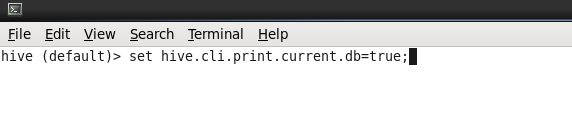
**set hive.cli.print.current.db=true;**

This is used to show which database we are using presently.



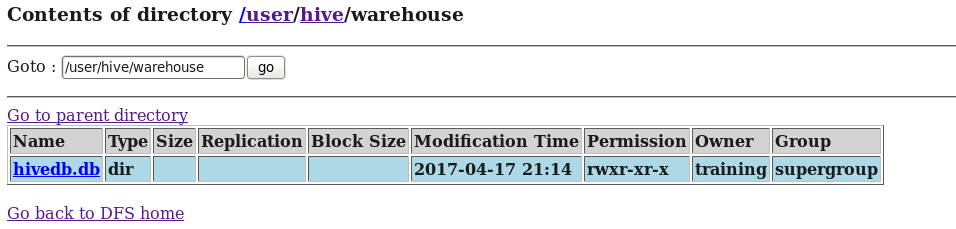
1. Creating Database:

Wherever a database is created, by default it is stored as directory in HDFS in path: /user/hive/warehouse.

**hive> create database HIVEDB;**

**OK**

**Time taken: 0.222 seconds**



1. Creating a Internal table:

CREATE TABLE IF NOT EXISTS EMPLOYEE\_INTERNAL (eid int, name String,

salary String, destination String)

row format delimited

fields terminated by ‘,’;

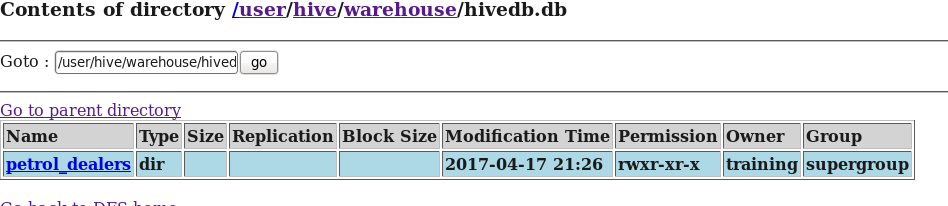
OK

Time taken: 2.057 seconds

1. LOADING data from Local:

hive (hive)> load data local inpath '/home/training/Desktop/Hive/employee' into table employee;

By default internal tables are stored in path : **/user/hive/warehouse/hivedb.db**

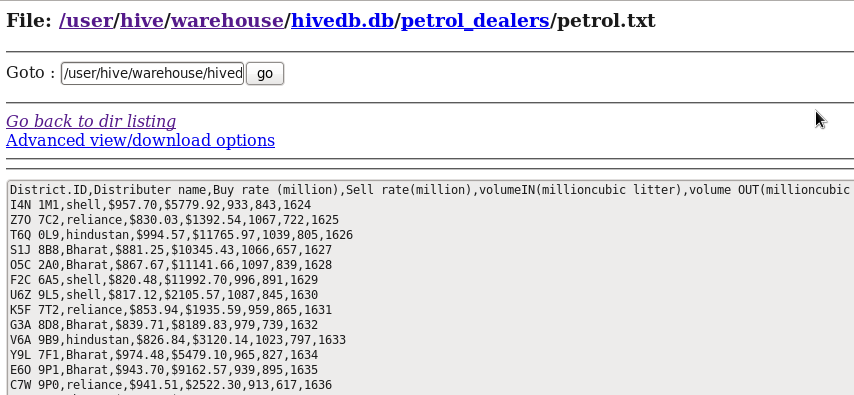


1. Loading data into table from HDFS:

First make sure the data is in HDFS path.

hadoop fs -put '/home/training/Desktop/Hive/employee.txt' /home/training/Desktop/Hive;

