physically a file, whereas partition is a directory. Internally, which salary will go to which bucket, is decided by a hashing algorithm.

CREATE EXTERNAL TABLE employee_salary (Empid INT, Salary INT) PARTITIONED BY (Dept String) CLUSTERED BY (Salary) INTO 2 BUCKETS;

CHAPTER 7: DIFFERENT TYPES OF FILES IN HIVE

Let’s create a table a little different this time:

```
CREATE TABLE data_types_table (
...
> ROW FORMAT DELIMITED
> FIELDS TERMINATED BY ','
> COLLECTION ITEMS TERMINATED BY '|'
> MAP KEYS TERMINATED BY '^'
> LINES TERMINATED BY '\n'
> STORED AS TEXTFILE;
```

The above lines define the Hive row format for your data_types_table and provide specifics on how fields will be separated or delimited whenever you insert or load data into the table. The last line defines the Hive file format — a text file — when the data is stored in the HDFS (or local file system). You may be wondering why our previous of creating a table lacks these extra keywords and delimiters. The reason is that Hive tables are created with the default configuration, unless you override the default settings.

The most important on the above code is the last line, “STORED AS TEXTFILE”. Although TEXTFILE is the default format for our Hive table records, it is not always the most suitable. Text files are slower to process, and they consume a lot of disk space unless you compress them. For these reasons and more, the Apache Hive community came up with several choices for storing our tables on the HDFS. The following list describes the file formats you can choose from as of Hive version 0.11 [1]:

TEXTFILE: The default file format for Hive records. Alphanumeric characters from the Unicode standard (see www.unicode.org) are used to store your data.

SEQUENCEFILE: The format for binary files composed of key/value pairs. Sequence files, which are used heavily by Hadoop, are often good choices for Hive table storage, especially if you want to integrate Hive with other technologies in the Hadoop ecosystem.

RCFILE: Stores records in a column-oriented fashion rather than a row-oriented fashion — like the TEXTFILE format approach. Using the RCFILE format makes sense when tables have a large number of columns, but only a few columns are typically accessed. RC files or record columnar files were the first columnar file format adopted in Hadoop. These are flat files consisting of binary key-value pairs, and it shares much similarity with sequence files. RC files are good and known for faster reads, but writing an RC file requires more memory and computation than non-column file formats. So we can summarize that RC file gives good read performance, but with a little compromise on write performance. RC file gives us significant compression. If data is suitable, then RC files can even be compressed with high compression ratio. There is no file format which provides all the features, and for RC file, it misses the schema evolution support. In order to add a column to our data, we must be rewrite every preexisting RC file.

ORC: A format that has significant optimizations to improve Hive reads and writes and the processing of tables (ORC stands for optimized row columnar.) For example, ORC files include optimizations for Hive complex types and new types such as DECIMAL. Also lightweight indexes are included with ORC files to improve performance. For a complete list of new ORC file format features, consult the Hive Language Manual at *https://cwiki.apache.org/confluence/display/Hive/ LanguageManual+ORC*

AVRO: Avro actually is not really a file format, it's a file format plus a serialization and de-serialization framework. Avro uses JSON for defining data types, and serializes data in a compact binary format. Talking about its read-write performance, avro files comes in, you can say middle, in terms of read and write performance, neither very fast nor very slow. So if your application is mainly dealing with IO operations, then Avro file is not the best choice you will go with. Avro supports block level compression and they are also split table. And most importantly, avro files are designed to support full schema evolution. Avro files store metadata with the data, and thus supports full scheme evolution. This file format is actually the best choice if you know that your file schema is going to be changed frequently.

PARQUET: Last comes the parquet file format. This is the most famous columnar file format adopted by the Hadoop community. Parquet stores nested data structures in a flat columnar format. Like RC and ORC, parquet is also designed for faster reads with less concern of write speeds. Parquet files can be compressed with the famous snappy compression codec. They are conditionally split table for some compression codecs. However, unlike RC and ORC files, parquet serDe support limited schema evolution. In parquet, new columns can be added at the end of structure.

CHAPTER 8: HIVE QL: BASIC COMMANDS AND FUNCTIONS

HIVE DATA MANIPULATION LANGUAGE (DML)

[7] The ability to manipulate data is critical in big data analysis. Manipulating data is the process of exchanging, moving, sorting transforming and modifying data. This technique is used in many situations, such as cleaning data, searching patterns, creating trends and so on. Hive QL offers commands which are used for inserting, retrieving, modifying, deleting, and updating data in the Hive table once the table and database schema has been defined using Hive Data Manipulation Language (DML) commands.

The various Hive DML commands are:

- LOAD: Loading data into a Hive table is one of the variants of inserting data into a Hive table. In this method, the entire file is copied/moved to a directory corresponds to Hive tables. If the table is partitioned, then data is loaded into partitions one at a time.

  The Hive syntax is as follows:

  LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename

LOAD DATA	Keywords for loading data in Hive.
LOCAL	If included, enables the users to load data from their local files. If omitted, the files are loaded from the path set in the Hadoop configuration variable fs.default.name.
INPATH 'filepath'	If LOCAL is used: file:///user/hive/example If LOCAL is omitted: hdfs://namenode:9000/user/hive/example
OVERWRITE	If included, enables the users to load data into an already populated table and replace the previous data. If omitted, enables the users to load data into an already populated table and append the new data to previous data.
INTO TABLE tablename	The tablename is the name of a table that exists in Hive

- SELECT: The SELECT statement in Hive is similar to the SELECT statement in SQL used for retrieving data from the database.

- INSERT: The Insert statement is used to append the data into a Hive table or partition. It keeps the existing data as it is and adds new data into one or more new data files.

- DELETE: Deleting data from a Hive table is the traditional way of deleting data in a table in any RDBMs. Deleting data in a table can only be performed if the table supports ACID properties.

- UPDATE: Updating data in a Hive table is the traditional way of updating data in a table in any RDBMs. Updating data in a table can only be performed if the table supports ACID properties.

- EXPORT: The Hive EXPORT statement exports the table or partition data along with the metadata to the specified output location in the HDFS.

- IMPORT: The Hive IMPORT command imports the data from a specified location to a new table or already existing table.

[7] Another aspect of manipulating data is properly sorting it in order to clearly identify important facts, such as top the N values, maximum, minimum and so on. Hive QL supports the following keywords for data sorting:

- ORDER BY [ASC | DESC]: It is similar to the SQL ORDER BY statement.

- SORT BY [ASC | DESC]: It specifies which columns to use to sort reducer input records. This means the sorting is completed before sending data to the reducer

- DISTRIBUTE BY: When the mapper decides to which reducer it can deliver the output.

- CLUSTER BY: It is a shortcut operator we can use to perform DISTRIBUTE BY and SORT BY operations on the same group of columns. The CLUSTER BY statement does not allow us to specify ASC or DESC yet.

We have describe Hive data types and Data Model, Hive DDL and Hive DML, but there are so many HiveQL features for querying and analyzing data, which are beyond the scope of this essay.

For an exhaustive list of HiveQL features, consult the Hive Language Manual at this page:

https://cwiki.apache.org/confluence/display/Hive/LanguageManual

HIVE FUNCTIONS

[7] To further manipulate data, we can also use operators, expressions, and functions in HQL to transform data.

The Hive wiki (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF) offers specifications for all supported expressions and functions, so we will mention only the most useful and common among them.

Functions in HQL are categorized as follows:

Mathematical functions: They are mainly used to perform mathematical calculations, such as rand(...) and pi(...).

Collection functions: They are used to find the size, keys, and values for complex types, such as size(...).

Type conversion functions: These are mainly cast(...) and binary(...) functions to convert one type to another.

Date functions: They are used to perform date-related calculations, such as year(...) and month(...).

Conditional functions: They are used to check specific conditions with a defined value returned, such as coalesce(...), if(...), and case when then else end

String functions: They are used to perform string-related operations, such as upper(...) and trim(...)

Aggregate functions: They are used to perform aggregation (introduced in the next chapter), such as sum(...) and count(*)

Table-generating functions: These functions transform a single input row into multiple output rows, such as explode(...) and json_tuple(...)

Customized functions: These functions are created by Java as extensions and will cover them in next chapter.

To list all operators, built-in functions, and user-defined functions, we can use the SHOW FUNCTIONS commands. For more details of a specific function, we can use DESC [EXTENDED] function name as follows:

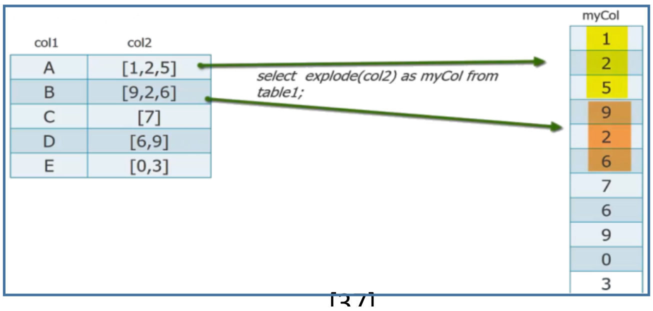
SHOW FUNCTIONS; -- List all functions

DESCRIBE FUNCTION <function_name>; -- Detail for the function

DESCRIBE FUNCTION EXTENDED <function_name>; -- More details

Explode and Lateral View functions:

Explode function takes an array as an input and results the elements of that array as separate rows. For example, we have the table in Figure 6-4 with two columns. Column_1 is of string datatype and column_2 is of array datatype. Now if we apply explode function on second column column_2 in select statement, then the output would look like Figure 6-4. Explode function has dispersed all the elements of array in individual rows.



col1	col2
A	[1,2,5]
B	[9,2,6]
C	[7]
D	[6,9]
E	[0,3]

select explode(col2) as myCol from table1;

myCol
1
2
5
9
2
6
7
6
9
0
3

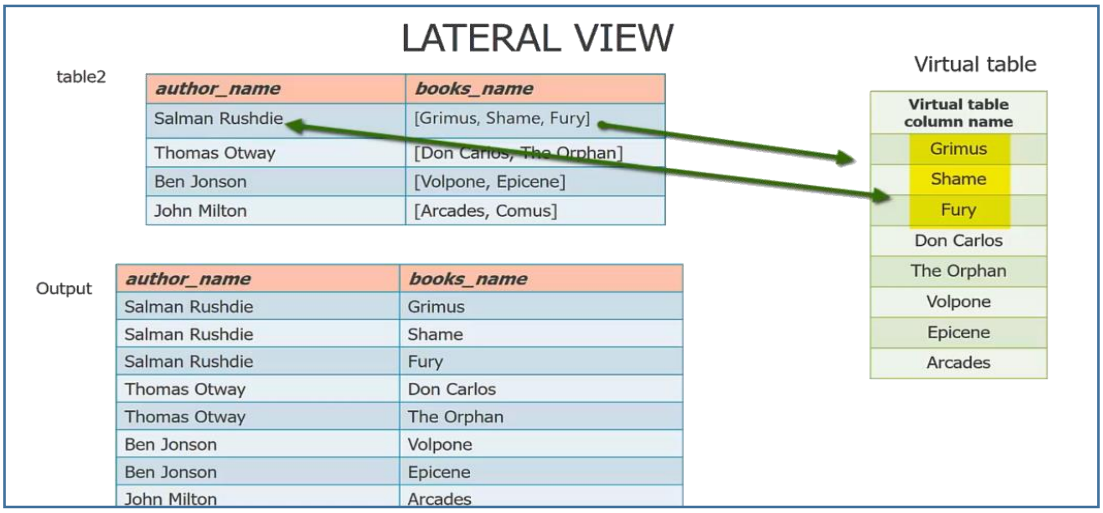
Figure 6-4

The limitation of explode function is that we can select only the column to be exploded in our select statement, we cannot select other columns of table along with the exploded column. For example, in this above case, we can use only column_2 in the select statement, and we cannot select both, column_1 and column_2. If we try to use both columns in the select statement, it would give us an error.

