# HiveQL: Data Definition

HiveQL is the Hive query language. Like all SQL dialects in widespread use, it doesn’t fully conform to any particular revision of the ANSI SQL standard. It is perhaps closest to MySQL’s dialect, but with significant differences. Hive offers no support for row-level inserts, updates, and deletes. Hive doesn’t support transactions. Hive adds extensions to provide better performance in the context of Hadoop and to integrate with custom extensions and even external programs.

Still, much of HiveQL will be familiar. This chapter and the ones that follow discuss the features of HiveQL using representative examples. In some cases, we will briefly mention details for completeness, then explore them more fully in later chapters.

This chapter starts with the so-called data definition language parts of HiveQL, which are used for creating, altering, and dropping databases, tables, views, functions, and indexes. We’ll discuss databases and tables in this chapter, deferring the discussion of views until [Chapter 7](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch07.html), indexes until [Chapter 8](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch08.html), and functions until [Chapter 13](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch13.html).

We’ll also discuss the SHOW and DESCRIBE commands for listing and describing items as we go.

Subsequent chapters explore the data manipulation language parts of HiveQL that are used to put data into Hive tables and to extract data to the filesystem, and how to explore and manipulate data with queries, grouping, filtering, joining, etc.

# Databases in Hive

The Hive concept of a database is essentially just a catalog or namespace of tables. However, they are very useful for larger clusters with multiple teams and users, as a way of avoiding table name collisions. It’s also common to use databases to organize production tables into logical groups.

If you don’t specify a database, the default database is used.

The simplest syntax for creating a database is shown in the following example:

hive> **CREATE** **DATABASE** financials;

Hive will throw an error if financials already exists. You can suppress these warnings with this variation:

hive> **CREATE** **DATABASE** IF **NOT** **EXISTS** financials;

While normally you might like to be warned if a database of the same name already exists, the IF NOT EXISTS clause is useful for scripts that should create a database on-the-fly, if necessary, before proceeding.

You can also use the keyword SCHEMA instead of DATABASE in all the database-related commands.

At any time, you can see the databases that already exist as follows:

hive> **SHOW** DATABASES;

**default**

financials

hive> **CREATE** **DATABASE** human\_resources;

hive> **SHOW** DATABASES;

**default**

financials

human\_resources

If you have a lot of databases, you can restrict the ones listed using a regular expression, a concept we’ll explain in [LIKE and RLIKE](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch06.html#LIKE-RLIKE), if it is new to you. The following example lists only those databases that start with the letter h and end with any other characters (the .\* part):

hive> **SHOW** DATABASES **LIKE** 'h.\*';

human\_resources

hive> ...

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.

The database directory is created under a top-level directory specified by the property hive.metastore.warehouse.dir, which we discussed in [Local Mode Configuration](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch02.html#Local-Mode-Configuration) and [Distributed and Pseudodistributed Mode Configuration](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch02.html#Distributed-and-Pseudo-Distributed-Mode-Configuration). Assuming you are using the default value for this property,/user/hive/warehouse, when the financials database is created, Hive will create the directory /user/hive/warehouse/financials.db. Note the .dbextension.

You can override this default location for the new directory as shown in this example:

hive> **CREATE** **DATABASE** financials

> **LOCATION** '/my/preferred/directory';

You can add a descriptive comment to the database, which will be shown by the DESCRIBE DATABASE <database> command.

hive> **CREATE** **DATABASE** financials

> **COMMENT** 'Holds all financial tables';

hive> **DESCRIBE** **DATABASE** financials;

financials Holds **all** financial tables

hdfs://master-server/**user**/hive/warehouse/financials.db

Note that DESCRIBE DATABASE also shows the directory location for the database. In this example, the URI scheme is hdfs. For a MapR installation, it would be maprfs. For an Amazon Elastic MapReduce (EMR) cluster, it would also be hdfs, but you could set hive.metastore.warehouse.dir to use Amazon S3 explicitly (i.e., by specifying s3n://bucketname/… as the property value). You could use s3 as the scheme, but the newer s3n is preferred.

In the output of DESCRIBE DATABASE, we’re showing master-server to indicate the URI authority, in this case a DNS name and optional port number (i.e., server:port) for the “master node” of the filesystem (i.e., where theNameNode service is running for HDFS). If you are running in pseudo-distributed mode, then the master server will be localhost. For local mode, the path will be a local path, file:///user/hive/warehouse/financials.db.

If the authority is omitted, Hive uses the master-server name and port defined by the property fs.default.name in the Hadoop configuration files, found in the $HADOOP\_HOME/conf directory.

To be clear, hdfs:///user/hive/warehouse/financials.db is equivalent tohdfs://master-server/user/hive/warehouse/financials.db, where master-server is your master node’s DNS name and optional port.

For completeness, when you specify a relative path (e.g., some/relative/path), Hive will put this under your home directory in the distributed filesystem (e.g.,hdfs:///user/<user-name>) for HDFS. However, if you are running in local mode, your current working directory is used as the parent ofsome/relative/path.

For script portability, it’s typical to omit the authority, only specifying it when referring to another distributed filesystem instance (including S3 buckets).

Lastly, you can associate key-value properties with the database, although their only function currently is to provide a way of adding information to the output of DESCRIBE DATABASE EXTENDED <database>:

hive> **CREATE** **DATABASE** financials

> **WITH** DBPROPERTIES ('creator' = 'Mark Moneybags', 'date' = '2012-01-02');

hive> **DESCRIBE** **DATABASE** financials;

financials hdfs://master-server/**user**/hive/warehouse/financials.db

hive> **DESCRIBE** **DATABASE** EXTENDED financials;

financials hdfs://master-server/**user**/hive/warehouse/financials.db

{date=2012-01-02, creator=Mark Moneybags);

The USE command sets a database as your working database, analogous to changing working directories in a filesystem:

hive> USE financials;

Now, commands such as SHOW TABLES; will list the tables in this database.

