Story of Hive:

What is Hive?

Apache Hive is a data warehouse system built on top of Hadoop and is used for analyzing structured and semi-structured data. Hive abstracts the complexity of Hadoop MapReduce.

Basically, it provides a mechanism to project structure onto the data and perform queries written in HQL (Hive Query Language) that are similar to SQL statements.

Internally, these queries or HQL gets converted to map reduce jobs by the Hive compiler. Therefore, you don’t need to worry about writing complex MapReduce programs to process your data using Hadoop. It is targeted towards users who are comfortable with SQL.

Apache Hive supports Data Definition Language (DDL), Data Manipulation Language (DML) and User Defined Functions (UDF)

SQL + Hadoop MapReduce = HiveQL

The story of Hive — from Facebook to Apache



Challenges at Facebook: Exponential Growth of Data

Before 2008, all the data processing infrastructure in Facebook was built around a data warehouse based on commercial RDBMS. These infrastructures were capable enough to suffice the needs of Facebook at that time. But, as the data started growing very fast, it became a huge challenge to manage and process this

huge dataset. According to a Facebook article, the data scaled from 15 TB dataset in 2007 to a 2 PB data in 2009. Also, many Facebook products involve the analysis of the data like Audience Insights, Facebook Lexicon, Facebook Ads, etc. So, they needed a scalable and economical solution to cope up with this very problem and, therefore started using the Hadoop framework.

Democratizing Hadoop — MapReduce

But, as the data grew, the complexity of Map-Reduce codes grew proportionally. So, training people with a non-programming background to write MapReduce programs became difficult. Also, for performing simple analysis one has to write a hundred lines of MapReduce code. Since, SQL was widely used by engineers and analysts, including Facebook, therefore, putting SQL on the top of Hadoop seemed a logical way to make Hadoop accessible to users with SQL background.

Hence, the ability of SQL to suffice for most of the analytic requirements and the scalability of Hadoop gave birth to Apache Hive that allows performing SQL like queries on the data present in HDFS. Later, the Hive project was open sourced in August’ 2008 by Facebook and is freely available as Apache Hive today.

Advantages of Hive

Useful for people who aren’t from a programming background as it eliminates the need to write complex MapReduce program.

Extensible and scalable to cope up with the growing volume and variety of data, without affecting the performance of the system.

It is as an efficient ETL (Extract, Transform, Load) tool.

Hive supports any client application written in Java, PHP, Python, C++ or Ruby by exposing its Thrift server. (You can use these client-side languages embedded with SQL for accessing a database such as DB2, etc.).

As the metadata information of Hive is stored in an RDBMS, it significantly reduces the time to perform semantic checks during query execution.

Where to use Apache Hive?

Apache Hive takes advantage of both the worlds i.e. SQL Database System and Hadoop — MapReduce framework.

Therefore, it is used by a vast multitude of companies. It is mostly used for data warehousing where you can perform analytics and data mining that does not require real-time processing.

Some of the fields where you can use Apache Hive are as follows:

Data Warehousing

Ad-hoc Analysis

There are two types of tables in Hive:

1. Managed Table:

CREATE TABLE <table\_name> (column1 data\_type, column2 data\_type);

LOAD DATA INPATH <HDFS\_file\_location> INTO table managed\_table;

As the name suggests (managed table), Hive is responsible for managing the data of a managed table. In other words, what I meant by saying, “Hive manages the data”, is that if you load the data from a file present in HDFS into a Hive Managed Table and issue a DROP command on it, the table along with its metadata will be deleted. So, the data belonging to the dropped managed\_table no longer exist anywhere in HDFS and you can’t retrieve it by any means. Basically, you are moving the data when you issue the LOAD command from the HDFS file location to the Hive warehouse directory.

2. External Table:

CREATE EXTERNAL TABLE <table\_name> (column1 data\_type, column2 data\_type) LOCATION ‘<table\_hive\_location>’;

LOAD DATA INPATH ‘<HDFS\_file\_location>’ INTO TABLE <table\_name>;

For an external table, Hive is not responsible for managing the data.

In this case, when you issue the LOAD command, Hive moves the data into its warehouse directory. Then, Hive creates the metadata information for the external table. Now, if you issue a DROP command on the external table, only metadata information regarding the external table will be deleted.

Therefore, you can still retrieve the data of that very external table from the warehouse directory using HDFS commands.