To overcome this limitation, lateral view function was introduced. With the help of lateral view, we can select any number of columns from our table along with the exploded column.

Suppose, we have the table shown in Figure 6-5 with two columns; author name as string datatype and book's name as array datatype. Book's name contains all the books an author has written. Now from this table, we want to select as each book name should come along with the author name.

This is how we want the output.



author_name	books_name
Salman Rushdie	[Grimus, Shame, Fury]
Thomas Otway	[Don Carlos, The Orphan]
Ben Jonson	[Volpone, Epicene]
John Milton	[Arcades, Comus]

Virtual table column name
Grimus
Shame
Fury
Don Carlos
The Orphan
Volpone
Epicene
Arcades

author_name	books_name
Salman Rushdie	Grimus
Salman Rushdie	Shame
Salman Rushdie	Fury
Thomas Otway	Don Carlos
Thomas Otway	The Orphan
Ben Jonson	Volpone
Ben Jonson	Epicene
John Milton	Arcades

Figure 6-5

Every book name has its corresponding author name. To perform this, we'll use lateral view function, because explode function alone can't do this job.

Lateral view will create a virtual table for the exploded column. Means, the output of the exploded column is stored temporarily in a virtual table, and then that virtual table is joined with our base table to get the desired output. If we see in this example, book's name column was exploded and stored in a virtual table, and then joined with author name column of base table to get the desired output.

Please note that we need not to worry about this virtual table and joining, as Hive does it internally by itself.

The code:

```
select author_name, dummy_booksname form table2 lateral view
explode(books_name) dummy as dummy_booksname;
```

CHAPTER 9: HIVE QL: VIEWS AND JOINS

VIEWS

Views are logical data structures that can be used to simplify queries by hiding the complexities, such as joins, subqueries, and filters. It is called logical because views are only defined in metastore without the footprint in HDFS. Unlike what's in the relational database, views in HQL do not store data or get materialized. In other words, *materialized views* are not currently supported by Hive.

When a query references a view, the information in its definition is combined with the rest of the query by Hive’s query planner. Logically, you can imagine that Hive executes the view and then uses the results in the rest of the query.

Basically in Hive, view is kind of a searchable object in a database which itself does not have any data but can be populated by the results of a query. Here are some few properties of view.

First as we already mentioned, hive views do not contain any data of its own, but they are merely the results of any hive query on a table.

Second is that, all type of DML operations can be performed on a view, same like table's DML operations. Views can be created by selecting any number of rows and columns from its base table or tables, because views can also reflect the results of join of N number of tables.

Once created, views become independent of it's base table schema. It means, after we created a view on a table with a schema, that schema of view gets frozen and it won't change if we change its base table schema. If we have to change view schema, we have to use alter views command. Also, vice versa is true.

Changing the view schema won't change our table schema. Views are read-only. Any drop or write queries on views will not affect our base table. Next, dropping the base table will make the view on that table ineffective. This makes sense because view does not have its own data and it is reflecting the queried data of its base table only.

[4] When a query becomes long or complicated, a view may be used to hide the complexity by dividing the query into smaller, more manageable pieces; similar to writing a function in a programming language or the concept of layered design in software. Encapsulating the complexity makes it easier for end users to construct complex queries from reusable parts. For example, consider the following query with a nested subquery:

```
FROM (
        SELECT * FROM people JOIN cart
                ON (cart.people_id=people.id) WHERE firstname='john'
) a SELECT a.lastname WHERE a.id=3;
```

It is common for Hive queries to have many levels of nesting. In the following example, the nested portion of the query is turned into a view:

```
CREATE VIEW shorter_join AS

SELECT * FROM people JOIN cart

ON (cart.people_id=people.id) WHERE firstname='john';
```

JOINS

[4][7][13] A join in Hive is used for the same purpose as in a traditional RDBMS. A join is used to fetch meaningful data from two or more tables based on a common value or field. In other words, a join is used to combine data from multiple tables. A join is performed whenever multiple tables are specified inside the FROM clause.

Hive supports most SQL JOIN operations, such as INNER JOIN and OUTER JOIN. In addition, HQL supports some special joins, such as MapJoin and Semi-Join too. In its earlier version, Hive only supported equal join. After v2.2.0, unequal join is also supported. However, you should be more careful when using unequal join unless you know what is expected, since unequal join is likely to return many rows by producing a Cartesian product of joined tables. When you want to restrict the output of a join, you should apply a WHERE clause after join as JOIN occurs before the WHERE clause.

INNER JOIN or JOIN returns rows meeting the join conditions from both sides of joined tables. The JOIN keyword can also be omitted by comma-separated table names; this is called an implicit join.

Besides INNER JOIN, HiveQL also supports regular OUTER JOIN and FULL JOIN. The logic of such a join is the same as what's in the SQL. The following table summarizes the differences between common joins. Here, we assume table_m has m rows and table_n has n rows with one-to-one mapping:

Join type	**Logic**	**Rows returned**
table_m JOIN table_n	This returns all rows matched in both tables.	m ∩ n
table_m LEFT JOIN table_n	This returns all rows in the left table and matched rows in the right table. If there is no match in the right table, it returns NULL in the right table.	m
table_m RIGHT JOIN table_n	This returns all rows in the right table and matched rows in the left table. If there is no match in the left table, it returns NULL in	n

	the left table.	
table_m FULL JOIN table_n	This returns all rows in both tables and matched rows in both tables. If there is no match in the left or right table, it returns NULL instead.	$m + n - m \cap n$
table_m CROSS JOIN table_n	This returns all row combinations in both the tables to produce a Cartesian product.	$m * n$

HiveQL also supports some special joins that we usually do not see in relational databases, such as MapJoin and Semi-join. MapJoin means doing the join operation only with map, without the reduce job. The MapJoin statement reads all the data from the small table to memory and broadcasts to all maps. During the map phase, the join operation is performed by comparing each row of data in the big table with small tables against the join conditions. Because there is no reduce needed, such kinds of join usually have better performance. In the newer version of Hive, Hive automatically converts join to MapJoin at runtime if possible.

However, you can also manually specify the broadcast table by providing a join hint, /*+ MAPJOIN(table_name) */. In addition, MapJoin can be used for unequal joins to improve performance since both MapJoin and WHERE are performed in the map phase.

CHAPTER 10: USER DEFINED FUNCTIONS

UDF means User Defined Functions. So what are these UDFs? Hive supports may different types of functions, as we have already mentioned in previous chapter, but in real-time projects, these functions are still not sufficient. For example, Hive provides a built-in function to convert the uppercase to lowercase, and lowercase to uppercase.

But if a requirement comes that we have to change the case of only first alphabet in a word i.e we have to make only the first alphabet to uppercase in a word, then we don't have a built-in function for this situation.

Hive provides a utility UDF to handle this situation. We can define our own functions i.e. user-defined functions, according to the requirement.

These functions will be written in Java, and there are some rules to be followed while writing the UDFs..

Here is the flow that we follow to run a UDF in Hive.

Step-1: is to create a java program in any platform (Eclipse, NetBeans),

Step-2: save or convert the program into a jar file.

Step-3: adding that jar file into our Hive shell.

Step-4: create a function from that jar file which we have added.

Step-5: use those functions in Hive query according to requirement.

Developing UDFs in Hive is by no means rocket science, and is an effective way of solving problems that could either be downright impossible, or very akward to solve (for example by using complex SQL constructs, multiple nested queries or intermediate tables). Although the lack of documentation and resources to help in this process it will make it sometimes quite painful.

A UDF processes one or several columns of one row and outputs one value. For example:

SELECT lower(str) from table

For each row in "table," the "lower" UDF takes one argument, the value of "str", and outputs one value, the lowercase representation of "str".

Each argument of a UDF can be:

- A column of the table
- A constant value
- The result of another UDF
- The result of an arithmetic computation

[19][20] First, you need to create a new class that extends UDF, with one or more methods named evaluate.

```
package com.example.hive.udf;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;
public final class Lower extends UDF {
  public Text evaluate(final Text s) {
    if (s == null) { return null; }
    return new Text(s.toString().toLowerCase());
  }
}
```

After compiling your code to a jar, you need to add this to the Hive classpath.

```
hive> addjar my-udf.jar
hive>create temporary function my_lower as 'com.example.hive.udf.Lower';
hive>hive> select my_lower(title), sum(freq) from titles group by my_lower(title);
```

UDF can accept a large variety of types to represent the column types. Notably, it accepts both Java primitive types and Hadoop IO types. Depending on the use cases the UDFs can be written, it will accept and produce different numbers of input and output values.

The general type of UDF will accept single input value and produce a single output value. If the UDF used in the query, then UDF will be called once for each row in the result data set.

In the other way, it can accept a group of values as input and return single output value as well.

CHAPTER 11: QUERYING SEMI-STRUCTURED DATA

[9] Hive would not be much of a useful data warehouse tool without the ability to query data. Luckily, querying and providing schema-on-read capabilities at scale is the core foundation for Hive use cases. The power Hive provides is the ability to translate a large variety of data formats as well as the ability to customize translations to fit your unique business needs. Hive adapts to your data formats instead of the other way around. This is the core foundation for a data-driven organization.