Unfortunately, there is no command to show you which database is your current working database! Fortunately, it’s always safe to repeat the USE …command; there is no concept in Hive of nesting of databases.

Recall that we pointed out a useful trick in [Variables and Properties](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch02.html#VariablesAndProperties) for setting a property to print the current database as part of the prompt (Hive v0.8.0 and later):

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

hive (financials)> USE **default**;

hive (**default**)> **set** hive.cli.print.**current**.db=**false**;

hive> ...

Finally, you can drop a database:

hive> **DROP** **DATABASE** IF **EXISTS** financials;

The IF EXISTS is optional and suppresses warnings if financials doesn’t exist.

By default, Hive won’t permit you to drop a database if it contains tables. You can either drop the tables first or append the CASCADE keyword to the command, which will cause the Hive to drop the tables in the database first:

hive> **DROP** **DATABASE** IF **EXISTS** financials **CASCADE**;

Using the RESTRICT keyword instead of CASCADE is equivalent to the default behavior, where existing tables must be dropped before dropping the database.

When a database is dropped, its directory is also deleted.

# Alter Database

You can set key-value pairs in the DBPROPERTIES associated with a database using the ALTER DATABASE command. No other metadata about the database can be changed, including its name and directory location:

hive> **ALTER** **DATABASE** financials **SET** DBPROPERTIES ('edited-by' = 'Joe Dba');

There is no way to delete or “unset” a DBPROPERTY.

# Creating Tables

The CREATE TABLE statement follows SQL conventions, but Hive’s version offers significant extensions to support a wide range of flexibility where the data files for tables are stored, the formats used, etc. We discussed many of these options in [Text File Encoding of Data Values](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch03.html#TextFileEncodingOfDataValues) and we’ll return to more advanced options later in [Chapter 15](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch15.html). In this section, we describe the other options available for the CREATE TABLE statement, adapting the employeestable declaration we used previously in [Collection Data Types](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch03.html#Collection-Data-Types):

**CREATE** **TABLE** IF **NOT** **EXISTS** mydb.employees (

name STRING **COMMENT** 'Employee name',

salary FLOAT **COMMENT** 'Employee salary',

subordinates ARRAY<STRING> **COMMENT** 'Names of subordinates',

deductions **MAP**<STRING, FLOAT>

**COMMENT** 'Keys are deductions names, values are percentages',

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

**COMMENT** 'Home address')

**COMMENT** 'Description of the table'

TBLPROPERTIES ('creator'='me', 'created\_at'='2012-01-02 10:00:00', ...)

**LOCATION** '/user/hive/warehouse/mydb.db/employees';

First, note that you can prefix a database name, mydb in this case, if you’re not currently working in the target database.

If you add the option IF NOT EXISTS, Hive will silently ignore the statement if the table already exists. This is useful in scripts that should create a table the first time they run.

However, the clause has a gotcha you should know. If the schema specified differs from the schema in the table that already exists, Hive won’t warn you. If your intention is for this table to have the new schema, you’ll have to drop the old table, losing your data, and then re-create it. Consider if you should use one or more ALTER TABLE statements to change the existing table schema instead. See [Alter Table](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#AlterTable) for details.

### **WARNING**

If you use IF NOT EXISTS and the existing table has a different schema than the schema in the CREATE TABLE statement, Hive will ignore the discrepancy.

You can add a comment to any column, after the type. Like databases, you can attach a comment to the table itself and you can define one or more table properties. In most cases, the primary benefit of TBLPROPERTIES is to add additional documentation in a key-value format. However, when we examine Hive’s integration with databases such as DynamoDB (see [DynamoDB](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch17.html#DynamoDBStorageHandler)), we’ll see that the TBLPROPERTIES can be used to express essential metadata about the database connection.

Hive automatically adds two table properties: last\_modified\_by holds the username of the last user to modify the table, and last\_modified\_time holds the epoch time in seconds of that modification.

### **NOTE**

A planned enhancement for Hive v0.10.0 is to add a SHOW TBLPROPERTIES table\_name command that will list just the TBLPROPERTIES for a table.

Finally, you can optionally specify a location for the table data (as opposed to metadata, which the metastore will always hold). In this example, we are showing the default location that Hive would use,/user/hive/warehouse/mydb.db/employees, where /user/hive/warehouse is the default “warehouse” location (as discussed previously), mydb.db is the database directory, and employees is the table directory.

By default, Hive always creates the table’s directory under the directory for the enclosing database. The exception is the default database. It doesn’t have a directory under /user/hive/warehouse, so a table in the default database will have its directory created directly in /user/hive/warehouse (unless explicitly overridden).

### **NOTE**

To avoid potential confusion, it’s usually better to use an external table if you don’t want to use the default location table. See [External Tables](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#ExternalTables) for details.

You can also copy the schema (but not the data) of an existing table:

**CREATE** **TABLE** IF **NOT** **EXISTS** mydb.employees2

**LIKE** mydb.employees;

This version also accepts the optional LOCATION clause, but note that no other properties, including the schema, can be defined; they are determined from the original table.

