**HIVE DATA DEFINATIONS**

**Hive database**

The database in Hive describes a collection of tables that are used for a similar purpose or belong to the same groups. If the database is not specified, the default database is used. Whenever a new database is created, Hive creates a directory for each database at /user/hive/warehouse, defined in hive.metastore.warehouse.dir. For example, the myhivebook database is located at /user/hive/datawarehouse/myhivebook.db. However, the default database doesn't have its own directory. The following is the core DDL for Hive databases:

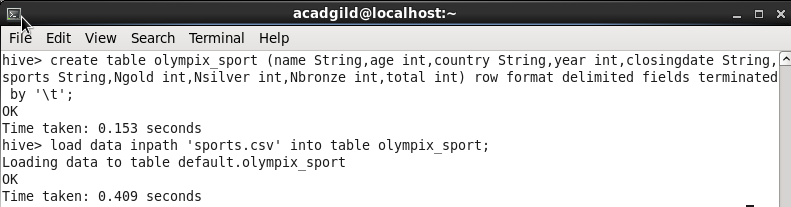
Create the database and check whether the database already exists:

Hive>**CREATE DATABASE IF NOT EXISTS myhivebook;**

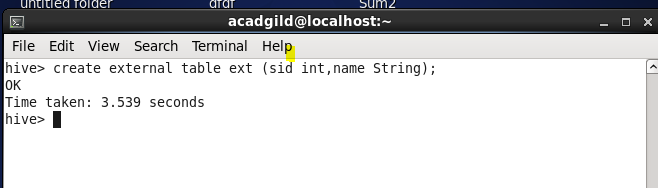
**Hive internal and external tables**

The concept of a table in Hive is very similar to the table in the relational database. Each table associates with a directory configured in ${HIVE\_HOME}/conf/hive-site.xml in HDFS. By default, it is /user/hive/warehouse in HDFS. For example, /user/hive/warehouse/employee is created by Hive in HDFS for the employee table. All the data in the table will be kept in the directory. The Hive table is also referred to as internal or managed tables.

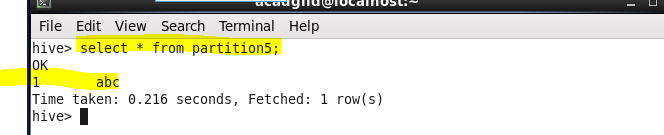
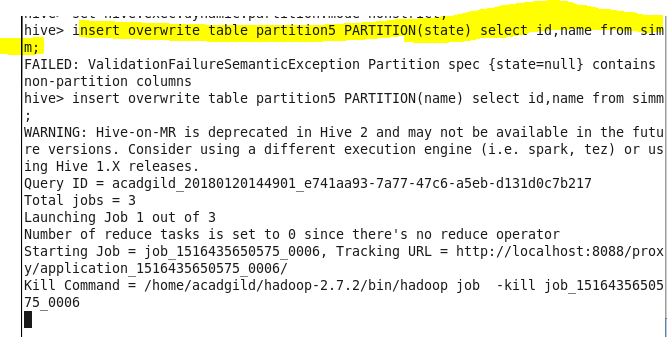
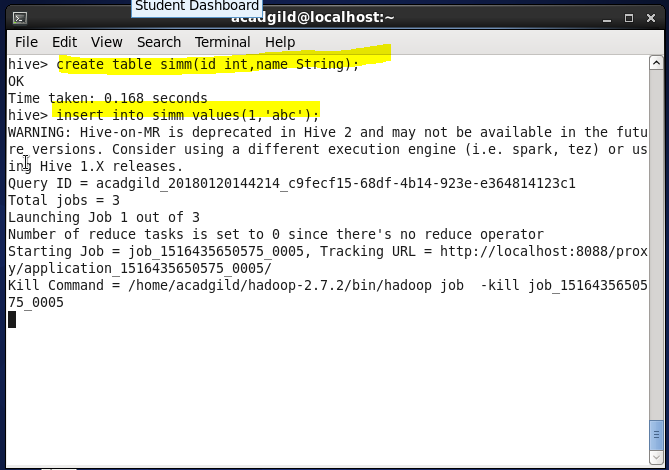
When there is data already in HDFS, an external Hive table can be created to describe the data. It is called EXTERNAL because the data in the external table is specified in the LOCATION properties instead of the default warehouse directory. When keeping data in the internal tables, Hive fully manages the life cycle of the table and data. This means the data is removed once the internal table is dropped. If the external table is dropped, the table metadata is deleted but the data is kept. Most of the time, an external table is preferred to avoid deleting data along with tables by mistake. The following are DDLs for Hive internal and external table examples:

Internal table: 

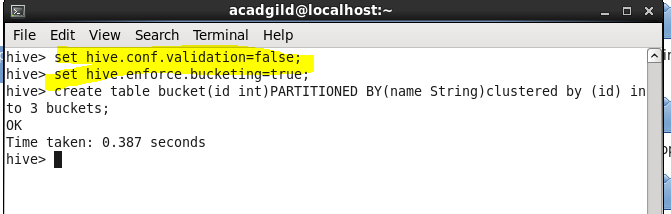
External table:



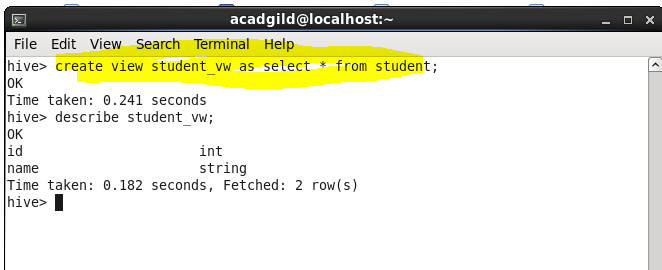
**Hive partitions**

By default, a simple query in Hive scans the whole Hive table. This slows down the performance when querying a large-size table. The issue could be resolved by creating Hive partitions, which is very similar to what's in the RDBMS. In Hive, each partition corresponds to a predefined partition column(s) and stores it as a subdirectory in the table's directory in HDFS. When the table gets queried, only the required partitions (directory) of data in the table are queried, so the I/O and time of query is greatly reduced. It is very easy to implement Hive partitions when the table is created and check the partitions created, as follows: 

**Hive buckets**

Besides partition, bucket is another technique to cluster datasets into more manageable parts to optimize query performance. Different from partition, the bucket corresponds to segments of files in HDFS. For example, the employee\_partitioned table from the previous section uses the year and month as the top-level partition. If there is a further request to use the employee\_id as the third level of partition, it leads to many deep and small partitions and directories. For instance, we can bucket the employee\_partitioned table using employee\_id as the bucket column. The value of this column will be hashed by a user-defined number into buckets. The records with the same employee\_id will always be stored in the same bucket (segment of files). By using buckets, Hive can easily and efficiently do sampling 

**Hive views**

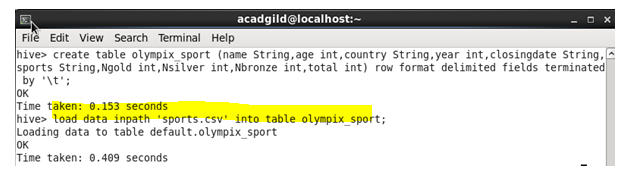
In Hive, views are logical data structures that can be used to simplify queries by either hiding the complexities such as joins, subqueries, and filters or by flatting the data. Unlike some RDBMS, Hive views do not store data or get materialized. Once the Hive view is created, its schema is frozen immediately. Subsequent changes to the underlying tables (for example, adding a column) will not be reflected in the view's schema. If an underlying table is dropped or changed, subsequent attempts to query the invalid view will fail, as follows: 

**HIVE DATA manipulations and HIVEQL manipulations**

**Loading files into tables**

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 that corresponds to Hive tables. If the table is partitioned, then data is loaded into partitions one at a time. The general syntax of loading the data into a table is as follows:

LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]

