UNIT – VII

Introduction to Hive -

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

The Problem & Solution -:

All Hadoop sub-projects such as Hive, Pig, and HBase support Linux operating system. Therefore, you need to install any Linux flavored OS. The following simple steps are executed for Hive installation:

Step 1: Verifying JAVA Installation

Java must be installed on your system before installing Hive. Let us verify java installation using the following command:

```
$ java –version
```

If Java is already installed on your system, you get to see the following response:

```
java version "1.7.0_71"
Java(TM) SE Runtime Environment (build 1.7.0_71-b13)
Java HotSpot(TM) Client VM (build 25.0-b02, mixed mode)
```

If java is not installed in your system, then follow the steps given below for installing java.

Installing Java

Step I:

Download java (JDK <latest version> - X64.tar.gz) by visiting the following link http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html.

Then jdk-7u71-linux-x64.tar.gz will be downloaded onto your system.

Step II:

Generally you will find the downloaded java file in the Downloads folder. Verify it and extract the jdk-7u71-linux-x64.gz file using the following commands.

```
$ cd Downloads/
$ ls
jdk-7u71-linux-x64.gz
$ tar zxf jdk-7u71-linux-x64.gz
$ ls
```

```
jdk1.7.0_71 jdk-7u71-linux-x64.gz
```

Step III:

To make java available to all the users, you have to move it to the location "/usr/local/". Open root, and type the following commands.

```
$ su
password:
# mv jdk1.7.0_71 /usr/local/
# exit
```

Step IV:

For setting up PATH and JAVA_HOME variables, add the following commands to ~/.bashrc file.

```
export JAVA_HOME=/usr/local/jdk1.7.0_71
export PATH=$PATH:$JAVA_HOME/bin
```

Now apply all the changes into the current running system.

```
$ source ~/.bashrc
```

Step V:

Use the following commands to configure java alternatives:

```
# alternatives --install /usr/bin/java java usr/local/java/bin/java 2

# alternatives --install /usr/bin/javac javac usr/local/java/bin/javac 2

# alternatives --install /usr/bin/jar jar usr/local/java/bin/jar 2

# alternatives --set java usr/local/java/bin/java

# alternatives --set javac usr/local/java/bin/javac
# alternatives --set jar usr/local/java/bin/jar
```

Now verify the installation using the command java -version from the terminal as explained above.

Step 2: Verifying Hadoop Installation

Hadoop must be installed on your system before installing Hive. Let us verify the Hadoop installation using the following command:

```
$ hadoop version
```

If Hadoop is already installed on your system, then you will get the following response:

```
Hadoop 2.4.1 Subversion https://svn.apache.org/repos/asf/hadoop/common
-r 1529768
Compiled by hortonmu on 2013-10-07T06:28Z
Compiled with protoc 2.5.0
From source with checksum 79e53ce7994d1628b240f09af91e1af4
```

If Hadoop is not installed on your system, then proceed with the following steps:

Downloading Hadoop

Download and extract Hadoop 2.4.1 from Apache Software Foundation using the following commands.

```
$ su
password:
# cd /usr/local
# wget http://apache.claz.org/hadoop/common/hadoop-2.4.1/
hadoop-2.4.1.tar.gz
# tar xzf hadoop-2.4.1.tar.gz
# mv hadoop-2.4.1/* to hadoop/
# exit
```

Installing Hadoop in Pseudo Distributed Mode

The following steps are used to install Hadoop 2.4.1 in pseudo distributed mode.

Step I: Setting up Hadoop

You can set Hadoop environment variables by appending the following commands to ~/.bashrc file.

```
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export
PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
```

Now apply all the changes into the current running system.

```
$ source ~/.bashrc
```

Step II: Hadoop Configuration

You can find all the Hadoop configuration files in the location "$HADOOP_HOME/etc/hadoop". You need to make suitable changes in those configuration files according to your Hadoop infrastructure.

```
$ cd $HADOOP_HOME/etc/hadoop
```

In order to develop Hadoop programs using java, you have to reset the java environment variables in hadoop-env.sh file by replacing JAVA_HOME value with the location of java in your system.

```
export JAVA_HOME=/usr/local/jdk1.7.0_71
```

Given below are the list of files that you have to edit to configure Hadoop.

core-site.xml

The core-site.xml file contains information such as the port number used for Hadoop instance, memory allocated for the file system, memory limit for storing the data, and the size of Read/Write buffers.

Open the core-site.xml and add the following properties in between the <configuration> and </configuration> tags.

```
<configuration>

  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>

</configuration>
```

hdfs-site.xml

The hdfs-site.xml file contains information such as the value of replication data, the namenode path, and the datanode path of your local file systems. It means the place where you want to store the Hadoop infra.

Let us assume the following data.

```
dfs.replication (data replication value) = 1

(In the following path /hadoop/ is the user name.
hadoopinfra/hdfs/namenode is the directory created by hdfs file system.)

namenode path = //home/hadoop/hadoopinfra/hdfs/namenode

(hadoopinfra/hdfs/datanode is the directory created by hdfs file system.)
datanode path = //home/hadoop/hadoopinfra/hdfs/datanode
```

Open this file and add the following properties in between the <configuration>, </configuration> tags in this file.

```
<configuration>


  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
    <value>file:///home/hadoop/hadoopinfra/hdfs/namenode </value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>file:///home/hadoop/hadoopinfra/hdfs/datanode </value >
  </property>


</configuration>
```

Note: In the above file, all the property values are user-defined and you can make changes according to your Hadoop infrastructure.

yarn-site.xml

This file is used to configure yarn into Hadoop. Open the yarn-site.xml file and add the following properties in between the <configuration>, </configuration> tags in this file.

```
<configuration>


  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>

</configuration>
```

mapred-site.xml

This file is used to specify which MapReduce framework we are using. By default, Hadoop contains a template of yarn-site.xml. First of all, you need to copy the file from mapred-site,xml.template to mapred-site.xml file using the following command.

```
$ cp mapred-site.xml.template mapred-site.xml
```

Open mapred-site.xml file and add the following properties in between the <configuration>, </configuration> tags in this file.

```
<configuration>


  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>

</configuration>
```

Verifying Hadoop Installation

The following steps are used to verify the Hadoop installation.