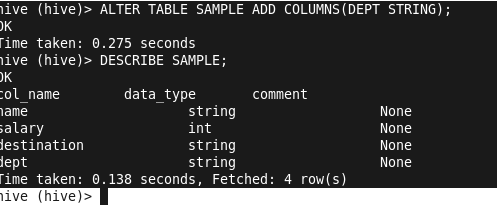
**hive> load data inpath '/home/training/HIVE/petrol.txt' into table petrol\_dealers;**



**Alter Table Statement**

1. Adding more column to an existing table:

ALTER TABLE SAMPLE ADD COLUMNS(DEPT STRING);



## Replace Statement

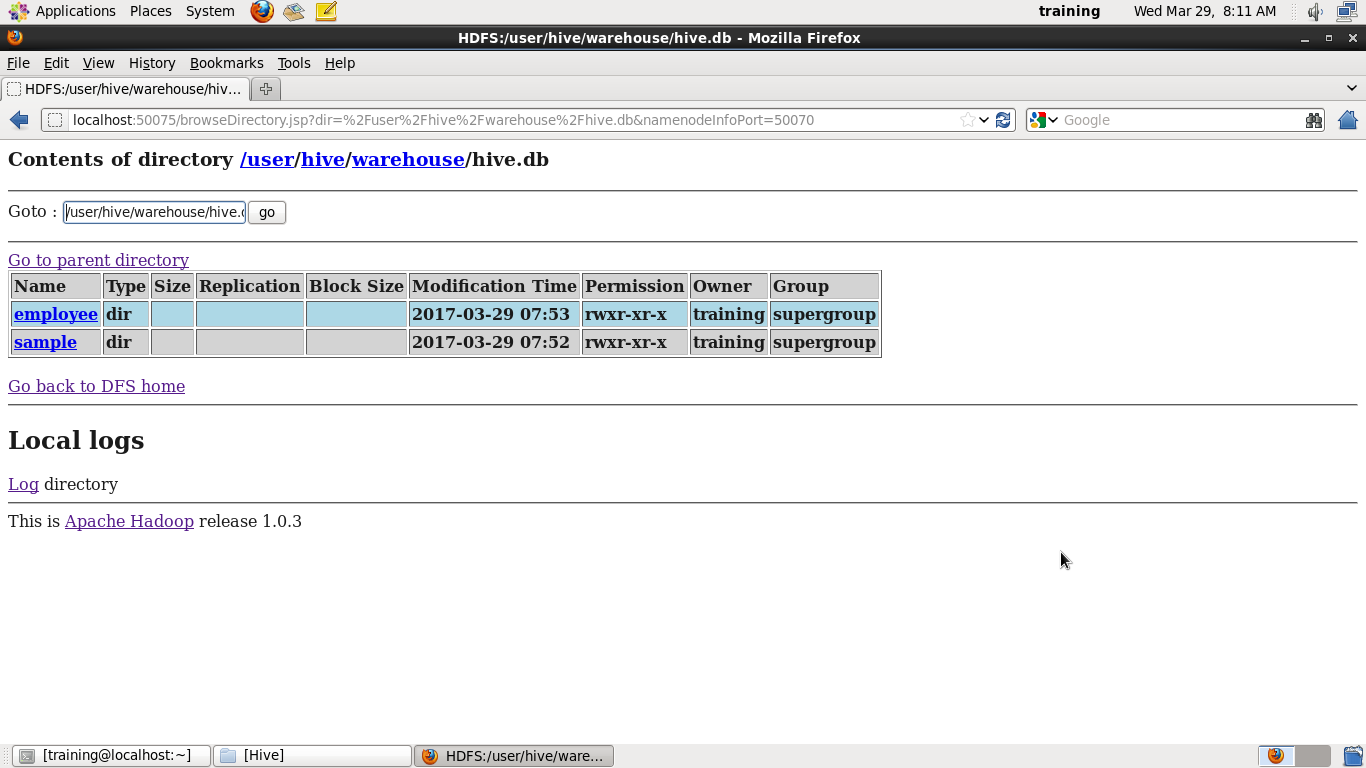
ALTER TABLE employee REPLACE COLUMNS (

eid INT empid Int,

ename STRING name String);

Actually you can check all the tables that you have created in HDFS:

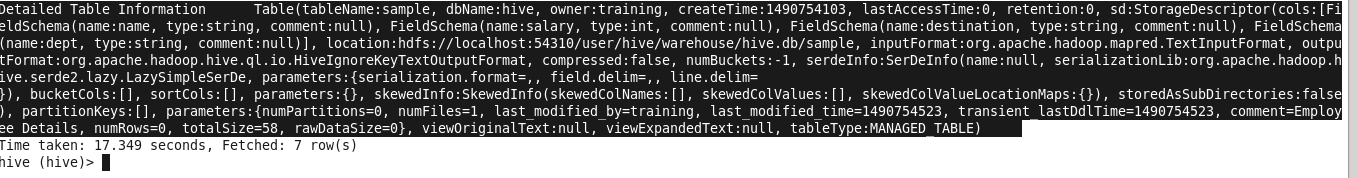
Path: /user/hive/warehouse/hive.db



All the data stored in this path because we have mentioned this path in hive conf file - hive-site.xml

To know complete information about a created table:

**describe extended sample;**



Detailed Table Information Table(tableName:sample, dbName:hive, owner:training, createTime:1490754103, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:[FieldSchema(name:name, type:string, comment:null), FieldSchema(name:salary, type:int, comment:null), FieldSchema(name:destination, type:string, comment:null), FieldSchema(name:dept, type:string, comment:null)], **location:hdfs://localhost:54310/user/hive/warehouse/hive.db/sample,** inputFormat:org.apache.hadoop.mapred.TextInputFormat, outputFormat:org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null, serializationLib:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, parameters:{serialization.format=,, field.delim=,, line.delim=

}), bucketCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNames:[], skewedColValues:[], skewedColValueLocationMaps:{}), storedAsSubDirectories:false), partitionKeys:[], parameters:{numPartitions=0, numFiles=1, last\_modified\_by=training, last\_modified\_time=1490754523, transient\_lastDdlTime=1490754523, comment=Employee Details, numRows=0, totalSize=58, rawDataSize=0}, viewOriginalText:null, viewExpandedText:null, tableType:MANAGED\_TABLE)

Time taken: 17.349 seconds, Fetched: 7 row(s)

What happens when the table is created in hive?

When a table is created in hive, it is stored in the form of ***direcory*** in HDFS. And it can be viewed using:

**hadoop fs -ls /user/hive/warehouse/hive.db/**



To know no of partitions that a table have?

Show partitions cca\_dynamic\_partition\_multiple;

To see table create statement:

show create table cca\_dynamic\_partition\_multiple;

To delete dabase even if it have tables in it

Drop DATABASE db\_name CASCADE;

To Rename a table:

ALTER TABLE table\_name RENAME to table\_name;

**To change column name:**

ALTER TABLE emp

CHANGE COLUMN emp\_name new\_emp\_name STRING

COMMENT ‘EMPLOYEE NAME’

AFTER unit;

Hive – Views :

Suppose if I want to run any complex queries again and again then we can create VIWES for those queries.

CREATE VIEW emp\_view AS select emp\_id, name from EMP\_DETAILS where emp\_id> 1000;

**Copying / Storing table contents into local or different location in HDFS :**

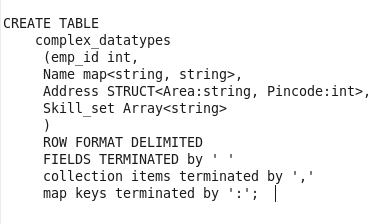
INSERT OVERWRITE LOCAL DIRECTORY ‘ ---path --------‘

ROW FORMAT DELIMITED

FIELDS TERMINATED by ‘,’

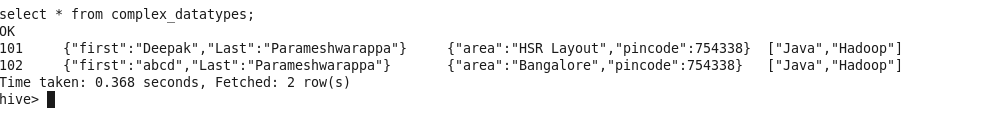
Select \* from emp\_details;