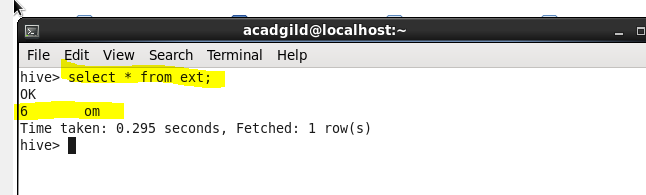
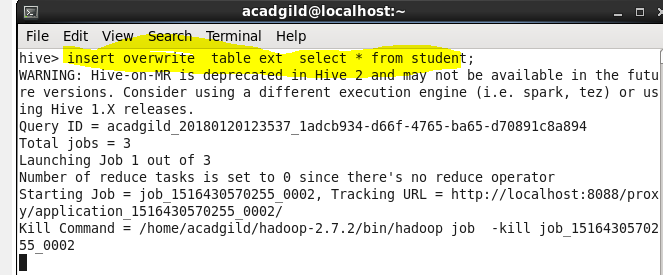
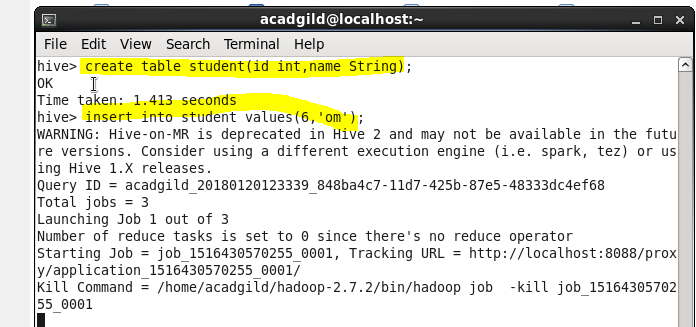


**Inserting data into Hive tables from queries**

In this recipe, you will learn how to insert data through queries into a table in Hive.

This is another variant of inserting data into a Hive table. Data can be appended into a Hive table that already contains data. Data can also be overwritten in the Hive table. Data can also be inserted into multiple tables through a single statement only. The general format of inserting data into a table from queries is as follows:

INSERT OVERWRITE TABLE tablename [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select select\_statement FROM from\_statement;



**Inserting data into dynamic partitions**

To insert data into multiple partitions through a single statement. The general syntax of inserting data into multiple partitions is as follows:

FROM tablename

INSERT OVERWRITE TABLE tablename1 PARTITION(root\_partition\_name='value',child\_partition\_name)

SELECT select\_statment;

**Writing data into files from queries**

Insert data into a file with the help of a query; that is, the output of a query to be saved into a file. The general format of inserting data into a file is as follows:

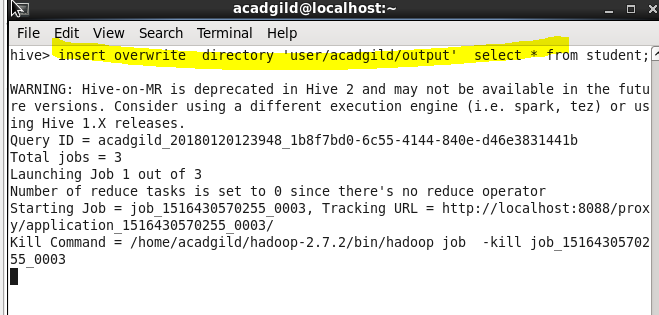
Standard syntax:

INSERT OVERWRITE [LOCAL] DIRECTORY directory1 [ROW FORMAT row\_format] [STORED AS file\_format]SELECT select\_statment FROM from\_statment.

Hive extension (multiple inserts):

FROM from\_statement

INSERT OVERWRITE [LOCAL] DIRECTORY directory1 select\_statement1

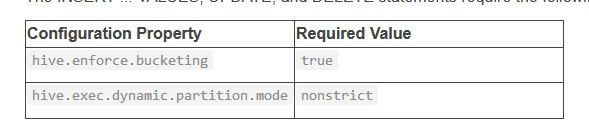
[INSERT OVERWRITE [LOCAL] DIRECTORY directory2 select\_statement2] ... 

**Enabling transactions in Hive**

Insert, Update and Delete are not possible in Hive until the ACID properties are not enabled. Also table must to be Bucketed in Hive if Insert, Update and Delete feature are to be used.