Step I: Name Node Setup

Set up the namenode using the command “hdfs namenode -format” as follows.

```
$ cd ~
$ hdfs namenode -format
```

The expected result is as follows.

```
10/24/14 21:30:55 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG: host = localhost/192.168.1.11
STARTUP_MSG: args = [-format]
STARTUP_MSG: version = 2.4.1
...
...
10/24/14 21:30:56 INFO common.Storage: Storage directory
/home/hadoop/hadoopinfra/hdfs/namenode has been successfully formatted.
10/24/14 21:30:56 INFO namenode.NNStorageRetentionManager: Going to
retain 1 images with txid >= 0
10/24/14 21:30:56 INFO util.ExitUtil: Exiting with status 0
10/24/14 21:30:56 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at localhost/192.168.1.11
 ************************************************************/
```

Step II: Verifying Hadoop dfs

The following command is used to start dfs. Executing this command will start your Hadoop file system.

```
$ start-dfs.sh
```

The expected output is as follows:

```
10/24/14 21:37:56
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-namenode-localhost.out
localhost: starting datanode, logging to /home/hadoop/hadoop-
2.4.1/logs/hadoop-hadoop-datanode-localhost.out
Starting secondary namenodes [0.0.0.0]
```

Step III: Verifying Yarn Script

The following command is used to start the yarn script. Executing this command will start your yarn daemons.

```
$ start-yarn.sh
```

The expected output is as follows:

```
starting yarn daemons
starting resourcemanager, logging to /home/hadoop/hadoop-2.4.1/logs/yarn-
hadoop-resourcemanager-localhost.out
localhost: starting nodemanager, logging to /home/hadoop/hadoop-
2.4.1/logs/yarn-hadoop-nodemanager-localhost.out
```

Step IV: Accessing Hadoop on Browser

The default port number to access Hadoop is 50070. Use the following url to get Hadoop services on your browser.

```
http://localhost:50070/
```

Hive Use Case:

Hive it takes large amount of unstructured data and place it into a structured view as shown in Fig.3.,that can be used by business analysts by the business tools. Hive supports use cases such as Ad-hoc queries,summarization,data analysis. Fig.3. HIVEQL can also be exchange with custom scalar functions means user defined functions(UDF ' S),aggregations(UDFA's) and table functions(UDTF's). 3. HIVE ARCHITECTURE Fig.4. As shown in Fig.4 . Hive Architecture consists of A. Meta store It stores schema information and provides a structure to stored data . B.HIVEQL Which is for query processing,compiling,optimizing and execution ,those queries turned into MapReduce job which invokes mapper and reducer for processing. HIVEQL is similar to other SQLs ,it uses familiar relational database concepts(tables,rows,columns,schema),it is based on the SQL-92 specifications. Hive supports multi-table inserts via the code means big data can be accessed via tables. It converts SQL queries into MapReduce jobs. User does not need to know MapReduce. Also supports plugging custom MapReduce scripts into queries. C. User Interface Hive can be approached by number of user interfaces. It has web UI,it has own command line interface and can be accessed using HDInsight. Hive is good choice when you want to query the data,when you need an answer to specific questions if you are familiar with SQL. 4. Using of HIVE with HADOOP. A .Prerequisites to work with Hive The prerequisites for setting up Hive and running queries are 1.User should have stable build of Hadoop 2.Machine Should have Java 1,6 installed 3.Basic Java Programming skills 4.Basic SQL Knowledge 5.Basic Knowledge of Linux To start Hive first start all the services of Hadoop using the command $ start-all.sh. Then check all services are running or not using JPS command ,if all are services are running then Use the command $ hive to start HIVE B.HIVE TABLES Here we are explaining about basics of Hive Tables. A Hive table consists of Data: It is typically a file or group of files in HDFS. Schema:It is in the form of meta data stored in a relational database. Schema and data are separate .A schema can be defined for existing data. Data can be added or removed independently. You have to define a schema if you have existing data in HDFS that you want to use in Hive C. Managing Tables See current tables hive>Show Tables; Check the schema hive>Describe mytable; change the table name hive> ALTER TABLE mytablename RENAME to mt; Add a Column hive>ALTER TABLE mytable ADD COLOUMNS (mycol STRING); Drop a Partition hive >ALTER TABLE mytable DROP PARTITION(age=17) D. Loading Data Use LOAD DATA to import data into Hive Table hive>LOAD DATA LOCAL INPATH ' path of input data' INTO TABLE mytable; The files are not modified by Hive. Use the word OVERWRITE[3][4] to write over a file of the same name. Hive warehouse default location is /hive/warehouse. Hive can read all of the files in a particular directory. The schema is checked when the data is required and if a row does not match the schema,it will be read as null. E. Insert Use INSERT[3] statement to populate data into a table from another HIVE table. Since query results are usually large it is best to use an INSERT clause to tell HIVE where to store your query. Example: hive>CREATE TABLE age_count(name STRING,age INT); hive>INSERT OVERWRITE TABLE age_count SELECT age,COUNT(age) FROM mytable; F. Performing Queries SELECT statement is used for this. It supports the following: Where clause UNION ALL and DISTINCT GROUP BY and HAVING LIMIT clause .

Data Growth :

The Nature of Analytic Data
The first place we started was by thinking about our data, now that we were moving out of the “validation” and into the “scaling” phase of our business.

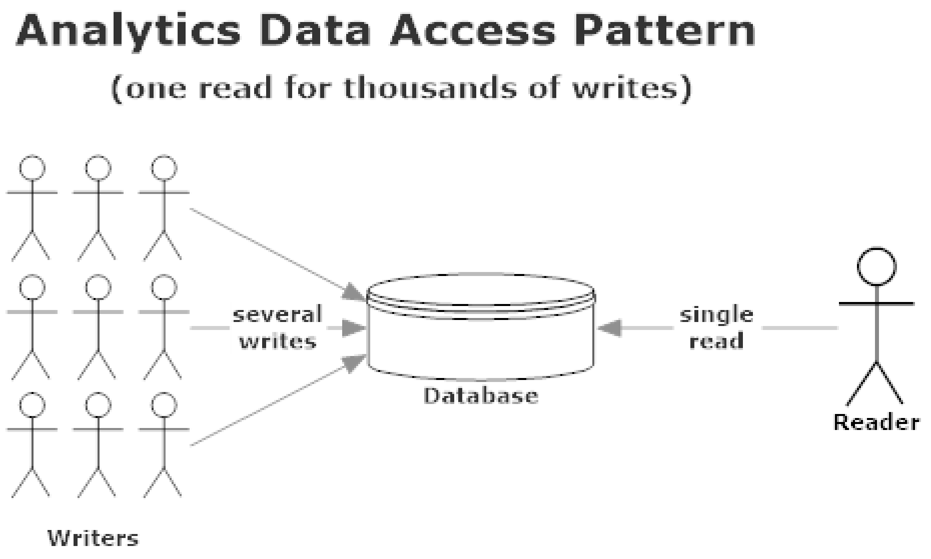
Analytics is a weird business when it comes to read / write characteristics and data access patterns.

In most CRUD applications, mobile apps, and e-commerce software you tend to see read / write characteristics like this:

This isn’t a controversial opinion – it’s just a fact of how most networked applications work. *Data is read far more often than it’s written.*

That’s why all relational databases and most document databases are optimized to cache frequently read items into memory – because that’s how the data is used in the vast majority of use cases.

In analytics though, the relationship is inverted:


Analytics Data Access Pattern
(one read for thousands of writes)
several writes
Database
single read
Reader
Writers

By the time a MarkedUp customer views a report on our dashboard, that data has been written to anywhere from 1,000 to 10,000,000 times since they viewed their report last. *In analytics, data is written multiple orders of magnitude more frequently than it's read.*

So what implications does this have for our choice of database?