The Hadoop noise machine was fond of referring to data as structured, semi-structured, or non- structured. Structured data always referred to data represented in rows and columns. This is what was most familiar to data analysts, especially professionals working with traditional transactional systems like point- of-sales or inventory management. Semi-structured data refers to a gray line between columns and rows and maybe something more exotic like key-value pairs, arrays, or nested data. Maybe the number of columns in the data structure was dynamic, or maybe there were multiple values in a single column. This data felt like traditional data but its representation was much different. Examples of this data include XML, HL7, and JSON.

Semi-structured data could also be associated with syslog or application event log files. Finally, there was unstructured data in the form of images, OCR, PDF, or spatial data. Unstructured data was complex data where potentially the structure was not in columns, rows, or arrays, but was in the byte patterns in an image of a cat on the Internet or a rib cage in a X-ray. The truth of the matter is no data is patternless. What matters is the algorithm used to detect the pattern. Granted, the pattern may change during moments of ingest or may not be readily or easily detectable, but all data still has a pattern and it is up to developers to glean those patterns using all the tools at their disposal, and it is up to the tools analyzing the data to have the flexibility to accommodate the potential range of patterns.

INGESTING AND QUERING XML DATA

XML is a semi-structured format, it contains various tags and inside those tags, the actual value is mentioned. So at last, those are the values which we have to insert in Hive table, and rest tags should be omitted.

Now the main point is how to load these semi-structured XML files into Hive.

Hive provides us various SerDes to read different types of data, and even it allows us to write our own SerDes if we have our own formatted data. SerDe is short notation for serializer\deserializer and is a means for Hive to read data from a table and write it out in any customizable format. Developers write SerDes so that Hive can interpret varying file formats.

But for XML files, we don't have to create our own SerDe because the SerDe for this is already available on the internet. The only thing we have to do is to download this

SerDe from the internet and add it to our Hive environment. So let's work with an XML file to see in practice how to load and query XML data with Hive.

We will use the following XML file "books.xml". As we can see, there are lots of tags in this XML file, and corresponding values for these tags resides in them.

Actually the structure of XM: is like that only, for each individual tag, there is a value associated with it.

```
<CATALOG>

<BOOK>

<TITLE>Hadoop Defnitive Guide</TITLE>

<AUTHOR>Tom White</AUTHOR>

<COUNTRY>US</COUNTRY>

<COMPANY>CLOUDERA</COMPANY>

<PRICE>24.90</PRICE>

<YEAR>2012</YEAR>

</BOOK>

<BOOK>

<TITLE>Programming Pig</TITLE>

<AUTHOR>Alan Gates</AUTHOR>

<COUNTRY>USA</COUNTRY>

<COMPANY>Horton Works</COMPANY>

<PRICE>30.90</PRICE>

<YEAR>2013</YEAR>

</BOOK>

</CATALOG>
```

From this XML, we have to load these values into our Hive table and we have to omit these tags. We can think of this XML file as, book will be our table, and the title, author, countries, will act as columns into our Hive table, and these columns will have different rows containing different values for different books.

We will do by using a specific SerDe, and the SerDe name is:

"com.IBM.SPSS.hive.SerDe2".

And inside these SerDes, we have this input format and output format specified.

So first of all, we will download this SerDe from internet and add it into my Hive environment, using the command add jar:

```
hive> ADD JAR /home/myjio/Downloads/hivexmlserde-1.0.0.0.jar
```

And the code for the query in which we create our “XML” table:

```
CREATE TABLE book_details(TITLE STRING, AUTHOR STRING,COUNTRY STRING,COMPANY STRING,PRICE FLOAT,YEAR INT)
ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
WITH SERDEPROPERTIES (
"column.xpath.TITLE"="/BOOK/TITLE/text()",
"column.xpath.AUTHOR"="/BOOK/AUTHOR/text()",
"column.xpath.COUNTRY"="/BOOK/COUNTRY/text()",
"column.xpath.COMPANY"="/BOOK/COMPANY/text()",
"column.xpath.PRICE"="/BOOK/PRICE/text()",
"column.xpath.YEAR"="/BOOK/YEAR/text()")
STORED AS
INPUTFORMAT 'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat'
TBLPROPERTIES ("xmlinput.start"="<BOOK","xmlinput.end"= "</BOOK>");
```

In the first line, we are telling Hive to take the value of title from the path */BOOK/TITLE/text()* as we can see in the XML file; under book, there is a tag "title", and inside the title, whatever the text it is, it should take this text as a value. So same goes for author, country, company, price and year.

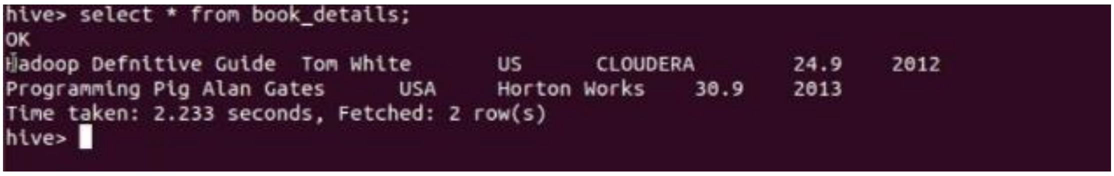
Then next is "stored as". Now here we have to explicitly mention input format and output format, because here we are not using that default input and output formats, we have to mention the input and output formats from our SerDe.

And at last, we have to mention table properties. In table properties, we'll mention the start and the end tag for our whole file, which is "xmlinput.start"=<BOOK”, and the end tag, "xmlinput.end"= "</BOOK>".

Now we have to load the data into our “XML” table using the LOAD statement:

hive>load data local inpath ‘home/myjio/files/books.xml’ into table book_details’;

hive>select * from book_details;



```
hive> select * from book_details;
OK
Hadoop Defnitive Guide  Tom White       US      CLOUDERA        24.9    2012
Programming Pig Alan Gates      USA     Horton Works    30.9    2013
Time taken: 2.233 seconds, Fetched: 2 row(s)
hive> 
```

Figure 11-1

CHAPTER 12: SECURITY

In today's era of big data, most of the organizations are concentrating to use Hadoop as a centralized data store. Data size is growing day by day, and organizations want to derive some insights and make decisions using the important information. While everyone is focusing on collecting the data, but having all the data at a centralized place increases the risk of data security. Securing the data access of HDFS is very important. For this reason, security in Hive is always considered an integral and important part of the ecosystem. The earlier version of Hive mainly relied on HDFS for security. The security of Hive gradually became mature after hiveserver2 was released. Furthermore, when we talk about security, there are two major things – Authentication and Authorization.

Hive v0.7.0 added integration with Hadoop security, meaning, for example, that when Hive sends MapReduce jobs to the JobTracker in a secure cluster, it will use the proper authentication procedures. User privileges can be granted and revoked, as we'll discuss below.

Authentication

[7] Authentication is the process of verifying the identity of a user by obtaining the user's credentials. Hive has offered authentication since hiveserver2. In the old version of Hive, hiveserver1 does not support Kerberos[3] authentication for thrift clients. As result, if we could access the host/port over the network, we could access the server. Instead, we can leverage the metastore server, which supports Kerberos, for authentication. [8] Kerberos allows for mutual authentication between client and server. In this system, the client's request for a ticket is passed along with the request.

[4] But Hive was created before any of this Kerberos support was added to Hadoop, and Hive is not yet fully compliant with the Hadoop security changes. For example, the connection to the Hive metastore may use a direct connection to a JDBC database or it may go through Thrift, which will have to take actions on behalf of the user.

Authorization

[7] Authorization is used to verify whether a user has permission to perform a certain action, such as creating, reading or writing data or metadata. Hive provides three authorization modes: legacy, store-based and SQL standard-based mode.

Legacy Mode

This is the default authorization mode in Hive, providing column-and-row-level authorization through HiveQL statements. However, it is not a completely secure authorization mode and has a couple of limitations. It can be mainly used to prevent good users from accidentally doing bad things rather than preventing malicious user operations.

Storage-based mode

[13] HDFS supports a permission model for files and directories that is much equivalent to the standard POSIX model. Similar to UNIX permissions, each file and directory in HDFS is associated with an owner, group and another users. There are three types of permissions in HDFS –read, write and execute.

Although this basic permission model is sufficient to handle a large number of security requirements at a bloc level, but using this model we cannot define finer level security named users or groups.

[13] HDFS also has a feature to configure an Access Control List (ACL), which can be used to define fine-grained permissions at the file level as well as the directory level for specific named users or groups.

[7] Considering its implementation, the storage-based authorization mode only offers authorization at the level of databases, tables and partitions rather than column-and-row level. With dependency on the HDFS permissions, it lacks the flexibility to manage authorization through HiveQL statements.

SQL standard-based mode

[15] For fine-grained access control on a column and row level, we can use SQL standard-based mode, available since Hive v0.13.0. It is similar to relational database authorization by using the GRANT and REVOKE statements to control access through the hiveserver2 configuration. However, tools such as Hive or HDFS commands do not access data through hiveserver2, so SQL standard-based mode cannot authorize their access.

Therefore, it is recommended we use storage-based mode together with SQL standard-based mode to authorize users connecting from various tools.

[3] Kerberos is a network authentication protocol developed by MIT as part of Project Athena. It uses time-sensitive tickets that are generated using symmetric key cryptology to securely authenticate a user in an unsecured network environment.

CHAPTER 13: FUTURE OF HIVE

ENHANCEMENTS

When Hive was first introduced over 10 years ago, the motivation of the authors was to expose a SQL-like interface (on top of Hadoop Map/Reduce) to users and translate data manipulation statements (queries) to a directed acyclic graph (DAG) of Map/Reduce jobs. With an abstract SQL interface, the users did not have to deal with low level implementation details for their parallel batch processing jobs. Hive focused mainly on Extract-Transform-Load (ETL) or batch reporting workloads that consisted of (i) reading huge amounts of data, (ii) executing transformations over that data (e.g., data wrangling, consolidation, aggregation) and finally (iii) loading the output into other systems that were used for further analysis.

As Hadoop became a ubiquitous platform for inexpensive data storage with HDFS, developers focused on increasing the range of workloads that could be executed efficiently within the platform. YARN, a resource management framework for Hadoop, was introduced, and shortly afterwards, data processing engines (other than MapReduce) such as Spark were enabled to run on Hadoop directly by supporting YARN.