The SHOW TABLES command lists the tables. With no additional arguments, it shows the tables in the current working database. Let’s assume we have already created a few other tables, table1 and table2, and we did so in the mydbdatabase:

hive> USE mydb;

hive> **SHOW** TABLES;

employees

table1

table2

If we aren’t in the same database, we can still list the tables in that database:

hive> USE **default**;

hive> **SHOW** TABLES **IN** mydb;

employees

table1

table2

If we have a lot of tables, we can limit the ones listed using a regular expression, a concept we’ll discuss in detail in [LIKE and RLIKE](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch06.html#LIKE-RLIKE):

hive> USE mydb;

hive> **SHOW** TABLES 'empl.\*';

employees

Not all regular expression features are supported. If you know regular expressions, it’s better to test a candidate regular expression to make sure it actually works!

The regular expression in the single quote looks for all tables with names starting with empl and ending with any other characters (the .\* part).

### **NOTE**

Using the IN database\_name clause and a regular expression for the table names together is not supported.

We can also use the DESCRIBE EXTENDED mydb.employees command to show details about the table. (We can drop the mydb. prefix if we’re currently using the mydb database.) We have reformatted the output for easier reading and we have suppressed many details to focus on the items that interest us now:

hive> **DESCRIBE** EXTENDED mydb.employees;

name string Employee name

salary float Employee salary

subordinates array<string> **Names** **of** subordinates

deductions **map**<string,float> Keys **are** deductions **names**, **values** **are** percentages

address struct<street:string,city:string,**state**:string,zip:int> Home address

Detailed **Table** Information **Table**(tableName:employees, dbName:mydb, **owner**:me,

...

**location**:hdfs://master-server/**user**/hive/warehouse/mydb.db/employees,

**parameters**:{creator=me, created\_at='2012-01-02 10:00:00',

last\_modified\_user=me, last\_modified\_time=1337544510,

**comment**:Description **of** the **table**, ...}, ...)

Replacing EXTENDED with FORMATTED provides more readable but also more verbose output.

The first section shows the output of DESCRIBE without EXTENDED or FORMATTED (i.e., the schema including the comments for each column).

If you only want to see the schema for a particular column, append the column to the table name. Here, EXTENDED adds no additional output:

hive> **DESCRIBE** mydb.employees.salary;

salary float Employee salary

Returning to the extended output, note the line in the description that starts with location:. It shows the full URI path in HDFS to the directory where Hive will keep all the data for this table, as we discussed above.

### **WARNING**

We said that the last\_modified\_by and last\_modified\_time table properties are automatically created. However, they are only shown in the Detailed Table Information if a user-specified table property has also been defined!

## **Managed Tables**

The tables we have created so far are called managed tables or sometimes called internal tables, because Hive controls the lifecycle of their data (more or less). As we’ve seen, Hive stores the data for these tables in a subdirectory under the directory defined by hive.metastore.warehouse.dir (e.g.,/user/hive/warehouse), by default.

When we drop a managed table (see [Dropping Tables](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#DroppingTables)), Hive deletes the data in the table.

However, managed tables are less convenient for sharing with other tools. For example, suppose we have data that is created and used primarily by Pig or other tools, but we want to run some queries against it, but not give Hiveownership of the data. We can define an external table that points to that data, but doesn’t take ownership of it.

## **External Tables**

Suppose we are analyzing data from the stock markets. Periodically, we ingest the data for NASDAQ and the NYSE from a source like Infochimps (<http://infochimps.com/datasets>) and we want to study this data with many tools. (See the data sets named infochimps\_dataset\_4777\_download\_16185and infochimps\_dataset\_4778\_download\_16677, respectively, which are actually sourced from Yahoo! Finance.) The schema we’ll use next matches the schemas of both these data sources. Let’s assume the data files are in the distributed filesystem directory /data/stocks.

The following table declaration creates an external table that can read all the data files for this comma-delimited data in /data/stocks:

**CREATE** **EXTERNAL** **TABLE** IF **NOT** **EXISTS** stocks (

exchange STRING,

symbol STRING,

ymd STRING,

price\_open FLOAT,

price\_high FLOAT,

price\_low FLOAT,

price\_close FLOAT,

volume INT,

price\_adj\_close FLOAT)

**ROW** FORMAT DELIMITED FIELDS TERMINATED **BY** ','

**LOCATION** '/data/stocks';

The EXTERNAL keyword tells Hive this table is external and the LOCATION …clause is required to tell Hive where it’s located.

Because it’s external, Hive does not assume it owns the data. Therefore, dropping the table does not delete the data, although the metadata for the table will be deleted.

There are a few other small differences between managed and external tables, where some HiveQL constructs are not permitted for external tables. We’ll discuss those when we come to them.

However, it’s important to note that the differences between managed and external tables are smaller than they appear at first. Even for managed tables, you know where they are located, so you can use other tools, hadoop dfscommands, etc., to modify and even delete the files in the directories for managed tables. Hive may technically own these directories and files, but it doesn’t have full control over them! Recall, in [Schema on Read](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch03.html#SchemaOnRead), we said that Hive really has no control over the integrity of the files used for storage and whether or not their contents are consistent with the table schema. Even managed tables don’t give us this control.

Still, a general principle of good software design is to express intent. If the data is shared between tools, then creating an external table makes this ownership explicit.

You can tell whether or not a table is managed or external using the output of DESCRIBE EXTENDED tablename. Near the end of the Detailed Table Information output, you will see the following for managed tables:

... tableType:MANAGED\_TABLE)

For external tables, you will see the following:

... tableType:EXTERNAL\_TABLE)

As for managed tables, you can also copy the schema (but not the data) of an existing table:

**CREATE** **EXTERNAL** **TABLE** IF **NOT** **EXISTS** mydb.employees3

**LIKE** mydb.employees

**LOCATION** '/path/to/data';

### **NOTE**

If you omit the EXTERNAL keyword and the original table is external, the new table will also be external. If you omit EXTERNAL and the original table is managed, the new table will also be managed. However, if you include the EXTERNAL keyword and the original table is managed, the new table will be external. Even in this scenario, the LOCATION clause will still be optional.