Database Criteria
Looking back to what went wrong with RavenDB, we determined that it was fundamentally flawed in the following ways:

- Raven's indexing system is very expensive on disk, which makes it difficult to scale vertically – even on SSDs *Raven's indexing system would keep indexes stale by as much as three or four days*;
- Raven's map/reduce system requires re-aggregation once it's written by our data collection API, which works great at low volumes but scales at an inverted ratio to data growth – the more people using us, the worse the performance gets for everyone;
- Raven's sharding system is really more of a hack at the client level which marries your network topology to your data, which is a really bad design choice – it literally appends the ID of your server to all document identifiers;
- Raven's sharding system actually makes read performance on indices orders of magnitude worse (has to hit every server in the cluster on every request to an index) and doesn't alleviate any issues with writing to indexes – no benefit there;

- Raven's map/reduce pipeline was too simplistic, which stopped us from being able to do some more in-depth queries that we wanted; and
- We had to figure out everything related to RavenDB on our own – we even had to write our own backup software and our own indexing-building tool for RavenDB; there's very little in the way of a RavenDB ecosystem.

So based on all of this, we decided that our next database system needed to be capable of:

1. Integrating with Hadoop and the Hadoop ecosystem, so we could get more powerful map/reduce capabilities;
2. "Linear" hardware scale – make it easy for us to increase our service's capacity with better / more hardware;
3. Aggregate-on-write – eliminate the need to constantly iterate over our data set;
4. Utilizing higher I/O – it's difficult to get RavenDB to move any of its I/O to memory, hence why it's so hard on disk;
5. Fast setup time – need to be able to move quickly;
6. Great ecosystem support – we don't want to be the biggest company using whatever database we pick next.

The Candidates
Based on all of the above criteria, we narrowed down the field of contenders to the following:

1. MongoDB
2. Riak
3. HBase
4. Cassandra

Evaluation Process
The biggest factor to consider in our migration was time to deployment – how quickly could we move off of Raven and restore a high quality of service for our customers?

We tested this in two phases:

1. Learning curve of the database – how long would it take us to set up an actual cluster and a basic test schema?
2. Acceptance test – how quickly could we recreate a median-difficulty query on any of these systems?

So we did this in phases, as a team – first up was HBase.

HBase

HBase was highly recommended to us by some of our friends on the analytics team at Hulu, so this was first on our list. HBase has a lot of attractive features and satisfied most of our technical requirements, save the most important one – time to deployment.

The fundamental problem with HBase is that cluster setup is difficult, particularly if you don't have much JVM experience (we didn't.) It also has a single point of failure (edit: turns out this hasn't been an issue since 0.9x,) is a memory hog, and has a lot of moving parts.

That being said, HBase is a workhorse – it's capable of handling immensely large workloads. Ultimately we decided that it was overkill for us at this stage in our company and the setup overhead was too expensive. We'll likely revisit HBase at some point in the future though.

Riak

One of our advisors is a heavy Riak user, so we decided it was worth exploring. Riak, on the surface, is a very impressive database – it's heinously easy to set up a cluster and the HTTP REST API made it possible for us to test it using only curl.

After getting an initial 4-node cluster setup and writing a couple of "hello world" applications, we decided that it was time to move onto phase 2: see how long it would take to port a real portion of our analytics engine over to Riak.

I decided to use Node.JS for this since there's great node drivers for both Raven and Riak and it was frankly a lot less work than C#. I should point out that CorrugatedIron is a decent C# driver for Riak though.

So, it took me about 6 hours to write the script to migrate a decent-sized data set into Riak – just enough to simulate a real query for a single MarkedUp app.

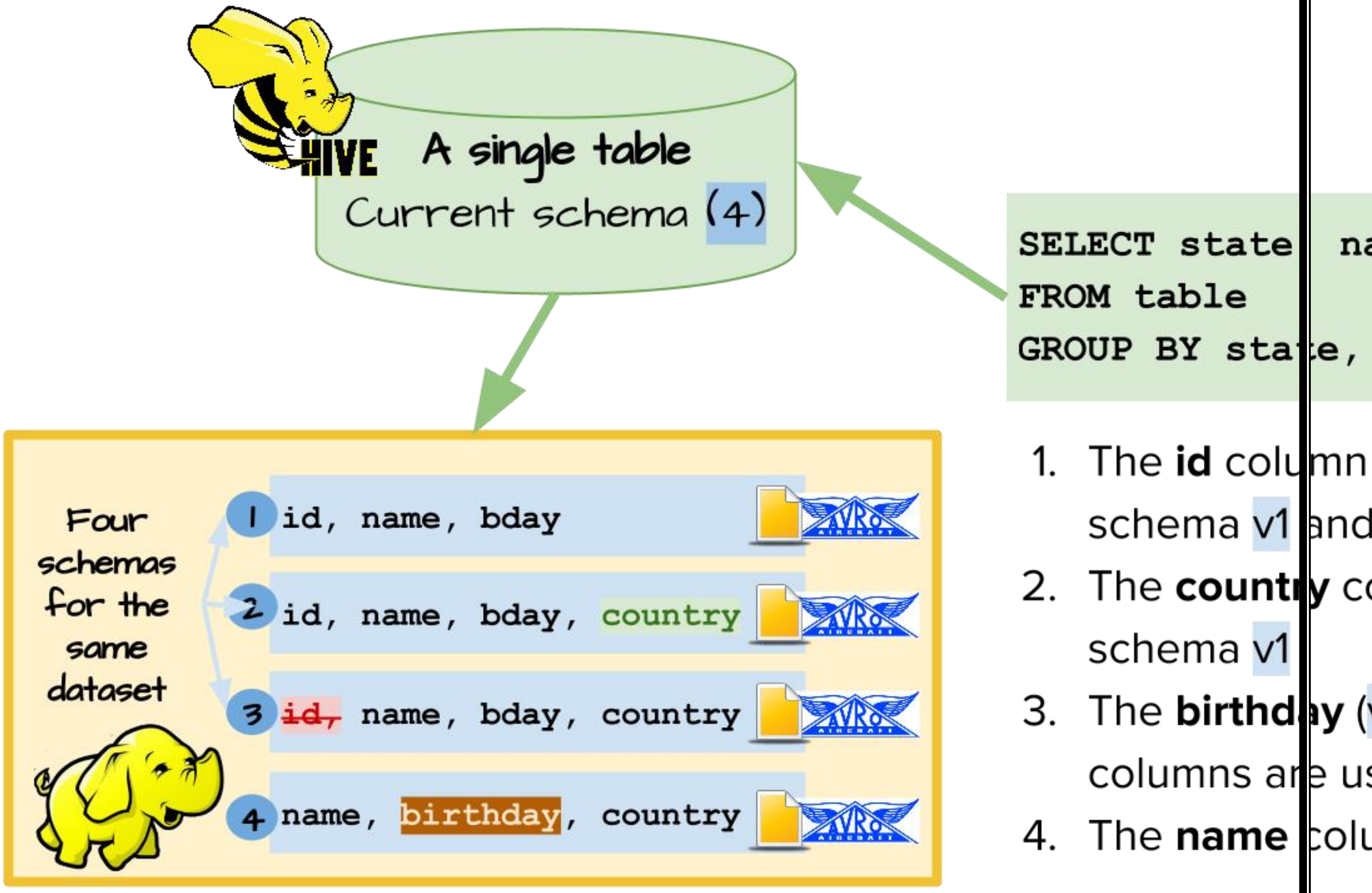
Once we had the data stuffed into our Riak cluster I wrote a simple map/reduce query using JavaScript and ran it – took 90 seconds to run a basic count query. *Yeesh.* And this map/reduce query even used key filtering and all of the other m/r best practices for Riak.