Partitions:

CREATE TABLE table\_name (column1 data\_type, column2 data\_type) PARTITIONED BY (partition1 data\_type, partition2 data\_type,….);

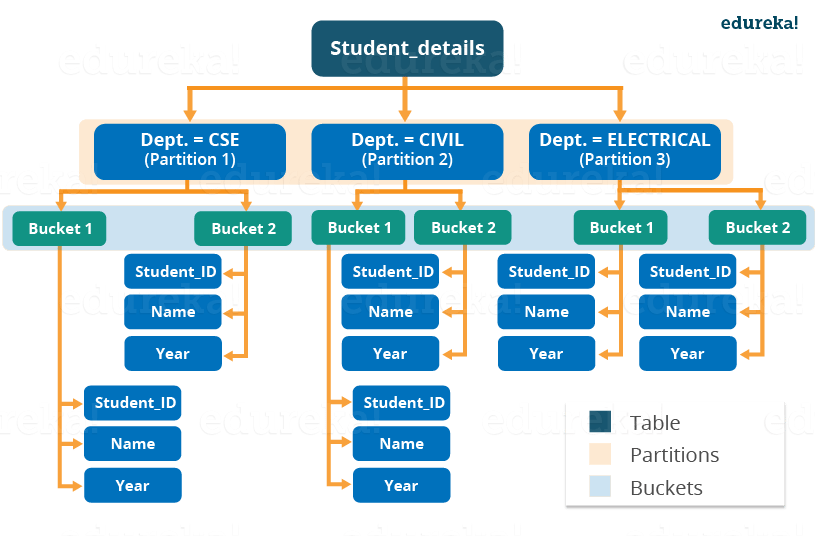
Hive organizes tables into partitions for grouping similar type of data together based on a column or partition key. Each Table can have one or more partition keys to identify a particular partition. This allows us to have a faster query on slices of the data.

Note: Remember, the most common mistake made while creating partitions is to specify an existing column name as a partition column. While doing so, you will receive an error — “Error in semantic analysis: Column repeated in partitioning columns”.

Let us understand partition by taking an example where I have a table student\_details containing the student information of some engineering college like student\_id, name, department, year, etc. Now, if I perform partitioning based on department column, the information of all the students belonging to a particular department will be stored together in that very partition. Physically, a partition is nothing but a sub-directory in the table directory.

Let’s say we have data for three departments in our student\_details table — CSE, ECE, and Civil. Therefore, we will have three partitions in total for each of the departments as shown in the image below. And, for each department, we will have all the data regarding that very department residing in a separate subdirectory under the Hive table directory. For example, all the student data regarding CSE departments will be stored in user/hive/warehouse/student\_details/dept.=CSE. So, the queries regarding CSE students would only have to look through the data present in the CSE partition. This makes partitioning very useful as it reduces the query latency by scanning only relevant partitioned data instead of the whole data set. In fact, in real-world implementations, you will be dealing with hundreds of TBs of data. So, imagine scanning this huge amount of data for some query where 95% data scanned by you was un-relevant to your query.

I would suggest you go through the blog on Hive commands where you will find different ways of implementing partitions with an example.



Buckets:

CREATE TABLE table\_name PARTITIONED BY (partition1 data\_type, partition2 data\_type,….) CLUSTERED BY (column\_name1, column\_name2, …) SORTED BY (column\_name [ASC|DESC], …)] INTO num\_buckets BUCKETS;

Now, you may divide each partition or the unpartitioned table into Buckets based on the hash function of a column in the table. Actually, each bucket is just a file in the partition directory or the table directory (unpartitioned table). Therefore, if you have chosen to divide the partitions into n buckets, you will have n files in each of your partition directories. For example, you can see the above image

where we have bucketed each partition into 2 buckets. So, each partition, say CSE, will have two files where each of them will be storing the CSE student’s data.

How Hive distributes the rows into buckets?

Well, Hive determines the bucket number for a row by using the formula: hash\_function (bucketing\_column) modulo (num\_of\_buckets). Here, hash\_function depends on the column data type. For example, if you are

bucketing the table on the basis of some column, let’s say user\_id, of INT datatype, the hash\_function will be — hash\_function (user\_id)= integer value of user\_id. And, suppose you have created two buckets, then Hive will determine the rows going to bucket 1 in each partition by calculating: (value of user\_id)

modulo (2). Therefore, in this case, rows having user\_id ending with an even integer digit will reside in the same bucket corresponding to each partition. The hash\_function for other data types is a bit complex to calculate and in fact, for a string, it is not even humanly recognizable.