# Partitioned, Managed Tables

The general notion of partitioning data is an old one. It can take many forms, but often it’s used for distributing load horizontally, moving data physically closer to its most frequent users, and other purposes.

Hive has the notion of partitioned tables. We’ll see that they have important performance benefits, and they can help organize data in a logical fashion, such as hierarchically.

We’ll discuss partitioned managed tables first. Let’s return to our employeestable and imagine that we work for a very large multinational corporation. Our HR people often run queries with WHERE clauses that restrict the results to a particular country or to a particular first-level subdivision (e.g., state in the United States or province in Canada). (First-level subdivision is an actual term, used here, for example: <http://www.commondatahub.com/state_source.jsp>.) We’ll just use the word state for simplicity. We have redundant state information in the address field. It is distinct from the state partition. We could remove the state element from address. There is no ambiguity in queries, since we have to use address.state to project the value inside theaddress. So, let’s partition the data first by country and then by state:

**CREATE** **TABLE** employees (

name STRING,

salary FLOAT,

subordinates ARRAY<STRING>,

deductions **MAP**<STRING, FLOAT>,

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

)

PARTITIONED **BY** (country STRING, **state** STRING);

Partitioning tables changes how Hive structures the data storage. If we create this table in the mydb database, there will still be an employees directory for the table:

hdfs://master\_server/user/hive/warehouse/mydb.db/employees

However, Hive will now create subdirectories reflecting the partitioning structure. For example:

...

.../employees/country=CA/state=AB

.../employees/country=CA/state=BC

...

.../employees/country=US/state=AL

.../employees/country=US/state=AK

...

Yes, those are the actual directory names. The state directories will contain zero or more files for the employees in those states.

Once created, the partition keys (country and state, in this case) behave like regular columns. There is one known exception, due to a bug (see [Aggregate functions](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch06.html#Aggregate-Functions)). In fact, users of the table don’t need to care if these “columns” are partitions or not, except when they want to optimize query performance.

For example, the following query selects all employees in the state of Illinois in the United States:

**SELECT** \* **FROM** employees

**WHERE** country = 'US' **AND** **state** = 'IL';

Note that because the country and state values are encoded in directory names, there is no reason to have this data in the data files themselves. In fact, the data just gets in the way in the files, since you have to account for it in the table schema, and this data wastes space.

Perhaps the most important reason to partition data is for faster queries. In the previous query, which limits the results to employees in Illinois, it is only necessary to scan the contents of one directory. Even if we have thousands of country and state directories, all but one can be ignored. For very large data sets, partitioning can dramatically improve query performance, but only if the partitioning scheme reflects common range filtering (e.g., by locations, timestamp ranges).

When we add predicates to WHERE clauses that filter on partition values, these predicates are called partition filters.

Even if you do a query across the entire US, Hive only reads the 65 directories covering the 50 states, 9 territories, and the District of Columbia, and 6 military “states” used by the armed services. You can see the full list here: <http://www.50states.com/abbreviations.htm>.

Of course, if you need to do a query for all employees around the globe, you can still do it. Hive will have to read every directory, but hopefully these broader disk scans will be relatively rare.

However, a query across all partitions could trigger an enormous MapReduce job if the table data and number of partitions are large. A highly suggested safety measure is putting Hive into “strict” mode, which prohibits queries of partitioned tables without a WHERE clause that filters on partitions. You can set the mode to “nonstrict,” as in the following session:

hive> **set** hive.mapred.**mode**=**strict**;

hive> **SELECT** e.name, e.salary **FROM** employees e **LIMIT** 100;

FAILED: Error **in** semantic analysis: **No** partition predicate **found** **for**

**Alias** "e" **Table** "employees"

hive> **set** hive.mapred.**mode**=nonstrict;

hive> **SELECT** e.name, e.salary **FROM** employees e **LIMIT** 100;

John Doe 100000.0

...

You can see the partitions that exist with the SHOW PARTITIONS command:

hive> **SHOW** PARTITIONS employees;

...

Country=CA/**state**=AB

country=CA/**state**=BC

...

country=US/**state**=AL

country=US/**state**=AK

...

If you have a lot of partitions and you want to see if partitions have been defined for particular partition keys, you can further restrict the command with an optional PARTITION clause that specifies one or more of the partitions with specific values:

hive> **SHOW** PARTITIONS employees PARTITION(country='US');

country=US/**state**=AL

country=US/**state**=AK

...

hive> **SHOW** PARTITIONS employees PARTITION(country='US', **state**='AK');

country=US/**state**=AK

The DESCRIBE EXTENDED employees command shows the partition keys:

hive> **DESCRIBE** EXTENDED employees;

name string,

salary float,

...

address struct<...>,

country string,

**state** string

Detailed **Table** Information...

partitionKeys:[FieldSchema(name:country, **type**:string, **comment**:**null**),

FieldSchema(name:**state**, **type**:string, **comment**:**null**)],

...

The schema part of the output lists the country and state with the other columns, because they are columns as far as queries are concerned. The Detailed Table Information includes the country and state as partition keys. The comments for both of these keys are null; we could have added comments just as for regular columns.