Users also increasingly focused on migrating their data warehousing workloads from other systems to Hadoop. These workloads included interactive and ad-hoc reporting, dash- boarding and other business intelligence use cases. There was a common requirement that made these workloads a challenge on Hadoop: They required a low-latency SQL engine.

Instead of implementing a new system, the Hive community concluded that the current implementation of the project provided a good foundation to support these workloads. Hive had been designed for large-scale reliable computation in Hadoop, and it already provided SQL compatibility (alas, limited) and connectivity to other data management systems. However, Hive needed to evolve and undergo major renovation to satisfy the requirements of these new use cases, adopting common data warehousing techniques that had been extensively studied over the years.

In particular the roadmap of enhancements and improvements focus on the following topics:

- ACID Support, MERGE and Locks
- LLAP (Live Long & Process)
- Hive on Spark

ACID Support, MERGE and Locks

[21][22] ACID transactions are critical requirements in enterprise data warehouses. In this section, we describe the improvements made to Hive in order to provide ACID guarantees on top of Hadoop.

Initially, Hive only had support to insert and drop full partitions from a table. Although the lack of row level operations was acceptable for ETL workloads, as Hive evolved to support many traditional data warehousing workloads, there was an increasing requirement for full DML support and ACID transactions. Hence, Hive now includes support to execute INSERT, UPDATE, DELETE, and MERGE statements. It provides ACID guarantees via Snapshot Isolation on read and well defined semantics in case of failure using a transaction manager built on top of the HMS. Currently transactions can only span a single statement; we plan to support multi-statement transactions in the near future. However, it is possible to write to multiple tables within a single transaction using Hive multi-insert statements.

[7] For now, all transactions in HQL are auto-committed without supporting BEGIN, COMMIT, and ROLLBACK, like as with relational databases. Also, the table that has a transaction feature enabled has to be a bucket table with the ORC file format.

[7] Locks ensure data isolation as described in the ACID principle. Hive has supported concurrency access and locking mechanisms since v0.7.0 and updated to a new lock manager in v0.13.0. There are two types of lock provided as follows:

- Shared lock: Also called S lock, it allows being shared concurrently. This is acquired when a table/partition is read.
- Exclusive lock: Also called X lock. This is acquired for all other operations that modify the table/partition.

For partition tables, only a shared lock is acquired if the change is only applicable to the newly-created partitions. An exclusive lock is acquired on the table if the change is applicable to all partitions. In addition, an exclusive lock on the table globally affects all partitions. For more information regarding locks, see:

https://cwiki.apache.org/ confluence/display/Hive/Locking

LLAP (Live Long & Process)

[2] The demand for sub-second queries calls for fast query execution and lower setup cost of tasks within the ecosystem. The challenge for Hive is to accomplish this without impacting the scale and flexibility that users require from a future distributed solution.

A future-proof methodology using a hybrid engine that leverages Tez and a new engine called LLAP (Live Long and Process) is the next phase for Hive. [21] LLAP is an optional layer that provides persistent multi-threaded query executors and multi-tenant in-memory cache to deliver faster SQL processing at large scale in Hive. LLAP does not replace the existing execution runtime used by Hive, such as Tez, but rather enhances it. In particular, execution is scheduled and monitored by Hive query coordinators transparently over both LLAP nodes as well as regular containers.

LLAP is an enhanced daemon process running on multiple nodes. YARN will be responsible for workload management in LLAP by means of delegation. Queries will transport information from YARN to LLAP about their authorized resource allocation.

LLAP processes will then distribute supplementary resources to assist the query as instructed by YARN.

The hybrid engine approach delivers fast response times by efficient in-memory data caching and low-latency processing, delivered by node resident processes. The effective limiting of LLAP usage during the initial phases of query processing means that Hive by-passes limitations around coordination, workload management, and failure isolation that are normally presented by running an entire query in this processing on the databases.

Hive on Spark

[2] Apache Spark is rapidly evolving into the programmatic successor to MapReduce for data processing on Apache Hadoop. The successful integration will open the enormous development that is done in the Spark ecosystem directly to Hive.

The biggest is the development in the deep-learning capability of spark. The evolving research into solutions using Spark and TensorFlow will deliver capacity to Hive solutions to use these investments via the Hive-on-Spark stack.

Machine learning has rapidly developed as a critical portion in mining Big Data for actionable insights.

Built on top of Spark, MLlib is a scalable machine-learning library that delivers high-quality algorithms.

FUTURE WORK

Additionally to fully support ACID capabilities, Hive plans to implement multi-statement transactions in the near future.

Materialized views work is still an ongoing project and one of the most requested features is the implementation of an advisor or recommender for Hive. Also improvements to LLAP performance and stability as well as the implementation of new connectors to other specialized systems, e.g., Kafka, are in progress too.

CHAPTER 14: HIVEMALL

With Apache Hive we have a powerful tool, for data preparation and preprocessor for our data, which reside in the distributed storage system of Hadoop ecosystem. But so far, we have not answered the big question, can we use SQL over Hive for Machine Learning?

The Hadoop ecosystem provides Apache Mahout (among others) to help develop our Machine Learning models. But Apache Mahout Core algorithms for clustering, classification and batch based collaborative filtering are implemented on top of Apache Hadoop using Map/Reduce paradigm. So, the main disadvantage for the analysts need to have a basic understanding of Hadoop and Map/Reduce programming is still lies here.

Why not to run Machine Learnings algorithms inside Hive? In that case we will have less components to manage and more scalable.

Apache Hivemall is a collection of Machine Learning algorithms and versatile data analytics functions. It provides a number of ease of use ML functionalities through the Apache Hive UDF/UDAF/UDTF interface.

Apache Hivemall offers a variety of functionalities: regression, classification, recommendation, anomaly detection, k-nearest neighbor, and feature engineering. It also supports state-of-the-art Machine Learning algorithms such as Soft Confidence Weighted, Adaptive Regularization of Weight Vectors, Factorization Machines, and AdaDelta.

HIVEMALL ARCHITECTURE

Apache Hivemall is mainly designed to run on Apache Hive but it also supports Apache Pig and Apache Spark for the runtime. Thus, it can be considered as a cross platform library for machine learning; prediction models built by a batch query of Apache Hive can be used on Apache Spark/Pig, and conversely, prediction models build by Apache Spark can be used from Apache Hive/Pig.

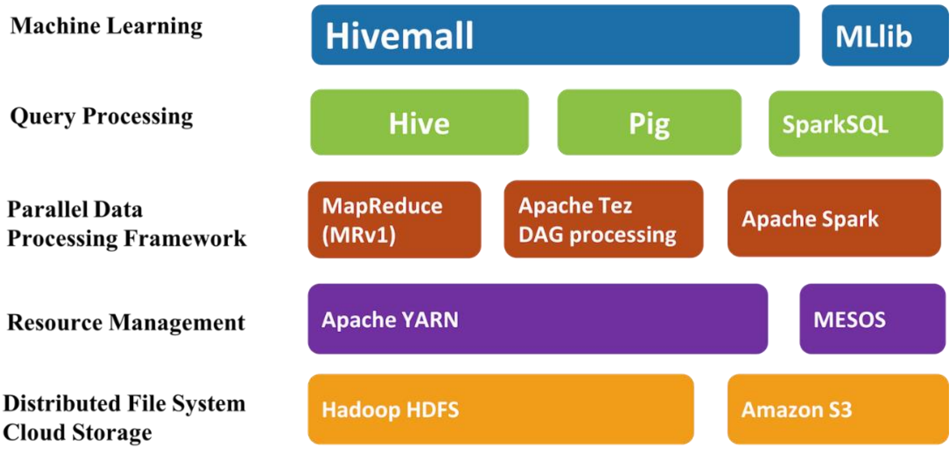


Figure 14-1

Hivemall is easy and scalable. Existing users of Hive can implement machine learning algorithms using the well-known Hive QL language. There is no need to compile programs and create executable jars as in MLlib or H2O. Just add UDFs or UDTFs and execute Hive queries. It also provides the scalability benefits of Hadoop and Hive with additional features to provide scalability to any number of training and testing instances and also any number of features.

For example, the following query automatically runs in parallel on Hadoop:

```
CREATE TABLE lr_model AS

SELECT feature, -- reducers perform model averaging in parallel

avg(weight) as weight

FROM ( SELECT logress(features,label,..) as (feature,weight)

FROM

train ) t -- map-only task

GROUP BY feature; -- shuffled to reducers
```

ALGORITHMS SUPPORTED BY HIVEMALL

Apache Hivemall provides machine learning functionality as well as feature engineering functions through UDFs/UDAFs/UDTFs of Hive.

Binary Classification

- Perceptron
- Passive Aggressive (PA, PA1, PA2)
- Confidence Weighted (CW)
- Adaptive Regularization of Weight Vectors (AROW)
- Soft Confidence Weighted (SCW1, SCW2)
- AdaGradRDA (w/ hinge loss)
- Factorization Machine (w/ logistic loss)

Recommended is AROW, SCW1, AdaGradRDA, and Factorization Machine while it depends.

Multi-class Classification

- Perceptron
- Passive Aggressive (PA, PA1, PA2)
- Confidence Weighted (CW)
- Adaptive Regularization of Weight Vectors (AROW)
- Soft Confidence Weighted (SCW1, SCW2)
- Random Forest Classifier

- Gradient Tree Boosting (Experimental)

Recommended is AROW and SCW while it depends.

Regression

- Logistic Regression using Stochastic Gradient Descent
- AdaGrad, AdaDelta (w/ logistic Loss)
- Passive Aggressive Regression (PA1, PA2)
- AROW regression
- Random Forest Regressor
- Factorization Machine (w/ squared loss)
- Polynomial Regression

Recommended is AROW regression, AdaDelta, and Factorization Machine while it depends.