Turns out that Map/Reduce performance with the JavaScript VM is atrocious and well-known in Riak.

Schema Flexibilitym, Evolution & Extensibility:

There are two main types of “schema-compatibility” – backward and forward. With backward compatibility, a new schema can be applied to read the data created using previous schemas. With forward compatibility, an older schema can be applied to read data created using newer schemas – it’s useful when schema evolves, but (legacy) applications may not be updated immediately and still need to read new files written in newer schemas (and perhaps skipping new features).

Below, you can find the picture that illustrates the backward schema compatibility, where a single Hive query can read Avro files written in four different, but backward compatible schemas.



You can see that a number of operations can be allowed as a simple requirement change:

 Adding a new column to a table (the “country” column in the 2nd file)

- Dropping a column from a table (the “id” column in the 3nd file)
- Renaming a column (the “birthday” column in the 4th file)

Let’s see the scheme evolution in practice. We will re-produce exactly the same example that you see on the picture above.

Download input Avro files

You can download the input Avro files here. Each file contains data written in a different, but backward-compatible schema.

Create a Hive table backed by Avro format

The table contains only three columns: `id`, `name` and `bday`.

Load an Avro file into the table

Our input file contains three fields – the same names and types as in our table’s columns.

```
$ java -jar avro-tools-1.7.4.jar tojson user1.avro

{"id":1,"name":"adam","bday":"1985-09-30"}
```

We can load this file into the Hive table by simply placing it in the table’s directory in HDFS.

```
$ hadoop fs -put user1.avro /*/hive/warehouse/user_profile
```

We can query our table and see that expected results:

```
hive> SELECT * FROM user_profile;

1    adam    1985-09-30
```

Modify the schema by adding an extra column

Let's now modify the schema of our Hive table by adding an extra column (at the end):

```
hive> ALTER TABLE user_profile ADD COLUMNS (country STRING);
```

From now, the Hive table will use new schema that contains four columns when reading its data. Because our input file doesn't contain the country column, we should see NULLs in its place.

```
hive> select * from user_profile;

1    adam    1985-09-30    NULL
```

```
$ hadoop fs -put user2.avro /apps/hive/warehouse/user_profile
```

We can query this table – the new schema will be applied for both the first and the second file.

```
hive> SELECT * FROM user_profile;

1    adam    1985-09-30    NULL
2    natalia    1989-08-09    Poland
```

So far so good!

Modify the schema by removing an existing column

```
hive> ALTER TABLE user_profile
   REPLACE COLUMNS (name STRING, bday STRING, country STRING);
...
FAILED: Execution Error, return code 1
    from org.apache.hadoop.hive.ql.exec.DDLTask.
    Replace columns is not supported for table default.user_profile.
    SerDe may be incompatible.
```

Well, the above command doesn't work for Avro format…

The working method is to actually drop the whole table, and create it with a new Avro schema.

```
hive> DROP TABLE user_profile;
hive> CREATE EXTERNAL TABLE user_profile (name STRING, bday STRING, country STRING)
   STORED AS avro;
hive> DESCRIBE FORMATTED user_profile;
```

Please note that this method should be only used when you create an EXTERNAL Hive table (because the data in HDFS is not removed when dropping the table).

If we now query the table, the ID column is ignored (not visible), while the values in the three other columns are shown:

```
hive> SELECT * FROM user_profile;

adam    1985-09-30    NULL
natalia    1989-08-09    Poland
```

Load the third Avro file into the table

Modify the schema by renaming an existing column

According to my knowledge, there is no easy way to rename column in Hive table. The way that I know is to re-create the whole table again and specify Avro schema as

```
hive> DROP TABLE user_profile;

hive> CREATE EXTERNAL TABLE user_profile
 ROW FORMAT SERDE
 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
 STORED AS INPUTFORMAT
 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
 OUTPUTFORMAT
 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
 TBLPROPERTIES (
  'avro.schema.literal'='{
   "type" : "record",
   "name" : "user",
   "namespace" : "com.getindata.avro",
   "fields" : [ {"name" : "name", "type" : "string"},
                {"name" : "birthday", "type" : "string", "aliases" : ["bday"]},
                {"name" : "country", "type" : "string", "default" : "NULL"}
              ]
}');
```

```

hive> DESCRIBE user_profile;

name                    string
birthday                string
country                 string
```

Note that we specify that birthday is the new name of the column that previously was named bday – this is specified using the aliases property.

```
hive> SELECT * FROM user_profile;

adam    1985-09-30    NULL
natalia    1989-08-09    Poland
tofi    2006-06-06    Sweden
```

```
java -jar avro-tools-1.7.4.jar tojson user4.avro

{"name":"fox","birthday":"2013-03-22","country":"Poland"}
```

```
hadoop fs -put user4.avro /apps/hive/warehouse/user_profile
```

As we can see, Hive is handling it well and printing both birthday and bday fields in

```
hive> SELECT * FROM user_profile;

```

```
3 adam     1985-09-30    NULL
4 natalia    1989-08-09    Poland
5 tofi    2006-06-06    Sweden
6 fox    2013-03-22    Poland
```

As you see the support for schema evolution with Hive and Avro is very good.

What is Hive?

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.

Initially Hive was developed by Facebook, later the Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive. It is used by different companies. For example, Amazon uses it in Amazon Elastic MapReduce.

Hive is not

- A relational database
- A design for OnLine Transaction Processing (OLTP)
- A language for real-time queries and row-level updates

History of Hive and Recent Releases of Hive :

Hadoop is an open-source framework to store and process Big Data in a distributed environment. It contains two modules, one is MapReduce and another is Hadoop Distributed File System (HDFS).

- MapReduce: It is a parallel programming model for processing large amounts of structured, semi-structured, and unstructured data on large clusters of commodity hardware.

- HDFS:Hadoop Distributed File System is a part of Hadoop framework, used to store and process the datasets. It provides a fault-tolerant file system to run on commodity hardware.

The Hadoop ecosystem contains different sub-projects (tools) such as Sqoop, Pig, and Hive that are used to help Hadoop modules.

- Sqoop: It is used to import and export data to and from between HDFS and RDBMS.
- Pig: It is a procedural language platform used to develop a script for MapReduce operations.
- Hive: It is a platform used to develop SQL type scripts to do MapReduce operations.