Hive complex data types

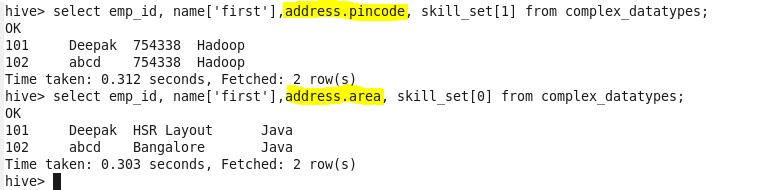


**LOAD the below data :**





**The struct type – address elements has to be accesses by . unlike others like arrays and maps**



Hive date functions:

Date data types do not exist in Hive. In fact the dates are treated as strings in Hive. The date functions are listed below.

**select \* from emp;**

**1 asdc**

**2 Deepak**

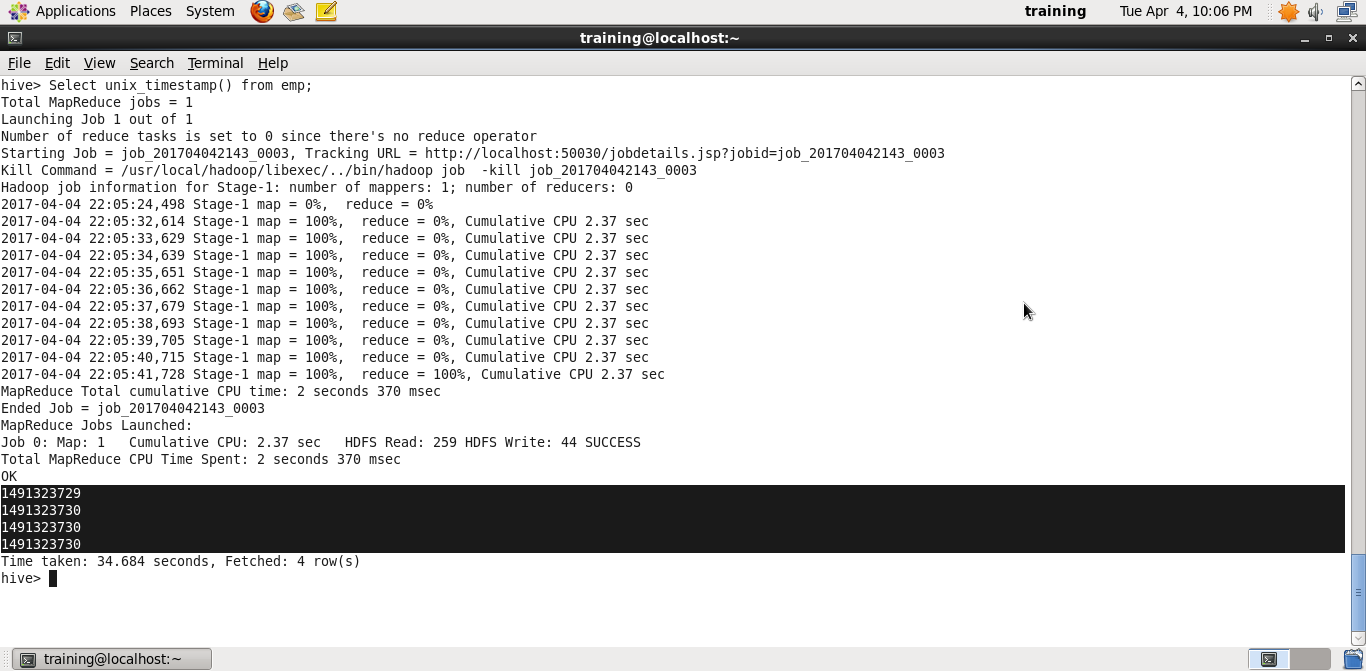
**3 qwwr**

**4 sfhihy**

1. **UNIX\_TIMESTAMP()**

This function returns the number of seconds from the Unix epoch (1970-01-01 00:00:00 UTC) using the default time zone.