Recommendation

- Minhash (LSH with jaccard index)
- Matrix Factorization (sgd, adagrad)
- Factorization Machine (squared loss for rating prediction)

k-Nearest Neighbor

- Minhash (LSH with jaccard index)
- b-Bit minhash
- Brute-force search using Cosine similarity

Anomaly Detection

- Local Outlier Factor (LOF)

Natural Language Processing

- English/Japanese Text Tokenizer

Feature engineering

- Feature Hashing (MurmurHash, SHA1)
- Feature scaling (Min-Max Normalization, Z-Score)
- Polynomial Features
- Feature instances amplifier that reduces iterations on training
- TF-IDF vectorizer
- Bias clause
- Data generator for one-vs-the-rest classifiers

System requirements

- Hive 0.13 or later, Java 7 or later
- Spark 2.1 or later for Apache Hivemall on Spark

CHAPTER 15: A CASE STUDY: BUILDING A MODEL FOR FOOTBALL PERFORMANCE WITH HIVE AND HIVEMALL

We now know that modern football and the performance of athletes are increasingly based on results and performance indicators, generating from data collection and analysis. Just ten years ago, the data used by football clubs were limited to basic statistics such as goal, shots, corners, possession, etc. Data that in the hands of the coaching staff were of little value, as they "captured" the final image of a match, without give the opportunity to process / analyze them to draw useful conclusions for the coaching staff. But with the ongoing development of technology (cloud, hd monitors, sensors) nowadays a team can literally collects GB of data during games and training from every aspect of the game. This data is not just statistical numbers, but on-the-ball event data. For example, Opta (there are now many companies, Statsbomb, Instats, Wyscout), now provides in a system of axes (x, y) the coordinates of each pass, defensive effort and final effort within the dimensions of the field.

The use of these on-the-ball event data/statistics, has now a very important role in the effort to develop/improve the skills of the players (especially the younger ones), but also in the in-depth analysis of a match for all areas of the game and in all its phases (attack, defense, counterattack). We can thus "discover" possible correlations that may lurk in the data and may indicate potential problems or areas for improvement. Also very important, a thorough analysis of a player's past performance or that of a team as a whole, can contribute to the creation of performance prediction models (individually/team) for the future (next match/season), such as which formation is best suited, against specific teams/formations or the percentage of chances of crosses from a specific side and/or time point to end up in a goal. Also with the thorough use of these on-the-ball event data/statistics, prediction models can be created for which players to buy fit the roster, based on the current formation and performance of the team. Finally, and in collaboration with the medical staff of the team, models could be created to predict the fatigue and serious injuries of the team players.

In this chapter we will try to build and demonstrate a simple binary regression model with Hive as our ETL tool and Hivemall as our Machine Learning library.

The scope is to build a model based on statistical (i.e pass, shot) and categorical attributes (position), which can predict the position of a new player based only on the statistical attributes. For our essay, we will try to build a model to predict for a given player, if his position is midfielder (we simplified the positions of a football team, to make simpler our paradigm).

For that purpose, we will use the public share soccer data available in this url:

https://figshare.com/authors/Luca_Pappalardo/6396764

and in particular the “PlayeRank” and “Events/Euro” datasets.

The first dataset contains among other, the Role of every player, which will be our target variable. This will be the only variable we will select from this dataset. The second dataset contains a variety of statistical attributes (the on-the-ball event data) for every player for every match he participated.

From that dataset we will choose the statistics/variables that will suit better our model and specifically: duel, foul, free kick, offside, challenge, pass, shot

STEP 1: Install Hivemall and its dependencies

We must download two files:

- hivemall-all-xxx.jar
- define-all.hive (of the hivemall version we download, e.g., v0.5.0)

Add the following two lines to your $HOME/.hiverc file.

```
add jar /home/myui/tmp/hivemall-all-xxx.jar;
source /home/myui/tmp/define-all.hive;
```

This automatically loads all Hivemall functions every time you start a Hive session. Alternatively, you can run the following command each time.

```
$ hive

add jar /tmp/hivemall-all-xxx.jar;

source /tmp/define-all.hive;
```

STEP 2: In Hive, prepare the tables of our model.

The datasets we have downloaded is in JSON format. Although Hive can query of course JSON data, we transformed them to a csv file with the help of Excel. Excel is a very useful tool for every data analyst for such transformations.

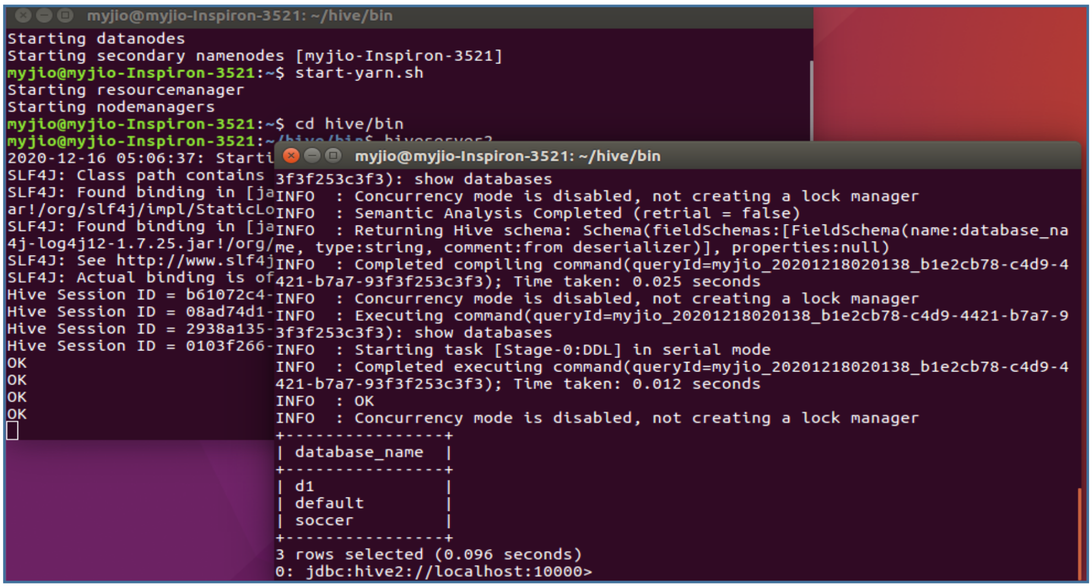
We saved our files as players_stats.csv, which contains the data for the training, and players_sample, which contains the data for the prediction phase.

After that, we start Hadoop deamons and Hive with beeline interface, via hiveserver2 and we are ready to build our model.

First we have to create a new database for our project and use it:

```
hive>create database soccer;

hive>use soccer;
```

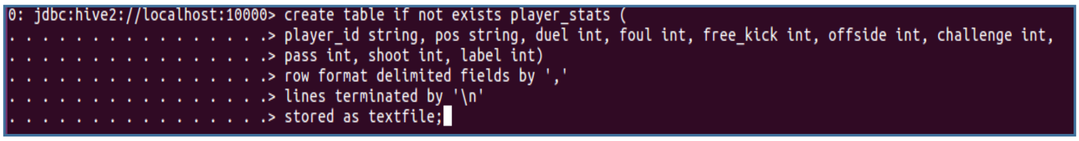


```
myjio@myjio-Inspiron-3521: ~/hive/bin
Starting datanodes
Starting secondary namenodes [myjio-Inspiron-3521]
myjio@myjio-Inspiron-3521:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
myjio@myjio-Inspiron-3521:~$ cd hive/bin
myjio@myjio-Inspiron-3521:~
2020-12-16 05:06:37: Starti
SLF4J: Class path contains
SLF4J: Found binding in [ja
ar!/org/slf4j/impl/StaticLo
SLF4J: Found binding in [ja
4j-log4j12-1.7.25.jar!/org/
SLF4J: See http://www.slf4j
SLF4J: Actual binding is of
Hive Session ID = b61072c4-
Hive Session ID = 08ad74d1-
Hive Session ID = 2938a135-
Hive Session ID = 0103f266-
OK
OK
OK
OK
```

```
myjio@myjio-Inspiron-3521: ~/hive/bin
3f3f253c3f3): show databases
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:[FieldSchema(name:database_na
me, type:string, comment:from deserializer)], properties:null)
INFO  : Completed compiling command(queryId=myjio_20201218020138_b1e2cb78-c4d9-4
421-b7a7-93f3f253c3f3); Time taken: 0.025 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=myjio_20201218020138_b1e2cb78-c4d9-4421-b7a7-9
3f3f253c3f3): show databases
INFO  : Starting task [Stage-0:DDL] in serial mode
INFO  : Completed executing command(queryId=myjio_20201218020138_b1e2cb78-c4d9-4
421-b7a7-93f3f253c3f3); Time taken: 0.012 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
+----------------+
| database_name  |
+----------------+
| d1             |
| default        |
| soccer         |
+----------------+
3 rows selected (0.096 seconds)
0: jdbc:hive2://localhost:10000>
```

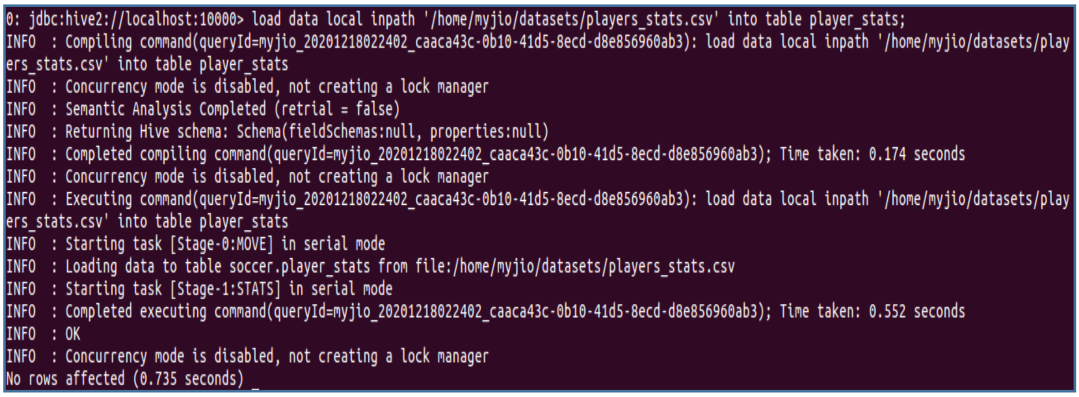
Image 15-1

Afterwards we have to create our tables in which we will load the data from the players_stats.csv file and player_samples.csv file:



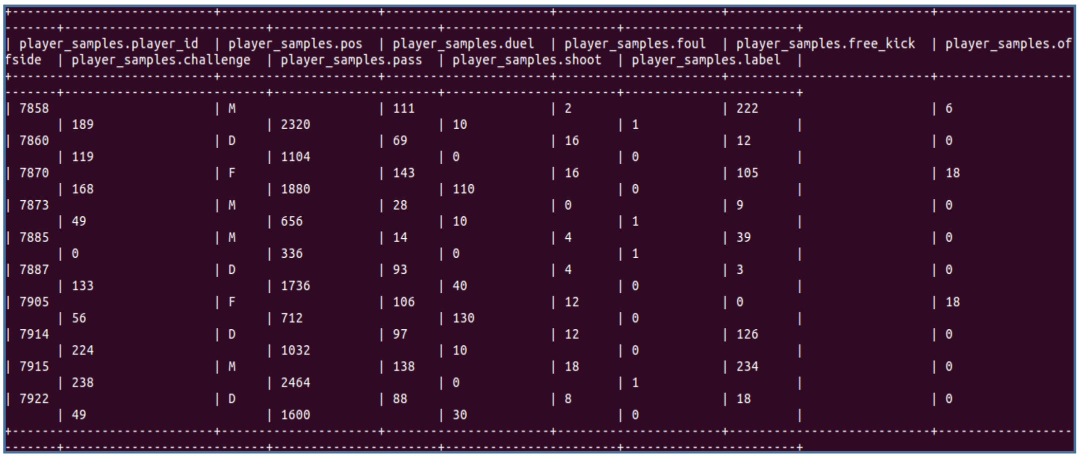
```
0: jdbc:hive2://localhost:10000> create table if not exists player_stats (
. . . . . . . . . . . . . . . .> player_id string, pos string, duel int, foul int, free_kick int, offside int, challenge int,
. . . . . . . . . . . . . . . .> pass int, shoot int, label int)
. . . . . . . . . . . . . . . .> row format delimited fields by ','
. . . . . . . . . . . . . . . .> lines terminated by '\n'
. . . . . . . . . . . . . . . .> stored as textfile;
```

Image 15-2



```
0: jdbc:hive2://localhost:10000> load data local inpath '/home/myjio/datasets/players_stats.csv' into table player_stats;
INFO  : Compiling command(queryId=myjio_20201218022402_caaca43c-0b10-41d5-8ecd-d8e856960ab3): load data local inpath '/home/myjio/datasets/play
ers_stats.csv' into table player_stats
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Semantic Analysis Completed (retrial = false)
INFO  : Returning Hive schema: Schema(fieldSchemas:null, properties:null)
INFO  : Completed compiling command(queryId=myjio_20201218022402_caaca43c-0b10-41d5-8ecd-d8e856960ab3); Time taken: 0.174 seconds
INFO  : Concurrency mode is disabled, not creating a lock manager
INFO  : Executing command(queryId=myjio_20201218022402_caaca43c-0b10-41d5-8ecd-d8e856960ab3): load data local inpath '/home/myjio/datasets/play
ers_stats.csv' into table player_stats
INFO  : Starting task [Stage-0:MOVE] in serial mode
INFO  : Loading data to table soccer.player_stats from file:/home/myjio/datasets/players_stats.csv
INFO  : Starting task [Stage-1:STATS] in serial mode
INFO  : Completed executing command(queryId=myjio_20201218022402_caaca43c-0b10-41d5-8ecd-d8e856960ab3); Time taken: 0.552 seconds
INFO  : OK
INFO  : Concurrency mode is disabled, not creating a lock manager
No rows affected (0.735 seconds) _
```

Image 15-3



```
+-----------------------------+-----------------------+-----------------------+-----------------------+-----------------------------+------------------
-------+-----------------------------+-----------------------+-----------------------+-----------------------------+
| player_samples.player_id  | player_samples.pos  | player_samples.duel  | player_samples.foul  | player_samples.free_kick  | player_samples.of
fside  | player_samples.challenge  | player_samples.pass  | player_samples.shoot  | player_samples.label  |
+-----------------------------+-----------------------+-----------------------+-----------------------+-----------------------------+------------------
-------+-----------------------------+-----------------------+-----------------------+-----------------------------+
| 7858                        | M                     | 111                   | 2                     | 222                         | 6
       | 189                         | 2320                  | 10                    | 1                     |
| 7860                        | D                     | 69                    | 16                    | 12                          | 0
       | 119                         | 1104                  | 0                     | 0                     |
| 7870                        | F                     | 143                   | 16                    | 105                         | 18
       | 168                         | 1880                  | 110                   | 0                     |
| 7873                        | M                     | 28                    | 0                     | 9                           | 0
       | 49                          | 656                   | 10                    | 1                     |
| 7885                        | M                     | 14                    | 4                     | 39                          | 0
       | 0                           | 336                   | 0                     | 1                     |
| 7887                        | D                     | 93                    | 4                     | 3                           | 0
       | 133                         | 1736                  | 40                    | 0                     |
| 7905                        | F                     | 106                   | 12                    | 0                           | 18
       | 56                          | 712                   | 130                   | 0                     |
| 7914                        | D                     | 97                    | 12                    | 126                         | 0
       | 224                         | 1032                  | 10                    | 0                     |
| 7915                        | M                     | 138                   | 18                    | 234                         | 0
       | 238                         | 2464                  | 0                     | 1                     |
| 7922                        | D                     | 88                    | 8                     | 18                          | 0
       | 49                          | 1600                  | 30                    | 0                     |
+-----------------------------+-----------------------+-----------------------+-----------------------+-----------------------------+------------------
-------+-----------------------------+-----------------------+-----------------------+-----------------------------+
```

Image 15-4

STEP 3: Feature representation

First of all, we have to convert the records into pairs of the feature vector and corresponding target value. Here, Hivemall requires you to represent input features in a specific format.

To be more precise, Hivemall represents single feature in a concatenation of index (i.e., name) and its value:

Quantitative feature: <index>:<value>

e.g., pass:135.0

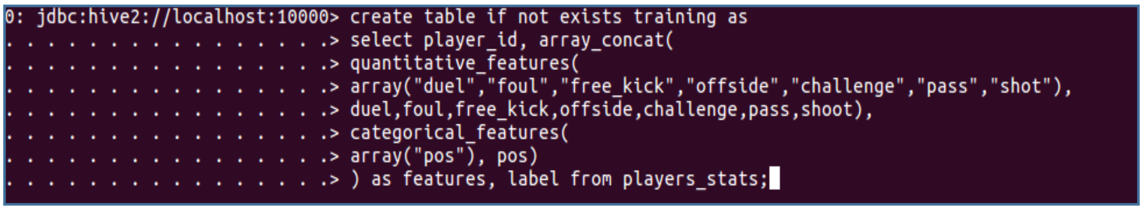
Categorical feature: <index>#<value>

e.g., pos#D

Feature index and feature value are separated by comma. Each of those features is a string value in Hive, and "feature vector" means an array of string values like:

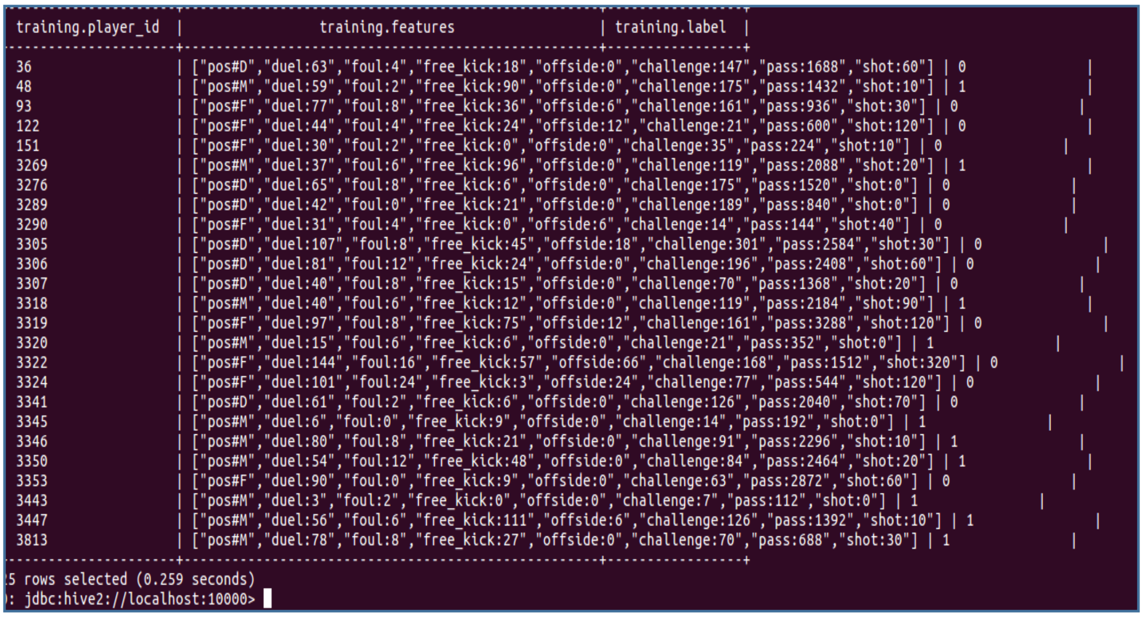
["pass:135.0", "shot:15", "offside:2", "pos#F"]

Therefore, what we first need to do is to convert the records into an array of feature strings, and Hivemall functions quantitative_features(), categorical_features() and array_concat() provide a simple way to create the pairs of feature vector and target value:



```
0: jdbc:hive2://localhost:10000> create table if not exists training as
. . . . . . . . . . . . . . . .> select player_id, array_concat(
. . . . . . . . . . . . . . . .> quantitative_features(
. . . . . . . . . . . . . . . .> array("duel","foul","free_kick","offside","challenge","pass","shot"),
. . . . . . . . . . . . . . . .> duel,foul,free_kick,offside,challenge,pass,shoot),
. . . . . . . . . . . . . . . .> categorical_features(
. . . . . . . . . . . . . . . .> array("pos"), pos)
. . . . . . . . . . . . . . . .> ) as features, label from players_stats;
```