Transactions including inserts, updates, and deletes are available from Hive 1.0.0 and above.

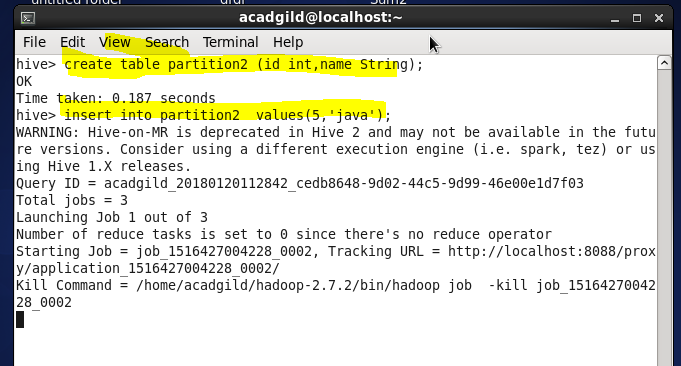
o allow the user to execute transactional commands, the user needs to configure the metastore with transactional tables. The user needs to set the following properties in hive-site.xml:

 ...

**Inserting values into tables from SQL**

Inserting data into a Hive table through a SQL statement is the third variant of inserting data. This is the traditional way of inserting data into a table in any RDBMS. Inserting in a table through SQL statements can only be performed if the table supports ACID. The general format of inserting data into a table is as follows:

INSERT INTO TABLE table\_name [PARTITION (partcol1[=val1], partcol2[=val2] ...)] VALUES values\_row [, values\_row ...]



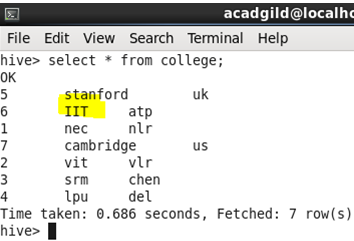
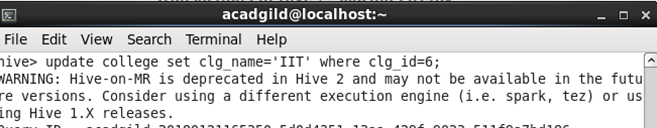
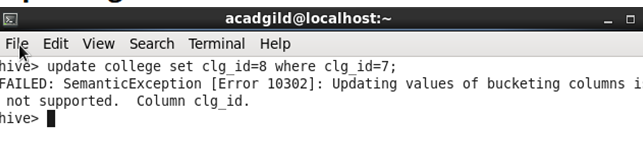
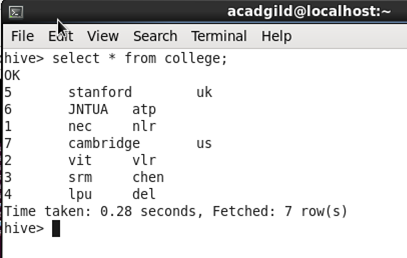
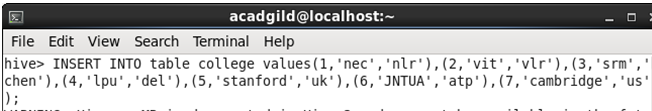
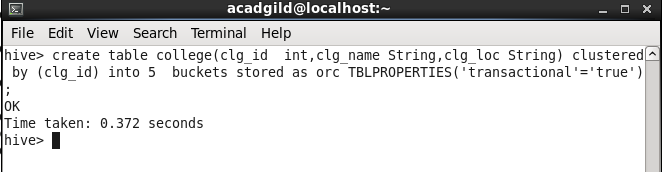
**Updating data**

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 **Atomicity**, **Consistency**, **Isolation**, **Durability** (**ACID**) properties. The general format of updating data in a table is as follows

This recipe requires having Hive installed. You will also need the Hive CLI or Beeline client to run the commands.

This recipe requires transactions enabled, so refer to *Enabling transactions in Hive* in Hive for that.

The following example demonstrates the correct usage of this statement:



**Deleting data**

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.

DELETE FROM tablename [WHERE expression] 