Note: There are various ways to execute MapReduce operations:

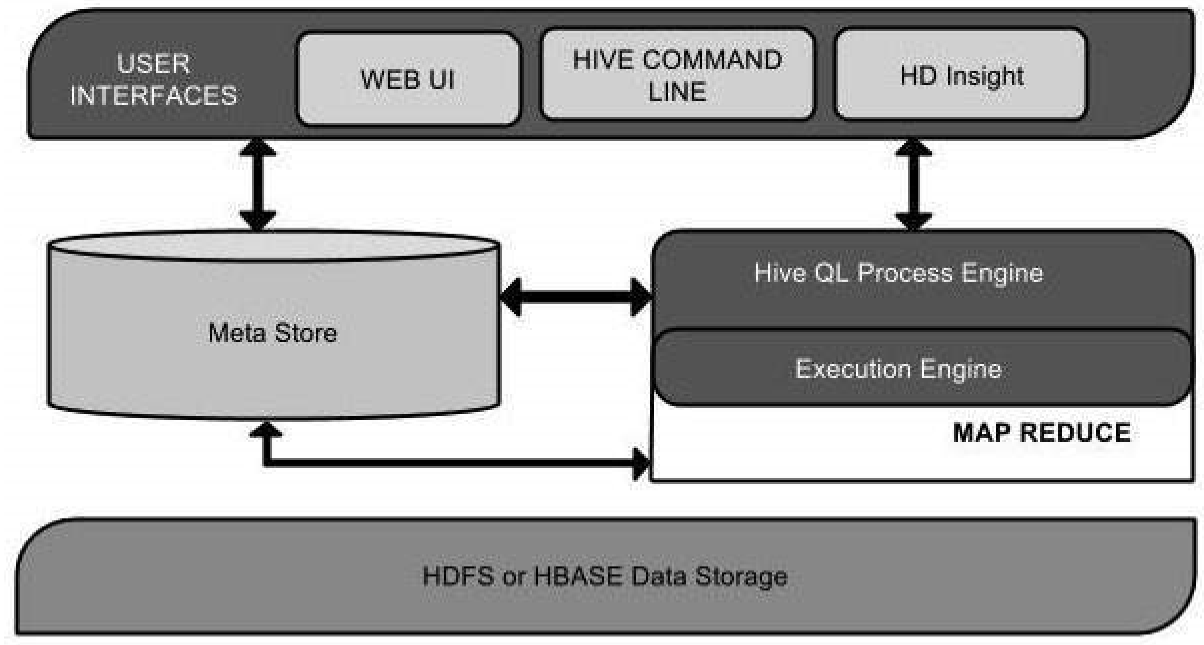
- The traditional approach using Java MapReduce program for structured, semi-structured, and unstructured data.
- The scripting approach for MapReduce to process structured and semi structured data using Pig.
- The Hive Query Language (HiveQL or HQL) for MapReduce to process structured data using Hive.

Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP.
- It provides SQL type language for querying called HiveQL or HQL.
- It is familiar, fast, scalable, and extensible.

Hive Integration and Work Flow & its Architecture :

The following component diagram depicts the architecture of Hive:



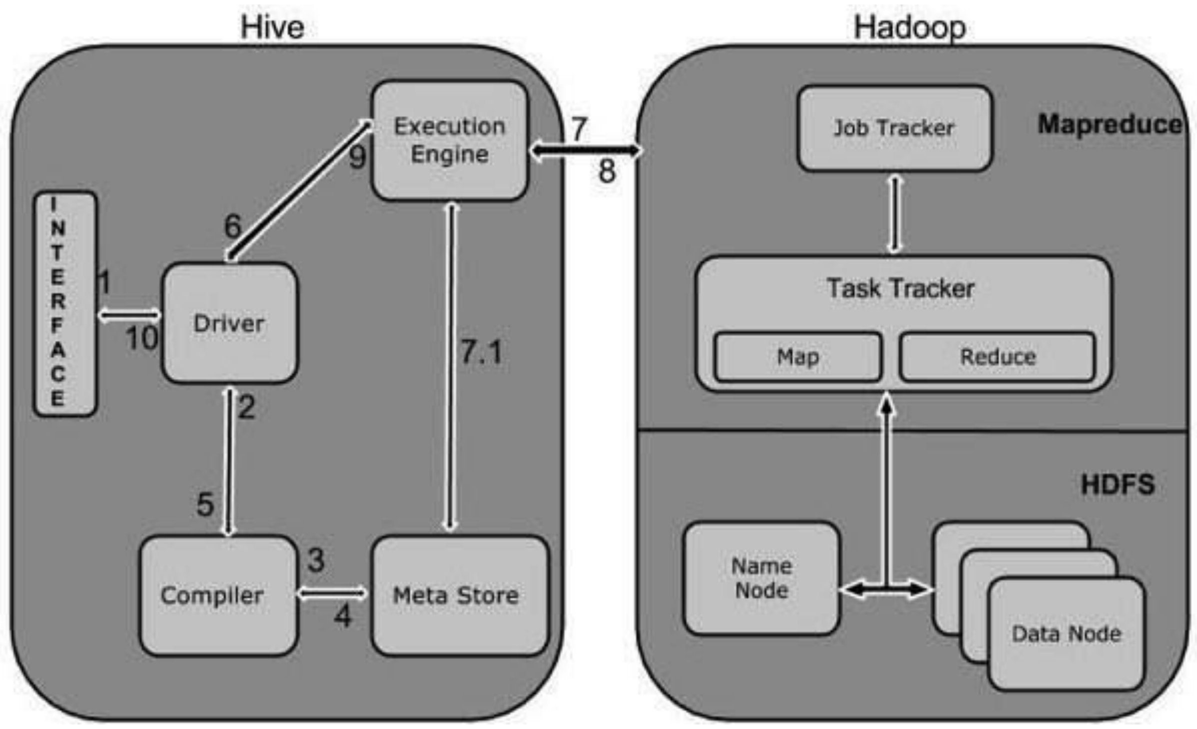
This component diagram contains different units. The following table describes each unit:

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution	The conjunction part of HiveQL process Engine and MapReduce

Engine	is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results. It uses the flavor of MapReduce.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive

The following diagram depicts the workflow between Hive and Hadoop.



The following table defines how Hive interacts with Hadoop framework:

Step No.	Operation
1	Execute Query The Hive interface such as Command Line or Web UI sends query to Driver (any database driver such as JDBC, ODBC, etc.) to execute.
2	Get Plan The driver takes the help of query compiler that parses the query to check the syntax and query plan or the requirement of query.
3	Get Metadata The compiler sends metadata request to Metastore (any database).
4	Send Metadata Metastore sends metadata as a response to the compiler.
5	Send Plan The compiler checks the requirement and resends the plan to the driver. Up to here, the parsing and compiling of a query is complete.
6	Execute Plan The driver sends the execute plan to the execution engine.
7	Execute Job

	Internally, the process of execution job is a MapReduce job. The execution engine sends the job to JobTracker, which is in Name node and it assigns this job to TaskTracker, which is in Data node. Here, the query executes MapReduce job.
7.1	Metadata Ops Meanwhile in execution, the execution engine can execute metadata operations with Metastore.
8	Fetch Result The execution engine receives the results from Data nodes.
9	Send Results The execution engine sends those resultant values to the driver.
10	Send Results The driver sends the results to Hive Interfaces.

Hive Data Units :

This chapter takes you through the different data types in Hive, which are involved in the table creation. All the data types in Hive are classified into four types, given as follows:

- Column Types
- Literals
- Null Values
- Complex Types

Column Types

Column type are used as column data types of Hive. They are as follows:

Integral Types

Integer type data can be specified using integral data types, INT. When the data range exceeds the range of INT, you need to use BIGINT and if the data range is smaller than the INT, you use SMALLINT. TINYINT is smaller than SMALLINT.

The following table depicts various INT data types:

Type	Postfix	Example
TINYINT	Y	10Y
SMALLINT	S	10S
INT	-	10
BIGINT	L	10L

String Types

String type data types can be specified using single quotes (' ') or double quotes (" "). It contains two data types: VARCHAR and CHAR. Hive follows C-types escape characters.