You create partitions in managed tables by loading data into them. The following example creates a US and CA (California) partition while loading data into it from a local directory, $HOME/california-employees. You must specify a value for each partition column. Notice how we reference the HOMEenvironment variable in HiveQL:

**LOAD** **DATA** **LOCAL** INPATH '${env:HOME}/california-employees'

**INTO** **TABLE** employees

PARTITION (country = 'US', **state** = 'CA');

The directory for this partition, …/employees/country=US/state=CA, will be created by Hive and all data files in $HOME/california-employees will be copied into it. See [Loading Data into Managed Tables](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch05.html#LoadingData) for more information on populating tables.

## **External Partitioned Tables**

You can use partitioning with external tables. In fact, you may find that this is your most common scenario for managing large production data sets. The combination gives you a way to “share” data with other tools, while still optimizing query performance.

You also have more flexibility in the directory structure used, as you define it yourself. We’ll see a particularly useful example in a moment.

Let’s consider a new example that fits this scenario well: logfile analysis. Most organizations use a standard format for log messages, recording a timestamp, severity (e.g., ERROR, WARNING, INFO), perhaps a server name and process ID, and then an arbitrary text message. Suppose our Extract, Transform, and Load (ETL) process ingests and aggregates logfiles in our environment, converting each log message to a tab-delimited record and also decomposing the timestamp into separate year, month, and day fields, and a combined hms field for the remaining hour, minute, and second parts of the timestamp, for reasons that will become clear in a moment. You could do this parsing of log messages using the string parsing functions built into Hive or Pig, for example. Alternatively, we could use smaller integer types for some of the timestamp-related fields to conserve space. Here, we are ignoring subsequent resolution.

Here’s how we might define the corresponding Hive table:

**CREATE** **EXTERNAL** **TABLE** IF **NOT** **EXISTS** log\_messages (

hms INT,

severity STRING,

server STRING,

process\_id INT,

message STRING)

PARTITIONED **BY** (**year** INT, **month** INT, **day** INT)

**ROW** FORMAT DELIMITED FIELDS TERMINATED **BY** '\t';

We’re assuming that a day’s worth of log data is about the correct size for a useful partition and finer grain queries over a day’s data will be fast enough.

Recall that when we created the nonpartitioned external stocks table, a LOCATION … clause was required. It isn’t used for external partitioned tables. Instead, an ALTER TABLE statement is used to add each partition separately. It must specify a value for each partition key, the year, month, and day, in this case (see [Alter Table](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#AlterTable) for more details on this feature). Here is an example, where we add a partition for January 2nd, 2012:

**ALTER** **TABLE** log\_messages **ADD** PARTITION(**year** = 2012, **month** = 1, **day** = 2)

**LOCATION** 'hdfs://master\_server/data/log\_messages/2012/01/02';

The directory convention we use is completely up to us. Here, we follow a hierarchical directory structure, because it’s a logical way to organize our data, but there is no requirement to do so. We could follow Hive’s directory naming convention (e.g., …/exchange=NASDAQ/symbol=AAPL), but there is no requirement to do so.

An interesting benefit of this flexibility is that we can archive old data on inexpensive storage, like Amazon’s S3, while keeping newer, more “interesting” data in HDFS. For example, each day we might use the following procedure to move data older than a month to S3:

* Copy the data for the partition being moved to S3. For example, you can use the hadoop distcp command:

hadoop distcp /data/log\_messages/2011/12/02 s3n://ourbucket/logs/2011/12/02

* Alter the table to point the partition to the S3 location:
* **ALTER** **TABLE** log\_messages PARTITION(**year** = 2011, **month** = 12, **day** = 2)

**SET** **LOCATION** 's3n://ourbucket/logs/2011/01/02';

* Remove the HDFS copy of the partition using the hadoop fs -rmrcommand:

hadoop fs -rmr /data/log\_messages/2011/01/02

You don’t have to be an Amazon Elastic MapReduce user to use S3 this way. S3 support is part of the Apache Hadoop distribution. You can still query this data, even queries that cross the month-old “boundary,” where some data is read from HDFS and some data is read from S3!

By the way, Hive doesn’t care if a partition directory doesn’t exist for a partition or if it has no files. In both cases, you’ll just get no results for a query that filters for the partition. This is convenient when you want to set up partitions before a separate process starts writing data to them. As soon as data is there, queries will return results from that data.

This feature illustrates another benefit: new data can be written to a dedicated directory with a clear distinction from older data in other directories. Also, whether you move old data to an “archive” location or delete it outright, the risk of tampering with newer data is reduced since the data subsets are in separate directories.

As for nonpartitioned external tables, Hive does not own the data and it does not delete the data if the table is dropped.

As for managed partitioned tables, you can see an external table’s partitions with SHOW PARTITIONS:

hive> **SHOW** PARTITIONS log\_messages;

...

**year**=2011/**month**=12/**day**=31

**year**=2012/**month**=1/**day**=1

**year**=2012/**month**=1/**day**=2

...

Similarly, the DESCRIBE EXTENDED log\_messages shows the partition keys both as part of the schema and in the list of partitionKeys:

hive> **DESCRIBE** EXTENDED log\_messages;

...

message string,

**year** int,

**month** int,

**day** int

Detailed **Table** Information...

partitionKeys:[FieldSchema(name:**year**, **type**:int, **comment**:**null**),

FieldSchema(name:**month**, **type**:int, **comment**:**null**),

FieldSchema(name:**day**, **type**:int, **comment**:**null**)],

...

This output is missing a useful bit of information, the actual location of the partition data. There is a location field, but it only shows Hive’s default directory that would be used if the table were a managed table. However, we can get a partition’s location as follows:

hive> **DESCRIBE** EXTENDED log\_messages PARTITION (**year**=2012, **month**=1, **day**=2);

...

**location**:s3n://ourbucket/logs/2011/01/02,

...