Image 15-5

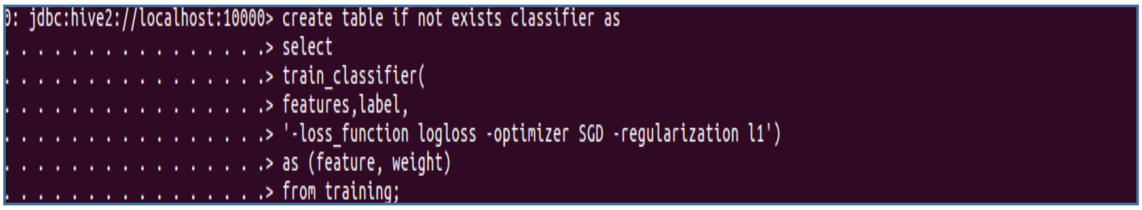


```
--------------------+-------------------------------------------------+----------------+
 training.player_id  |                training.features                | training.label  |
--------------------+-------------------------------------------------+----------------+
 36                  | ["pos#D","duel:63","foul:4","free_kick:18","offside:0","challenge:147","pass:1688","shot:60"] | 0              |
 48                  | ["pos#M","duel:59","foul:2","free_kick:90","offside:0","challenge:175","pass:1432","shot:10"] | 1              |
 93                  | ["pos#F","duel:77","foul:8","free_kick:36","offside:6","challenge:161","pass:936","shot:30"] | 0              |
 122                 | ["pos#F","duel:44","foul:4","free_kick:24","offside:12","challenge:21","pass:600","shot:120"] | 0              |
 151                 | ["pos#F","duel:30","foul:2","free_kick:0","offside:0","challenge:35","pass:224","shot:10"] | 0              |
 3269                | ["pos#M","duel:37","foul:6","free_kick:96","offside:0","challenge:119","pass:2088","shot:20"] | 1              |
 3276                | ["pos#D","duel:65","foul:8","free_kick:6","offside:0","challenge:175","pass:1520","shot:0"] | 0              |
 3289                | ["pos#D","duel:42","foul:0","free_kick:21","offside:0","challenge:189","pass:840","shot:0"] | 0              |
 3290                | ["pos#F","duel:31","foul:4","free_kick:0","offside:6","challenge:14","pass:144","shot:40"] | 0              |
 3305                | ["pos#D","duel:107","foul:8","free_kick:45","offside:18","challenge:301","pass:2584","shot:30"] | 0              |
 3306                | ["pos#D","duel:81","foul:12","free_kick:24","offside:0","challenge:196","pass:2408","shot:60"] | 0              |
 3307                | ["pos#D","duel:40","foul:8","free_kick:15","offside:0","challenge:70","pass:1368","shot:20"] | 0              |
 3318                | ["pos#M","duel:40","foul:6","free_kick:12","offside:0","challenge:119","pass:2184","shot:90"] | 1              |
 3319                | ["pos#F","duel:97","foul:8","free_kick:75","offside:12","challenge:161","pass:3288","shot:120"] | 0              |
 3320                | ["pos#M","duel:15","foul:6","free_kick:6","offside:0","challenge:21","pass:352","shot:0"] | 1              |
 3322                | ["pos#F","duel:144","foul:16","free_kick:57","offside:66","challenge:168","pass:1512","shot:320"] | 0              |
 3324                | ["pos#F","duel:101","foul:24","free_kick:3","offside:24","challenge:77","pass:544","shot:120"] | 0              |
 3341                | ["pos#D","duel:61","foul:2","free_kick:6","offside:0","challenge:126","pass:2040","shot:70"] | 0              |
 3345                | ["pos#M","duel:6","foul:0","free_kick:9","offside:0","challenge:14","pass:192","shot:0"] | 1              |
 3346                | ["pos#M","duel:80","foul:8","free_kick:21","offside:0","challenge:91","pass:2296","shot:10"] | 1              |
 3350                | ["pos#M","duel:54","foul:12","free_kick:48","offside:0","challenge:84","pass:2464","shot:20"] | 1              |
 3353                | ["pos#F","duel:90","foul:0","free_kick:9","offside:0","challenge:63","pass:2872","shot:60"] | 0              |
 3443                | ["pos#M","duel:3","foul:2","free_kick:0","offside:0","challenge:7","pass:112","shot:0"] | 1              |
 3447                | ["pos#M","duel:56","foul:6","free_kick:111","offside:6","challenge:126","pass:1392","shot:10"] | 1              |
 3813                | ["pos#M","duel:78","foul:8","free_kick:27","offside:0","challenge:70","pass:688","shot:30"] | 1              |
--------------------+-------------------------------------------------+----------------+
5 rows selected (0.259 seconds)
: jdbc:hive2://localhost:10000>
```

Image 15-6

STEP 4: Training

Once the original table player_stats has been converted into pairs of features and label, you can build a binary classifier by running the following query:



```
0: jdbc:hive2://localhost:10000> create table if not exists classifier as
. . . . . . . . . . . . . . . .> select
. . . . . . . . . . . . . . . .> train_classifier(
. . . . . . . . . . . . . . . .> features,label,
. . . . . . . . . . . . . . . .> '-loss_function logloss -optimizer SGD -regularization l1')
. . . . . . . . . . . . . . . .> as (feature, weight)
. . . . . . . . . . . . . . . .> from training;
```

Image 15-7

What the above query does is to build a binary classifier with:

- **loss_function logloss**

  Use logistic loss i.e., logistic regression

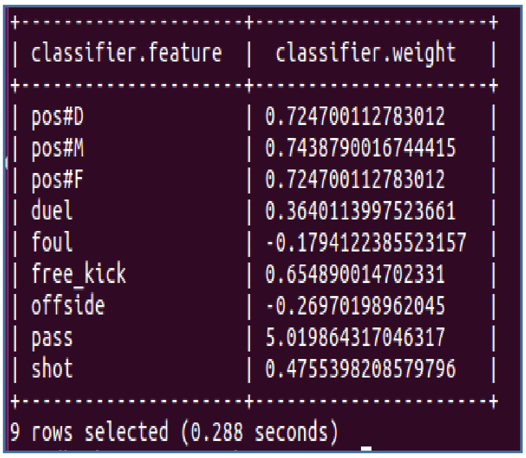
- **optimizer SGD**

  Learn model parameters with the SGD optimization

- **regularization l1**

  Apply L1 regularization

Eventually, the output table classifier stores model parameters as:



```
+--------------------+---------------------+
| classifier.feature |  classifier.weight  |
+--------------------+---------------------+
| pos#D              | 0.724700112783012   |
| pos#M              | 0.7438790016744415  |
| pos#F              | 0.724700112783012   |
| duel               | 0.3640113997523661  |
| foul               | -0.1794122385523157 |
| free_kick          | 0.654890014702331   |
| offside            | -0.26970198962045   |
| pass               | 5.019864317046317   |
| shot               | 0.4755398208579796  |
+--------------------+---------------------+
9 rows selected (0.288 seconds)
```

Image 15-8

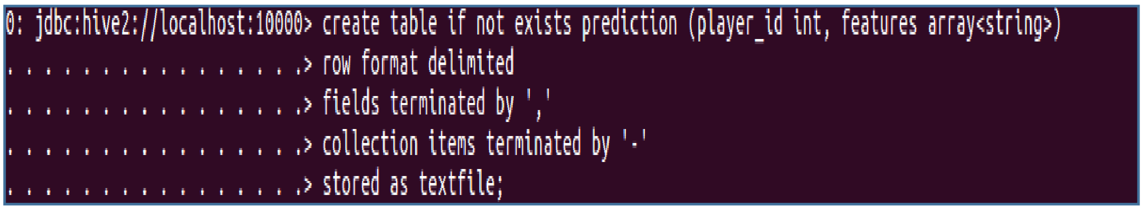
Notice that weight is learned for each possible value in a categorical feature, and for every single quantitative feature.

STEP 5: Prediction

Now, the table classifier has liner coefficients for given features, we can predict unforeseen samples by computing a weighted sum of their features. It is time to test our model if it can predict correct the role of a player based on his statistical attributes.

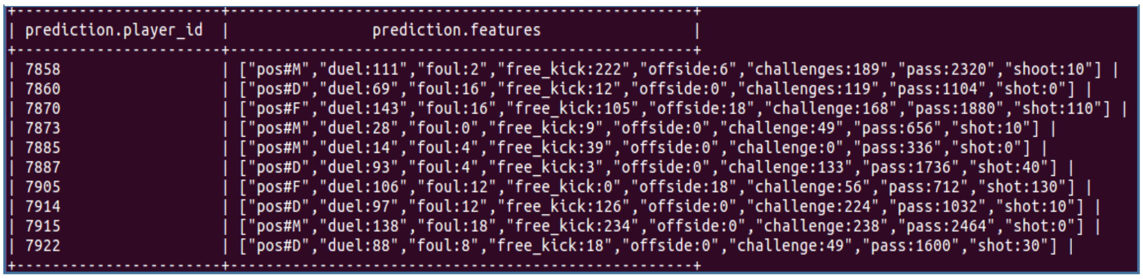
Our goal is for the examples in the table player_samples, our model to predict correctly the players whose position is Midfielder (“M”).

Prediction for the feature vectors can be made by join operation between prediction table (in which we stored a few samples from the table picture_sample) and classifier table on each feature as:



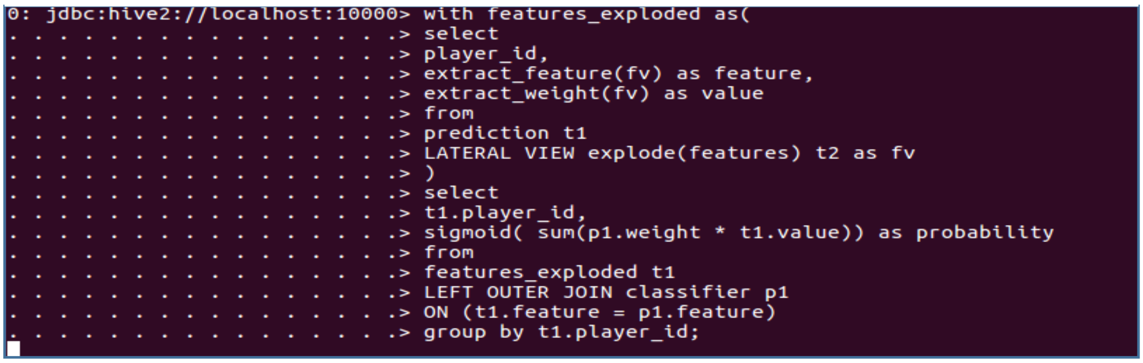
```
0: jdbc:hive2://localhost:10000> create table if not exists prediction (player_id int, features array<string>)
. . . . . . . . . . . . . . . . .> row format delimited
. . . . . . . . . . . . . . . . .> fields terminated by ','
. . . . . . . . . . . . . . . . .> collection items terminated by '-'
. . . . . . . . . . . . . . . . .> stored as textfile;
```