The following table depicts various CHAR data types:

Data Type	Length
VARCHAR	1 to 65355
CHAR	255

Timestamp

It supports traditional UNIX timestamp with optional nanosecond precision. It supports java.sql.Timestamp format “YYYY-MM-DD HH:MM:SS.fffffffff” and format “yyyy-mm-dd hh:mm:ss.ffffffffff”.

Dates

DATE values are described in year/month/day format in the form {{YYYY-MM-DD}}.

Decimals

The DECIMAL type in Hive is as same as Big Decimal format of Java. It is used for representing immutable arbitrary precision. The syntax and example is as follows:

```
DECIMAL(precision, scale)
decimal(10,0)
```

Union Types

Union is a collection of heterogeneous data types. You can create an instance using create union. The syntax and example is as follows:

```
UNIONTYPE<int, double, array<string>, struct<a:int,b:string>>


{0:1}
{1:2.0}
{2:["three","four"]}
{3:{"a":5,"b":"five"}}
{2:["six","seven"]}
{3:{"a":8,"b":"eight"}}
{0:9}
{1:10.0}
```

Literals

The following literals are used in Hive:

Floating Point Types

Floating point types are nothing but numbers with decimal points. Generally, this type of data is composed of DOUBLE data type.

Decimal Type

Decimal type data is nothing but floating point value with higher range than DOUBLE data type. The range of decimal type is approximately -10^{-308} to 10^{308}.

Null Value

Missing values are represented by the special value NULL.

Complex Types

The Hive complex data types are as follows:

Arrays

Arrays in Hive are used the same way they are used in Java.

```
Syntax: ARRAY<data_type>
```

Maps

Maps in Hive are similar to Java Maps.

```
Syntax: MAP<primitive_type, data_type>
```

Structs

Structs in Hive is similar to using complex data with comment.

```
Syntax: STRUCT<col_name : data_type [COMMENT col_comment], ...>
```

Hive Primitive Data Types and Collection Types Hive File Formats:

HIVE Data Types

Hive data types are categorized in numeric types, string types, misc types and complex types. A list of Hive data types are given below.

Numeric Types

TINYINT (1-byte signed integer, from -128 to 127)

SMALLINT (2-byte signed integer, from -32,768 to 32,767)

INT (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)

BIGINT (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

FLOAT (4-byte single precision floating point number)

DOUBLE (8-byte double precision floating point number)

DECIMAL (Hive 0.13.0 introduced user definable precision and scale)

Date/Time Types

TIMESTAMP
DATE

String Types

STRING
VARCHAR
CHAR

Misc Types

BOOLEAN
BINARY

Complex Type

arrays: ARRAY<data_type>

maps: MAP<primitive_type, data_type>

structs: STRUCT<col_name : data_type [COMMENT col_comment], ...>

union: UNIONTYPE<data_type, data_type, ...>

Hive Query Language – Statements - DDL,DML

Create/Drop/Alter/Use Database

Create Database

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
  [COMMENT database_comment]
  [LOCATION hdfs_path]
  [WITH DBPROPERTIES (property_name=property_value, ...)];
```

The uses of SCHEMA and DATABASE are interchangeable – they mean the same thing. CREATE DATABASE was added in Hive 0.6 (HIVE-675). The WITH DBPROPERTIES clause was added in Hive 0.7 (HIVE-1836).

Drop Database

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT|CASCADE];
```

The uses of SCHEMA and DATABASE are interchangeable – they mean the same thing. DROP DATABASE was added in Hive 0.6 (HIVE-675). The default behavior is RESTRICT, where DROP DATABASE will fail if the database is not empty. To drop the tables in the database as well, use DROP DATABASE ... CASCADE. Support for RESTRICT and CASCADE was added in Hive 0.8 (HIVE-2090).

Alter Database

```
ALTER (DATABASE|SCHEMA) database_name SET DBPROPERTIES (property_name=proper

ALTER (DATABASE|SCHEMA) database_name SET OWNER [USER|ROLE] user_or_role;   --
```

The uses of SCHEMA and DATABASE are interchangeable – they mean the same thing. ALTER SCHEMA was added in Hive 0.14 (HIVE-6601).

No other metadata about a database can be changed.

Use Database

```
USE database_name;
USE DEFAULT;
```

USE sets the current database for all subsequent HiveQL statements. To revert to the default database, use the keyword "default" instead of a database name. To check which database is currently being used: SELECT current_database() (as of Hive 0.13.0).

USE database_name was added in Hive 0.6 (HIVE-675).

Create/Drop/Truncate Table

- Create Table

- Managed and External Tables
- Storage Formats
- Row Formats & SerDe
- Partitioned Tables
- External Tables
- Create Table As Select (CTAS)
- Create Table Like
- Bucketed Sorted Tables
- Skewed Tables
- Temporary Tables
- Constraints
- Drop Table
- Truncate Table

Create Table

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name    -- (No
  [(col_name data_type [COMMENT col_comment], ... [constraint_specification])]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
  [CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)] INTO
  [SKEWED BY (col_name, col_name, ...)                  -- (Note: Available in Hive 0.10.0 and late
     ON ((col_value, col_value, ...), (col_value, col_value, ...), ...)
     [STORED AS DIRECTORIES]
  [
   [ROW FORMAT row_format]
   [STORED AS file_format]
     | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]  -- (Note: Ava
  ]
  [LOCATION hdfs_path]
  [TBLPROPERTIES (property_name=property_value, ...)]   -- (Note: Available in Hive 0.6.0 an
  [AS select_statement];   -- (Note: Available in Hive 0.5.0 and later; not supported for externa

CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  LIKE existing_table_or_view_name
  [LOCATION hdfs_path];

data_type
  : primitive_type
  | array_type
  | map_type
  | struct_type
  | union_type  -- (Note: Available in Hive 0.7.0 and later)

primitive_type
  : TINYINT
  | SMALLINT
  | INT
  | BIGINT
  | BOOLEAN
  | FLOAT
  | DOUBLE
  | DOUBLE PRECISION -- (Note: Available in Hive 2.2.0 and later)
  | STRING
  | BINARY      -- (Note: Available in Hive 0.8.0 and later)
  | TIMESTAMP   -- (Note: Available in Hive 0.8.0 and later)
  | DECIMAL     -- (Note: Available in Hive 0.11.0 and later)
  | DECIMAL(precision, scale)  -- (Note: Available in Hive 0.13.0 and later)
  | DATE        -- (Note: Available in Hive 0.12.0 and later)
  | VARCHAR     -- (Note: Available in Hive 0.12.0 and later)
  | CHAR        -- (Note: Available in Hive 0.13.0 and later)

array_type
  : ARRAY < data_type >

map_type
  : MAP < primitive_type, data_type >

struct_type
  : STRUCT < col_name : data_type [COMMENT col_comment], ...>

union_type
   : UNIONTYPE < data_type, data_type, ... >  -- (Note: Available in Hive 0.7.0 and later)

row_format
  : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS T
        [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]
        [NULL DEFINED AS char]   -- (Note: Available in Hive 0.13 and later)
  | SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, propert

file_format:
  : SEQUENCEFILE
  | TEXTFILE    -- (Default, depending on hive.default.fileformat configuration)
  | RCFILE      -- (Note: Available in Hive 0.6.0 and later)
  | ORC         -- (Note: Available in Hive 0.11.0 and later)
  | PARQUET     -- (Note: Available in Hive 0.13.0 and later)
  | AVRO        -- (Note: Available in Hive 0.14.0 and later)
  | INPUTFORMAT input_format_classname OUTPUTFORMAT output_format_classname