Select unix\_timestamp() from emp;



1. **UNIX\_TIMESTAMP( string date )**

This function converts the date in format 'yyyy-MM-dd HH:mm:ss' into Unix timestamp. This will return the number of seconds between the specified date and the Unix epoch. If it fails, then it returns 0.

1. **UNIX\_TIMESTAMP( string date, string pattern )**

When to use dynamic partitioning?

The values of dynamic partition columns are known only at execution time of the data-load query. Hive automatically takes care of not only creating the partitions, but also loading the data into the correct partitions. No manual user intervention is required. Partition columns are specified by a “PARTITIONED BY” clause in “CREATE TABLE” query. For each distinct set of partition column values, Hive creates a unique HDFS path and loads data into it.

Table creation semantics are the same for both static and dynamic partitioning. With static partitioning, partitions are explicitly added or dropped by the user, using “ALTER TABLE ADD/DROP PARTITION …” queries. This updates the Hive metastore with table partition information. With dynamic partitioning, since partition values are not known in advance, the user does not need to perform the explicit “alter table” step. Hive automatically takes care of updating the Hive metastore when using dynamic partitions.

**Loading from an existing table that is not partitioned:**

 In this scenario, the user doesn’t employ partitioning initially because the table in question is expected to remain relatively small. However, over the course of time, the table grows quite large and performance issues begin to appear. These issues should be corrected using a one-time load to dynamically partition the table.

**Unknown values for partition columns:** In some scenarios, it’s very difficult to know the unique values of all partition columns unless the data is manually inspected. As you can imagine, manual inspection isn’t a realistic option when dealing with batch pipelines or terabytes of data. You can try writing a Hive query to retrieve all unique value sets of partition columns. Let’s say, the query result contains many unique value sets. You’ll end up creating and executing an “ALTER TABLE” statement for each unique value set. Running a Hive query, preparing alter table statements, and executing them will significantly delay your data pipelines. Also, it’s an error-prone and cumbersome process. Hence, static partitioning is not used under these circumstances. However, dynamic partitioning can come to your rescue if you’d like to offload this work to Hive. Hive can detect the unique value sets for partition columns and create partitions on-the-fly.

Imagine you have a very big table with data accumulated over many years. You want to improve the performance by partitioning the data. To complicate the scenario further, imagine you decide to not only partition the data by year, but also by month, day, hour, and advertiser id (let’s assume your data has an advertiser\_id column). In this particular case, dynamic partitioning is very helpful. Why? You may be running a pipeline that ingests data hourly. You know exactly which year, month, day, and hour the data belongs to, but you can’t assume or tell which advertiser\_id each record contains unless you manually inspect the data. Hive can automatically partition the data on all the required columns if dynamic partitioning is used.

Next, let’s look at what can happen with dynamic partitioning when the data is skewed. The data that belongs to one partition is sent to only one reducer. However, consider a scenario where 90 percent of the data belongs to only one partition and the rest is spread across multiple partitions. In this situation, one reducer will be heavily loaded, while all other reducers have finished their work. The time required to finish the job will depend solely on the longest running reducer. This will significantly increase the data load time.

To overcome this problem, we suggest our customers perform some queries on the data to check how evenly it’s distributed. The key idea is to distribute the data evenly across the reducers. To achieve even distribution, the table can be further divided by buckets or new partition columns. The solution will vary depending on the use case and the nature of the data. In one recent example, we worked with one of our customers to address their performance issues. After gaining a deeper understanding of their use case and data, we determined that the data was skewed and suggested bucketing the data by another column to evenly spread the data load across the reducers. This resolved the problem and significantly reduced their data load times.

Create table –as:

This will create a table and copies both table schema and table contents

Create table petrol\_as location ‘/usr/hive/warehouse’ as select \* from petrol;

Create table – like:

This will create a table and copies only table schema and not table contents.