Image 15-9



```
+----------------------+----------------------------------------------------+
| prediction.player_id |                prediction.features                 |
+----------------------+----------------------------------------------------+
| 7858                 | ["pos#M","duel:111","foul:2","free_kick:222","offside:6","challenges:189","pass:2320","shoot:10"] |
| 7860                 | ["pos#D","duel:69","foul:16","free_kick:12","offside:0","challenges:119","pass:1104","shot:0"] |
| 7870                 | ["pos#F","duel:143","foul:16","free_kick:105","offside:18","challenge:168","pass:1880","shot:110"] |
| 7873                 | ["pos#M","duel:28","foul:0","free_kick:9","offside:0","challenge:49","pass:656","shot:10"] |
| 7885                 | ["pos#M","duel:14","foul:4","free_kick:39","offside:0","challenge:0","pass:336","shot:0"] |
| 7887                 | ["pos#D","duel:93","foul:4","free_kick:3","offside:0","challenge:133","pass:1736","shot:40"] |
| 7905                 | ["pos#F","duel:106","foul:12","free_kick:0","offside:18","challenge:56","pass:712","shot:130"] |
| 7914                 | ["pos#D","duel:97","foul:12","free_kick:126","offside:0","challenge:224","pass:1032","shot:10"] |
| 7915                 | ["pos#M","duel:138","foul:18","free_kick:234","offside:0","challenge:238","pass:2464","shot:0"] |
| 7922                 | ["pos#D","duel:88","foul:8","free_kick:18","offside:0","challenge:49","pass:1600","shot:30"] |
+----------------------+----------------------------------------------------+
```

Image 15-10



```
0: jdbc:hive2://localhost:10000> with features_exploded as(
. . . . . . . . . . . . . . . . .> select
. . . . . . . . . . . . . . . . .> player_id,
. . . . . . . . . . . . . . . . .> extract_feature(fv) as feature,
. . . . . . . . . . . . . . . . .> extract_weight(fv) as value
. . . . . . . . . . . . . . . . .> from
. . . . . . . . . . . . . . . . .> prediction t1
. . . . . . . . . . . . . . . . .> LATERAL VIEW explode(features) t2 as fv
. . . . . . . . . . . . . . . . .> )
. . . . . . . . . . . . . . . . .> select
. . . . . . . . . . . . . . . . .> t1.player_id,
. . . . . . . . . . . . . . . . .> sigmoid( sum(p1.weight * t1.value)) as probability
. . . . . . . . . . . . . . . . .> from
. . . . . . . . . . . . . . . . .> features_exploded t1
. . . . . . . . . . . . . . . . .> LEFT OUTER JOIN classifier p1
. . . . . . . . . . . . . . . . .> ON (t1.feature = p1.feature)
. . . . . . . . . . . . . . . . .> group by t1.player_id;
```

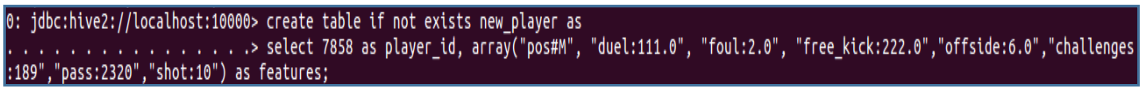
Image 15-11

Some explanations on the above code:

With the code extract_feature(fv) as feature, we split the feature string of prediction table into its name and value,

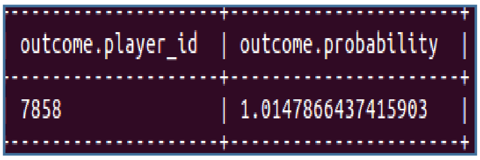
With the line extract_weight(fv) as value, -- to join with the model table.

Because the MapReduce jobs were pushing our system to fail due to lack of physical memory, we made a new table (new_player) and populated with just one sample to test our model. This player is actually a Midfielder so we expect the probability to be equal or exceed 1.



```
0: jdbc:hive2://localhost:10000> create table if not exists new_player as
. . . . . . . . . . . . . . . .> select 7858 as player_id, array("pos#M", "duel:111.0", "foul:2.0", "free_kick:222.0","offside:6.0","challenges
:189","pass:2320","shot:10") as features;
```

Image 15-12

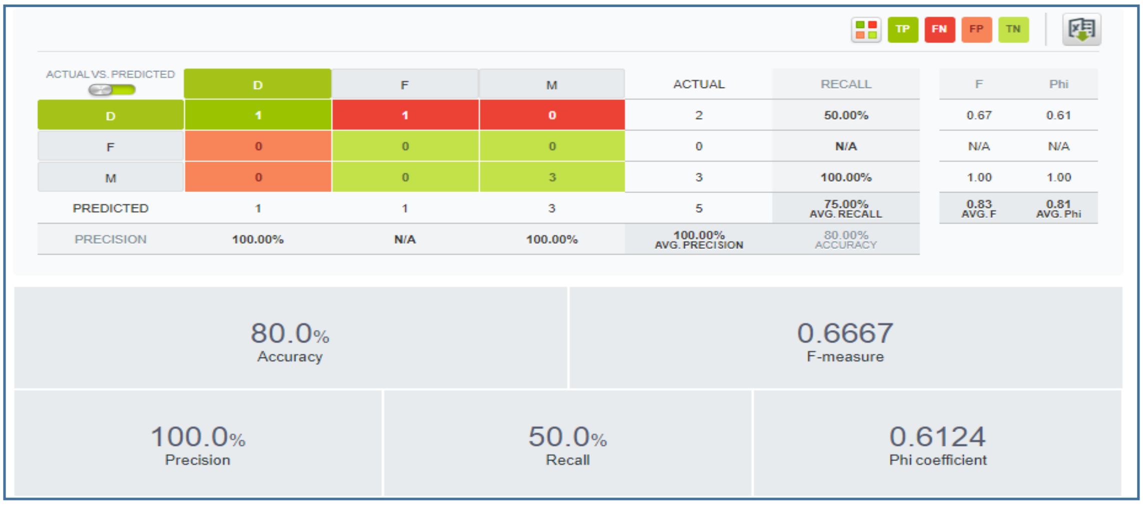


```
+--------------------+-----------------------+
| outcome.player_id  | outcome.probability   |
+--------------------+-----------------------+
| 7858               | 1.0147866437415903    |
+--------------------+-----------------------+
```

Image 15-13

We are thrilled to see that our model has predicted the desired output, because the player with id=7858 is indeed a Midfielder.

Although it is not precise as we should want. We tried to run the same model with an online tool, using the prediction table as our training table and it failed to give 100% accuracy. In fact out of five samples our model could not classify one player.



ACTUAL VS. PREDICTED	D	F	M	ACTUAL	RECALL	F	Phi
D	1	1	0	2	50.00%	0.67	0.61
F	0	0	0	0	N/A	N/A	N/A
M	0	0	3	3	100.00%	1.00	1.00
PREDICTED	1	1	3	5	75.00% AVG. RECALL	0.83 AVG. F	0.81 AVG. Phi
PRECISION	100.00%	N/A	100.00%	100.00% AVG. PRECISION	80.00% ACCURACY		

80.0% Accuracy

0.6667 F-measure

100.0% Precision

50.0% Recall

0.6124 Phi coefficient

Image 15-14

We could add or remove some statistical properties, or even better to get the average values of all the stats of every player, to have a better view of the impact of every player in every statistical area. The main reason is the dataset has all the players of the European Championship 2016 and their statistics accumulated, even though some players played all the matches and others only one. Without this transformation from accumulated to average stats, it is very possible to have some outliers entries in our data set, which we have to erase them.

Hive and Hivemall provides plenty of functions (e.g., rescale(), feature_hashing(), l1_normalize()) for the features in this step to make your prediction model more accurate and stable; it is known as feature engineering in the context of ML.

CHAPTER 16: CONCLUSION

Big Data analysis is the latest popular area for the research and business around the globe. Hadoop is a flexible and open source implementation for analyzing large datasets using map-reduce, but relatively difficult to implement and programming.

As a result, Apache Hive's early success stemmed from the ability to exploit parallelism for batch operations with a well-known interface. It made data load and management simple, handling node, software and hardware failures gracefully without expensive repair or recovery times.

Apache Hive architecture and design principles have proven to be powerful in today's analytic landscape. With the continuous improvements on storage efficiency and query execution performance by the Hive community, Hive has expanded from an ETL tool to fully-fledged enterprise-grade data warehouse. We believe it will continue to thrive in new deployment and storage environments as they emerge, as it is showing today with containerization and cloud.

With Hivemall, Hive users have now the capability to execute Machine Learning algorithms via Hive. Although it is not recommended for large and complex projects and models, it allows users, who want to test their early models for various types of ML algorithms, on the same environment they pre-processing and analyzing with queries their datasets. It is a welcome alternative for testing simple models on sample datasets.

REFERENCES

[1] Hadoop for Dummies, 2014, Dirk deRoos, Paul C.Zikopoulos, Bruce Brown, Rafael Coss, Roman B. Melnyk. John Wiley & Sons, Inc.

[2] Practical Hive_A Guide to Hadoop's Data Warehouse System, 2016, Apress, Scott Shaw, Andreas Francois Vermeulen, Ankur Gupta, David Kjerrumgarrd,

[3] Hadoop Operations, 2012, O'Reilly Media, Eric Sammer

[4] Programming Hive, 2012, O'Reilly Media, Edward Capriolo, Dean Wampler, Jason Rutherglen

[5] Διάλεξη 4η – Τεχνολογίες και Εφαρμογές Υπολογιστικού Νέφους, Ηλίας Σάββας

[6] https://data-flair.training/blogs/apache-hive-tutorial/

[7] Apache Hive Essentials 2nd Edition, 2018, Dayong Du. Packt Publishing

[8] https://www.wikiwand.com/en/Apache_Hive [11.05.2020]

[9] Practical Hive: A Guide to Hadoop's Data Warehouse System, 2016, Scott Shaw, Andreas François Vermeulen, Ankur Gupta, David Kjerrumgaard. Apress; 1st ed. Edition.

[11] https://www.guru99.com/introduction-hive.html

[12] https://searchdatamanagement.techtarget.com/definition/Apache-Hadoop-YARN-Yet-Another-Resource-Negotiator

[13] Apache Hive Cookbook, 2016, Hanish Bansal, Saurabh Chauhan, Shrey Mehrotra. Packt Publishing

[14] https://data-flair.training/blogs/apache-hive-tutorial/

[15] https://streamsets.com/documentation/controlhub/latest/help/datacollector/UserGuide/Executors/HiveQuery.html [15.12.2020]

[10][16] https://cwiki.apache.org/ [15.12.2020]

[17] https://www.geeksforgeeks.org/architecture-and-working-of-hive/

[18] https://www.tutorialspoint.com/hive/hive_create_database.htm

[19] https://blog.dataiku.com/2013/05/01/a-complete-guide-to-writing-hive-udf

[20] https://cwiki.apache.org/confluence/display/Hive/HivePlugins

[21] Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing, various researchers, report, 26 March 2019

[22] Apache Hive: Enterprise SQL on Big Data frameworks, Manish A. Kukreja, report, 27 July 2016