constraint_specification:
  : [, PRIMARY KEY (col_name, ...) DISABLE NOVALIDATE ]
    [, CONSTRAINT constraint_name FOREIGN KEY (col_name, ...) REFERENCES table_name
```

Hive Partitions – Bucketing, Views, Sub Query, Joins, Hive User Defined Function

Hive organizes tables into partitions. It is a way of dividing a table into related parts based on the values of partitioned columns such as date, city, and department. Using partition, it is easy to query a portion of the data.

Tables or partitions are sub-divided into buckets, to provide extra structure to the data that may be used for more efficient querying. Bucketing works based on the value of hash function of some column of a table.

For example, a table named Tab1 contains employee data such as id, name, dept, and yoj (i.e., year of joining). Suppose you need to retrieve the details of all employees who joined in 2012. A query searches the whole table for the required information. However, if you partition the employee data with the year and store it in a separate file, it reduces the query processing time. The following example shows how to partition a file and its data:

The following file contains employeedata table.

/tab1/employeedata/file1

```
id, name, dept, yoj
1, gopal, TP, 2012
2, kiran, HR, 2012
3, kaleel,SC, 2013
4, Prasanth, SC, 2013
```

The above data is partitioned into two files using year.

/tab1/employeedata/2012/file2

```
1, gopal, TP, 2012
2, kiran, HR, 2012
```

/tab1/employeedata/2013/file3

```
3, kaleel,SC, 2013
```

```
4, Prasanth, SC, 2013
```

Adding a Partition

We can add partitions to a table by altering the table. Let us assume we have a table called employee with fields such as Id, Name, Salary, Designation, Dept, and yoj.

Syntax:

```
ALTER TABLE table_name ADD [IF NOT EXISTS] PARTITION partition_spec
[LOCATION 'location1'] partition_spec [LOCATION 'location2'] ...;


partition_spec:
: (p_column = p_col_value, p_column = p_col_value, ...)
```

The following query is used to add a partition to the employee table.

```
hive> ALTER TABLE employee
> ADD PARTITION (year='2013')
> location '/2012/part2012';
```

Renaming a Partition

The syntax of this command is as follows.

```
ALTER TABLE table_name PARTITION partition_spec RENAME TO PARTITION
partition_spec;
```

The following query is used to rename a partition:

```
hive> ALTER TABLE employee PARTITION (year='1203')
   > RENAME TO PARTITION (Yoj='1203');
```

Dropping a Partition

The following syntax is used to drop a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec,
PARTITION partition_spec,...;
```

The following query is used to drop a partition:

```
hive> ALTER TABLE employee DROP [IF EXISTS]
   > PARTITION (year='1203');
```

Aggregations in Hive

Return Type	Name(Signature)	Description
bigint	count(*), count(expr), count(DISTINCT expr[, expr_.])	count(*) - Returns the total number of retrieved rows, including rows containing NULL values; count(expr) - Returns the number of rows for which the supplied expression is non-NULL; count(DISTINCT expr[, expr]) - Returns the number of rows for which the supplied expression(s) are unique and non-NULL.
double	sum(col), sum(DISTINCT col)	Returns the sum of the elements in the group or the sum of the distinct values of the column in the group
double	avg(col), avg(DISTINCT col)	Returns the average of the elements in the group or the average of the distinct values of the column in the group
double	min(col)	Returns the minimum of the column in the group
double	max(col)	Returns the maximum value of the column in the group
double	variance(col), var_pop(col)	Returns the variance of a numeric column in the group

double	var_samp(col)	Returns the unbiased sample variance of a numeric column in the group
double	stddev_pop(col)	Returns the standard deviation of a numeric column in the group
double	stddev_samp(col)	Returns the unbiased sample standard deviation of a numeric column in the group
double	covar_pop(col1, col2)	Returns the population covariance of a pair of numeric columns in the group
double	covar_samp(col1, col2)	Returns the sample covariance of a pair of a numeric columns in the group
double	corr(col1, col2)	Returns the Pearson coefficient of correlation of a pair of a numeric columns in the group
double	percentile(BIGINT col, p)	Returns the exact p^{th} percentile of a column in the group (does not work with floating point types). p must be between 0 and 1. NOTE: A true percentile can only be computed for integer values. Use PERCENTILE_APPROX if your input is non-integral.
array<double>	percentile(BIGINT col, array(p_1 [, p_2]...))	Returns the exact percentiles p_1, p_2, ... of a column in the group (does not work with floating point types). p_i must be between 0 and 1. NOTE: A true percentile can only be computed for integer values. Use PERCENTILE_APPROX if your input is non-integral.

double	percentile_approx(DOUBLE col, p [, B])	Returns an approximate pth percentile of a numeric column (including floating point types) in the group. The B parameter controls approximation accuracy at the cost of memory. Higher values yield better approximations, and the default is 10,000. When the number of distinct values in col is smaller than B, this gives an exact percentile value.
array<double>	percentile_approx(DOUBLE col, array(p_1 [, p_2]...) [, B])	Same as above, but accepts and returns an array of percentile values instead of a single one.
array<struct {'x','y'}>	histogram_numeric(col, b)	Computes a histogram of a numeric column in the group using b non-uniformly spaced bins. The output is an array of size b of double-valued (x,y) coordinates that represent the bin centers and heights
array	collect_set(col)	Returns a set of objects with duplicate elements eliminated

Group by and Having :

Group By Clause

Group By clause is used for grouping the records of the database table(s).This clause creates a single row for each group and this process is called aggregation. To use group by clause we have to use at least one aggregate function in Select statement. We can use group by clause without where clause.

Syntax for Group By Clause

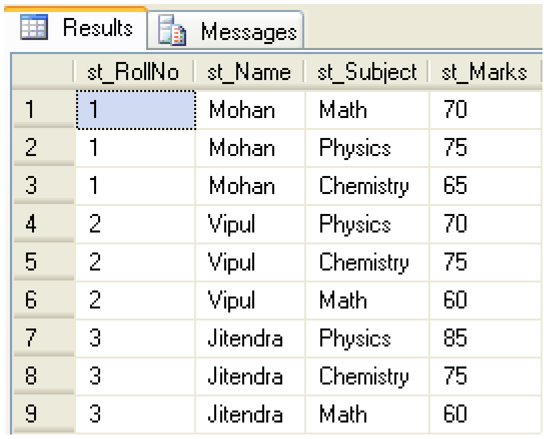
```
 SELECT Col1, Col2, Aggreate_function

FROM Table_Name
```

```
WHERE Condition

GROUP BY Col1, Col2
```

Let's see how the Group By clause works. Suppose we have a table StudentMarks that contains marks in each subject of the student.

```
 Create table StudentMarks

(

 st_RollNo int ,

 st_Name varchar(50),

 st_Subject varchar(50),

 st_Marks int

)

--Insert data in StudentMarks table

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)

values(1,'Mohan','Physics',75);

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)

values(1,'Mohan','Chemistry',65);

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)

values(1,'Mohan','Math',70);

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks) values(2,'Vip
ul','Physics',70);

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)

values(2,'Vipul','Chemistry',75);

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks) values(2,'Vip
ul','Math',60);
```