We frequently use external partitioned tables because of the many benefits they provide, such as logical data management, performant queries, etc.

ALTER TABLE … ADD PARTITION is not limited to external tables. You can use it with managed tables, too, when you have (or will have) data for partitions in directories created outside of the LOAD and INSERT options we discussed above. You’ll need to remember that not all of the table’s data will be under the usual Hive “warehouse” directory, and this data won’t be deleted when you drop the managed table! Hence, from a “sanity” perspective, it’s questionable whether you should dare to use this feature with managed tables.

## **Customizing Table Storage Formats**

In [Text File Encoding of Data Values](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch03.html#TextFileEncodingOfDataValues), we discussed that Hive defaults to a text file format, which is indicated by the optional clause STORED AS TEXTFILE, and you can overload the default values for the various delimiters when creating the table. Here we repeat the definition of the employees table we used in that discussion:

**CREATE** **TABLE** employees (

name STRING,

salary FLOAT,

subordinates ARRAY<STRING>,

deductions **MAP**<STRING, FLOAT>,

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

)

**ROW** FORMAT DELIMITED

FIELDS TERMINATED **BY** '\001'

COLLECTION ITEMS TERMINATED **BY** '\002'

**MAP** KEYS TERMINATED **BY** '\003'

LINES TERMINATED **BY** '\n'

STORED **AS** TEXTFILE;

TEXTFILE implies that all fields are encoded using alphanumeric characters, including those from international character sets, although we observed that Hive uses non-printing characters as “terminators” (delimiters), by default. When TEXTFILE is used, each line is considered a separate record.

You can replace TEXTFILE with one of the other built-in file formats supported by Hive, including SEQUENCEFILE and RCFILE, both of which optimize disk space usage and I/O bandwidth performance using binary encoding and optional compression. These formats are discussed in more detail in [Chapter 11](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch11.html)and [Chapter 15](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch15.html).

Hive draws a distinction between how records are encoded into files and how columns are encoded into records. You customize these behaviors separately.

The record encoding is handled by an input format object (e.g., the Java code behind TEXTFILE.) Hive uses a Java class (compiled module) namedorg.apache.hadoop.mapred.TextInputFormat. If you are unfamiliar with Java, the dotted name syntax indicates a hierarchical namespace tree ofpackages that actually corresponds to the directory structure for the Java code. The last name, TextInputFormat, is a class in the lowest-level package mapred.

The record parsing is handled by a serializer/deserializer or SerDe for short. For TEXTFILE and the encoding we described in [Chapter 3](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch03.html) and repeated in the example above, the SerDe Hive uses is another Java class called org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe.

For completeness, there is also an output format that Hive uses for writing the output of queries to files and to the console. For TEXTFILE, the Java class named org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat is used for output.

### **NOTE**

Hive uses an input format to split input streams into records, an output format to format records into output streams (i.e., the output of queries), and a SerDe to parse records into columns, when reading, and encodes columns into records, when writing. We’ll explore these distinctions in greater depth in [Chapter 15](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch15.html).

Third-party input and output formats and SerDes can be specified, a feature which permits users to customize Hive for a wide range of file formats not supported natively.

Here is a complete example that uses a custom SerDe, input format, and output format for files accessible through the Avro protocol, which we will discuss in detail in [Avro Hive SerDe](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch15.html#AVROHiveSerDe):

**CREATE** **TABLE** kst

PARTITIONED **BY** (ds string)

**ROW** FORMAT SERDE 'com.linkedin.haivvreo.AvroSerDe'

**WITH** SERDEPROPERTIES ('schema.url'='http://schema\_provider/kst.avsc')

STORED **AS**

INPUTFORMAT 'com.linkedin.haivvreo.AvroContainerInputFormat'

OUTPUTFORMAT 'com.linkedin.haivvreo.AvroContainerOutputFormat';

The ROW FORMAT SERDE … specifies the SerDe to use. Hive provides the WITH SERDEPROPERTIES feature that allows users to pass configuration information to the SerDe. Hive knows nothing about the meaning of these properties. It’s up to the SerDe to decide their meaning. Note that the name and value of each property must be a quoted string.

Finally, the STORED AS INPUTFORMAT … OUTPUTFORMAT … clause specifies the Java classes to use for the input and output formats, respectively. If you specify one of these formats, you are required to specify both of them.

Note that the DESCRIBE EXTENDED table command lists the input and output formats, the SerDe, and any SerDe properties in the DETAILED TABLE INFORMATION. For our example, we would see the following:

hive> **DESCRIBE** EXTENDED kst

...

inputFormat:com.linkedin.haivvreo.AvroContainerInputFormat,

outputFormat:com.linkedin.haivvreo.AvroContainerOutputFormat,

...

serdeInfo:SerDeInfo(name:**null**,

serializationLib:com.linkedin.haivvreo.AvroSerDe,

**parameters**:{**schema**.url=http://schema\_provider/kst.avsc})

...

Finally, there are a few additional CREATE TABLE clauses that describe more details about how the data is supposed to be stored. Let’s extend our previous stocks table example from [External Tables](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#ExternalTables):

**CREATE** **EXTERNAL** **TABLE** IF **NOT** **EXISTS** stocks (

exchange STRING,

symbol STRING,

ymd STRING,

price\_open FLOAT,

price\_high FLOAT,

price\_low FLOAT,

price\_close FLOAT,

volume INT,

price\_adj\_close FLOAT)

CLUSTERED **BY** (exchange, symbol)

SORTED **BY** (ymd **ASC**)

**INTO** 96 BUCKETS

**ROW** FORMAT DELIMITED FIELDS TERMINATED **BY** ','

**LOCATION** '/data/stocks';

The CLUSTERED BY … INTO … BUCKETS clause, with an optional SORTED BY …clause is used to optimize certain kinds of queries, which we discuss in detail in [Bucketing Table Data Storage](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch09.html#Bucketing).