**Analytic and Windowing in hive**

<https://www.cloudera.com/documentation/enterprise/5-8-x/topics/impala_analytic_functions.html#over>

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+WindowingAndAnalytics>

Rank – row num : <https://blog.jooq.org/2014/08/12/the-difference-between-row_number-rank-and-dense_rank/>

Analytic functions (also known as window functions) are a special category of built-in functions. Like aggregate functions, they examine the contents of multiple input rows to compute each output value. However, rather than being limited to one result value per GROUP BY group, they operate on **windows** where the input rows are ordered and grouped using flexible conditions expressed through an **OVER()** clause.

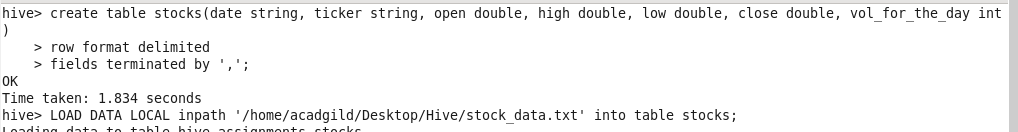
Although analytic functions often compute the same value you would see from an aggregate function in a GROUP BY query, the analytic functions produce a value for each row in the result set rather than a single value for each group. **This flexibility lets you include additional columns in the SELECT list, offering more opportunities for organizing and filtering the result set.**

Analytic function calls are only allowed in the SELECT list and in the outermost ORDER BY clause of the query. During query processing, analytic functions are evaluated after other query stages such as joins, WHERE, and GROUP BY,

<https://acadgild.com/blog/windowing-functions-in-hive/>

Create a table stocks and load data.

create table stocks (date\_ String, Ticker String, Open Double, High Double, Low Double, Close Double, Volume\_for\_the\_day int) row format delimited fields terminated by ',';



## OVER Clause

Syntax:

**function(*args*)**OVER([*partition\_by\_clause*] [*order\_by\_clause* [*window\_clause*]])

1. **Rank**( )

The rank function will return the rank of the values as per the result set of the over clause. If two values are same then it will give the same rank to those 2 values and then for the next value, the sub-sequent rank will be skipped.

The below query will rank the closing prices of the stock for each ticker

Rank() skips intermediate numbers in case of a tie.

--

Select rank() over(order by salary), id, name, salary, unit from users ;

Rank id

1 1 Deepak 100 DNA

1 5 Rama 100 FCS --------- both have sal 100.

3 2 Raja 200 DNA --------- intermediate no is skipped

3 4 Ranga 200 FCS

5 3 Yadav 500 DNA -------- intermediate no is skipped

1. **Dense\_Rank( )**

Doesn’t skips intermediate numbers in case of a tie.

1 1 Deepak 100 DNA

1 5 Rama 100 FCS --------- both have sal 100.

2 2 Raja 200 DNA --------- intermediate no is not skipped

3 4 Ranga 200 FCS

4 3 Yadav 500 DNA

Transactions in Hive

ACID properties, which are vital for any transaction.

* Atomicity means, a transaction should complete successfully or else it should fail completely i.e. it should not be left partially.
* Consistency ensures that any transaction will bring the database from one valid state to another state
* Isolation states that every transaction should be independent of each other i.e. one transaction should not affect another
* Durability states that if a transaction is completed, it should be preserved in the database even if the machine state is lost or a system failure might occur

ORC is the file format supported by Hive transaction. It is now essential to have ORC file format for performing transactions in Hive. The table needs to be bucketed in order to support transactions.

The below properties needs to be set appropriately in ***hive shell***, order-wise to work with transactions in Hive:

**hive>set hive.support.concurrency = true;**

**hive>set hive.enforce.bucketing = true;**

**hive>set hive.exec.dynamic.partition.mode = nonstrict;**

**hive>set hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;**

**hive>set hive.compactor.initiator.on = true;**

**hive>set hive.compactor.worker.threads = a positive number on at least one instance of the Thrift metastore service;**

If the above properties are not set properly, the ‘Insert’ operation will work but ‘Update’ and ‘Delete’ will not work and you will receive the following error:

**FAILED: SemanticException [Error 10294]: Attempt to do update or delete usingtransaction manager thatdoes not support these operations.**

## Creating a Table That Supports Hive Transactions