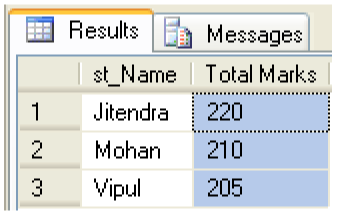
```
insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)

values(3,'Jitendra','Physics',85);

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)

values(3,'Jitendra','Chemistry',75);

insert into StudentMarks(st_RollNo,st_Name,st_Subject,st_Marks)

values(3,'Jitendra','Math',60);

--Now see data in table

select * from StudentMarks
```



Results Messages

	st_RollNo	st_Name	st_Subject	st_Marks
1	1	Mohan	Math	70
2	1	Mohan	Physics	75
3	1	Mohan	Chemistry	65
4	2	Vipul	Physics	70
5	2	Vipul	Chemistry	75
6	2	Vipul	Math	60
7	3	Jitendra	Physics	85
8	3	Jitendra	Chemistry	75
9	3	Jitendra	Math	60

```
 -- Group By clause without where condition

SELECT st_Name, SUM(st_Marks) AS 'Total Marks'

FROM StudentMarks

GROUP BY st_Name;
```

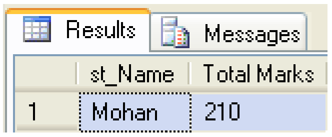


Results Messages

	st_Name	Total Marks
1	Jitendra	220
2	Mohan	210
3	Vipul	205

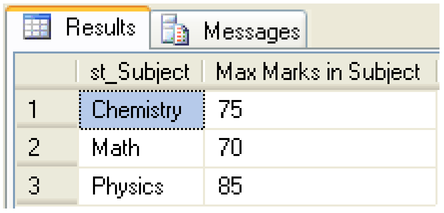
```
 -- Group By clause with where condition
```

```
SELECT st_Name, SUM(st_Marks) AS 'Total Marks'

FROM StudentMarks

where st_Name='Mohan'

GROUP BY st_Name;
```



```
   st_Name  Total Marks
1  Mohan    210
```

```
 -- Group By clause to find max marks in subject

SELECT st_Subject,max(st_Marks) AS 'Max Marks in Subject'

FROM StudentMarks

GROUP BY st_Subject;
```

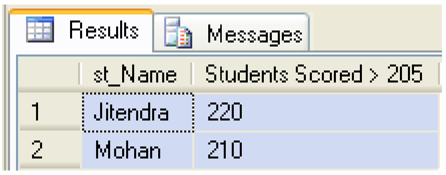


```
   st_Subject  Max Marks in Subject
1  Chemistry   75
2  Math        70
3  Physics     85
```

Having Clause

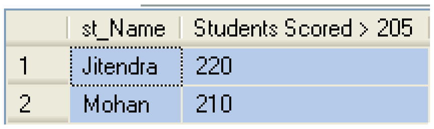
This clause operates only on group rows of table(s) and act as a filter like as where clause. We use having clause to filter data that we get from group by clause. To use having clause we need to use group by clause first.

```
 -- Having clause without where condition

SELECT st_Name, SUM(st_Marks) AS 'Students Scored > 205'

FROM StudentMarks

GROUP BY st_Name

HAVING SUM(st_Marks) > 205
```



	st_Name	Students Scored > 205
1	Jitendra	220
2	Mohan	210

```
-- Having clause with where condition

SELECT st_Name, SUM(st_Marks) AS 'Students Scored > 205'

FROM StudentMarks

where st_RollNo between 1 and 3

GROUP BY st_Name

HAVING SUM(st_Marks) > 205
```



	st_Name	Students Scored > 205
1	Jitendra	220
2	Mohan	210

Note

1. To use Group By Clause, we need to use at least one aggregate function
2. All columns that are not used by aggregate function(s) must be in the Group By list
3. We can use Group By Clause with or without Where Clause.
4. To use Having Clause, we have to use Group By Clause since it filters data that we get from Group By Clause

Serialization and Deserialization :

Name	Type	Parent	Implementing Interfaces	Detail
HCatRecordSerDe	Class	java.lang.Object	org.apache.hadoop.hive.serde2.Deserializer , org.apache.hadoop.hive.serde2.SerDe , org.apache.hadoop.hive.serde2.Serializer	Detail
HCatRecordSerDe	Class	java.lang.Object	org.apache.hadoop.hive.serde2.Deserializer , org.apache.hadoop.hive.serde2.SerDe , org.apache.hadoop.hive.serde2.Serializer	Detail

HCatRecordSerDe	Class	java.lang.Object	org.apache.hadoop.hive.serde2.Deserializer , org.apache.hadoop.hive.serde2.SerDe , org.apache.hadoop.hive.serde2.Serializer	Detail
JsonSerDe	Class	java.lang.Object	org.apache.hadoop.hive.serde2.Deserializer , org.apache.hadoop.hive.serde2.SerDe , org.apache.hadoop.hive.serde2.Serializer	Detail
JsonSerDe	Class	java.lang.Object	org.apache.hadoop.hive.serde2.Deserializer , org.apache.hadoop.hive.serde2.SerDe , org.apache.hadoop.hive.serde2.Serializer	Detail
JsonSerDe	Class	java.lang.Object	org.apache.hadoop.hive.serde2.Deserializer , org.apache.hadoop.hive.serde2.SerDe , org.apache.hadoop.hive.serde2.Serializer	Detail

Hive Analytic Functions :

This section introduces the Hive QL enhancements for windowing and analytics functions. See "Windowing Specifications in HQL" (attached to HIVE-4197) for details. HIVE-896 has more information, including links to earlier documentation in the initial comments.

All of the windowing and analytics functions operate as per the SQL standard.

The current release supports the following functions for windowing and analytics:

1. Windowing functions

- LEAD
- The number of rows to lead can optionally be specified. If the number of rows to lead is not specified, the lead is one row.
- Returns null when the lead for the current row extends beyond the end of the window.
- LAG
- The number of rows to lag can optionally be specified. If the number of rows to lag is not specified, the lag is one row.
- Returns null when the lag for the current row extends before the beginning of the window.
- FIRST_VALUE

- LAST_VALUE

2. The OVER clause

- OVER with standard aggregates:
- COUNT
- SUM
- MIN
- MAX
- AVG
- OVER with a PARTITION BY statement with one or more partitioning columns of any primitive datatype.
- OVER with PARTITION BY and ORDER BY with one or more partitioning and/or ordering columns of any datatype.
- OVER with a window specification. Windows can be defined separately in a WINDOW clause. Window specifications support the following formats:
- (ROWS | RANGE) BETWEEN (UNBOUNDED | [num]) PRECEDING AND ([num] PRECEDING | CURRENT ROW | (UNBOUNDED | [num]) FOLLOWING)
- (ROWS | RANGE) BETWEEN CURRENT ROW AND (CURRENT ROW | (UNBOUNDED | [num]) FOLLOWING)
  (ROWS | RANGE) BETWEEN [num] FOLLOWING AND (UNBOUNDED | [num]) FOLLOWING

When ORDER BY is specified with missing WINDOW clause, the WINDOW specification defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.