# Dropping Tables

The familiar DROP TABLE command from SQL is supported:

**DROP** **TABLE** IF **EXISTS** employees;

The IF EXISTS keywords are optional. If not used and the table doesn’t exist, Hive returns an error.

For managed tables, the table metadata and data are deleted.

### **NOTE**

Actually, if you enable the Hadoop Trash feature, which is not on by default, the data is moved to the.Trash directory in the distributed filesystem for the user, which in HDFS is /user/$USER/.Trash. To enable this feature, set the property fs.trash.interval to a reasonable positive number. It’s the number of minutes between “trash checkpoints”; 1,440 would be 24 hours. While it’s not guaranteed to work for all versions of all distributed filesystems, if you accidentally drop a managed table with important data, you may be able to re-create the table, re-create any partitions, and then move the files from .Trash to the correct directories (using the filesystem commands) to restore the data.

For external tables, the metadata is deleted but the data is not.

# Alter Table

Most table properties can be altered with ALTER TABLE statements, which change metadata about the table but not the data itself. These statements can be used to fix mistakes in schema, move partition locations (as we saw in [External Partitioned Tables](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#ExternalPartitionedTables)), and do other operations.

### **WARNING**

ALTER TABLE modifies table metadata only. The data for the table is untouched. It’s up to you to ensure that any modifications are consistent with the actual data.

## **Renaming a Table**

Use this statement to rename the table log\_messages to logmsgs:

**ALTER** **TABLE** log\_messages **RENAME** **TO** logmsgs;

## **Adding, Modifying, and Dropping a Table Partition**

As we saw previously, ALTER TABLE table ADD PARTITION … is used to add a new partition to a table (usually an external table). Here we repeat the same command shown previously with the additional options available:

**ALTER** **TABLE** log\_messages **ADD** IF **NOT** **EXISTS**

PARTITION (**year** = 2011, **month** = 1, **day** = 1) **LOCATION** '/logs/2011/01/01'

PARTITION (**year** = 2011, **month** = 1, **day** = 2) **LOCATION** '/logs/2011/01/02'

PARTITION (**year** = 2011, **month** = 1, **day** = 3) **LOCATION** '/logs/2011/01/03'

...;

Multiple partitions can be added in the same query when using Hive v0.8.0 and later. As always, IF NOT EXISTS is optional and has the usual meaning.

### **WARNING**

Hive v0.7.X allows you to use the syntax with multiple partition specifications, but it actually uses just the first partition specification, silently ignoring the others! Instead, use a separate ALTER STATEMENTstatement for each partition.

Similarly, you can change a partition location, effectively moving it:

**ALTER** **TABLE** log\_messages PARTITION(**year** = 2011, **month** = 12, **day** = 2)

**SET** **LOCATION** 's3n://ourbucket/logs/2011/01/02';

This command does not move the data from the old location, nor does it delete the old data.

Finally, you can drop a partition:

**ALTER** **TABLE** log\_messages **DROP** IF **EXISTS** PARTITION(**year** = 2011, **month** = 12, **day** = 2);

The IF EXISTS clause is optional, as usual. For managed tables, the data for the partition is deleted, along with the metadata, even if the partition was created using ALTER TABLE … ADD PARTITION. For external tables, the data is not deleted.

There are a few more ALTER statements that affect partitions discussed later in [Alter Storage Properties](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#AlterStorageProperties) and [Miscellaneous Alter Table Statements](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#MiscellaneousAlterTableStatements).

## **Changing Columns**

You can rename a column, change its position, type, or comment:

**ALTER** **TABLE** log\_messages

CHANGE **COLUMN** hms hours\_minutes\_seconds INT

**COMMENT** 'The hours, minutes, and seconds part of the timestamp'

**AFTER** severity;

You have to specify the old name, a new name, and the type, even if the name or type is not changing. The keyword COLUMN is optional as is the COMMENTclause. If you aren’t moving the column, the AFTER other\_column clause is not necessary. In the example shown, we move the column after the severitycolumn. If you want to move the column to the first position, use FIRST instead of AFTER other\_column.

As always, this command changes metadata only. If you are moving columns, the data must already match the new schema or you must change it to match by some other means.

## **Adding Columns**

You can add new columns to the end of the existing columns, before any partition columns.

**ALTER** **TABLE** log\_messages **ADD** COLUMNS (

app\_name STRING **COMMENT** 'Application name',

session\_id LONG **COMMENT** 'The current session id');

The COMMENT clauses are optional, as usual. If any of the new columns are in the wrong position, use an ALTER COLUMN table CHANGE COLUMN statement for each one to move it to the correct position.

## **Deleting or Replacing Columns**

The following example removes all the existing columns and replaces them with the new columns specified:

**ALTER** **TABLE** log\_messages **REPLACE** COLUMNS (

hours\_mins\_secs INT **COMMENT** 'hour, minute, seconds from timestamp',

severity STRING **COMMENT** 'The message severity'

message STRING **COMMENT** 'The rest of the message');

This statement effectively renames the original hms column and removes the server and process\_id columns from the original schema definition. As for all ALTER statements, only the table metadata is changed.

The REPLACE statement can only be used with tables that use one of the nativeSerDe modules: DynamicSerDe or MetadataTypedColumnsetSerDe. Recall that the SerDe determines how records are parsed into columns (deserialization) and how a record’s columns are written to storage (serialization). See [Chapter 15](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch15.html) for more details on SerDes.

## **Alter Table Properties**

You can add additional table properties or modify existing properties, but not remove them:

**ALTER** **TABLE** log\_messages **SET** TBLPROPERTIES (

'notes' = 'The process id is no longer captured; this column is always NULL');

## **Alter Storage Properties**

There are several ALTER TABLE statements for modifying format and SerDe properties.

The following statement changes the storage format for a partition to be SEQUENCEFILE, as we discussed in [Creating Tables](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#CreatingTables) (see [Sequence Files](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch11.html#SequenceFiles) and [Chapter 15](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch15.html) for more information):

**ALTER** **TABLE** log\_messages

PARTITION(**year** = 2012, **month** = 1, **day** = 1)

**SET** FILEFORMAT SEQUENCEFILE;

The PARTITION clause is required if the table is partitioned.

You can specify a new SerDe along with SerDe properties or change the properties for the existing SerDe. The following example specifies that a table will use a Java class named com.example.JSONSerDe to process a file of JSON-encoded records:

**ALTER** **TABLE** table\_using\_JSON\_storage

**SET** SERDE 'com.example.JSONSerDe'

**WITH** SERDEPROPERTIES (

'prop1' = 'value1',

'prop2' = 'value2');

The SERDEPROPERTIES are passed to the SerDe module (the Java class com.example.JSONSerDe, in this case). Note that both the property names (e.g., prop1) and the values (e.g., value1) must be quoted strings.

The SERDEPROPERTIES feature is a convenient mechanism that SerDe implementations can exploit to permit user customization. We’ll see a real-world example of a JSON SerDe and how it uses SERDEPROPERTIES in [JSON SerDe](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch15.html#JSONSerDe).

The following example demonstrates how to add new SERDEPROPERTIES for the current SerDe:

**ALTER** **TABLE** table\_using\_JSON\_storage

**SET** SERDEPROPERTIES (

'prop3' = 'value3',

'prop4' = 'value4');

You can alter the storage properties that we discussed in [Creating Tables](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch04.html#CreatingTables):

**ALTER** **TABLE** stocks

CLUSTERED **BY** (exchange, symbol)

SORTED **BY** (symbol)

**INTO** 48 BUCKETS;

The SORTED BY clause is optional, but the CLUSTER BY and INTO … BUCKETSare required. (See also [Bucketing Table Data Storage](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch09.html#Bucketing) for information on the use of data bucketing.)

## **Miscellaneous Alter Table Statements**

In [Execution Hooks](https://www.safaribooksonline.com/library/view/programming-hive/9781449326944/ch12.html#ExecutionHooks), we’ll discuss a technique for adding execution “hooks” for various operations. The ALTER TABLE … TOUCH statement is used to trigger these hooks:

**ALTER** **TABLE** log\_messages TOUCH

PARTITION(**year** = 2012, **month** = 1, **day** = 1);

The PARTITION clause is required for partitioned tables. A typical scenario for this statement is to trigger execution of the hooks when table storage files have been modified outside of Hive. For example, a script that has just written new files for the 2012/01/01 partition for log\_message can make the following call to the Hive CLI:

hive -e 'ALTER TABLE log\_messages TOUCH PARTITION(year = 2012, month = 1, day = 1);'

This statement won’t create the table or partition if it doesn’t already exist. Use the appropriate creation commands in that case.

The ALTER TABLE … ARCHIVE PARTITION statement captures the partition files into a Hadoop archive (HAR) file. This only reduces the number of files in the filesystem, reducing the load on the NameNode, but doesn’t provide any space savings (e.g., through compression):

**ALTER** **TABLE** log\_messages ARCHIVE

PARTITION(**year** = 2012, **month** = 1, **day** = 1);

To reverse the operation, substitute UNARCHIVE for ARCHIVE. This feature is only available for individual partitions of partitioned tables.

Finally, various protections are available. The following statements prevent the partition from being dropped and queried:

**ALTER** **TABLE** log\_messages

PARTITION(**year** = 2012, **month** = 1, **day** = 1) ENABLE NO\_DROP;

**ALTER** **TABLE** log\_messages

PARTITION(**year** = 2012, **month** = 1, **day** = 1) ENABLE OFFLINE;

To reverse either operation, replace ENABLE with DISABLE. These operations also can’t be used with nonpartitioned tables.

# Hive Data Manipulation Language

There are multiple ways to modify data in Hive:

* + LOAD
  + INSERT
  + into Hive tables from queries
  + into directories from queries
  + into Hive tables from SQL
  + UPDATE
  + DELETE
  + MERGE
  + EXPORT and IMPORT commands

### Loading files into tables

Hive does not do any transformation while loading data into tables. Load operations are currently pure copy/move operations that move datafiles into locations corresponding to Hive tables.

##### Syntax

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

##### Synopsis

Load operations are currently pure copy/move operations that move datafiles into locations corresponding to Hive tables.

* filepath can be:
  + a relative path, such as project/data1
  + an absolute path, such as /user/hive/project/data1
  + a full URI with scheme and (optionally) an authority, such as hdfs://namenode:9000/user/hive/project/data1
* The target being loaded to can be a table or a partition. If the table is partitioned, then one must specify a specific partition of the table by specifying values for all of the partitioning columns.
* filepath can refer to a file (in which case Hive will move the file into the table) or it can be a directory (in which case Hive will move all the files within that directory into the table). In either case, filepath addresses a set of files.
* If the keyword LOCAL is specified, then:
  + the load command will look for filepath in the local file system. If a relative path is specified, it will be interpreted relative to the user's current working directory. The user can specify a full URI for local files as well - for example: [file:///user/hive/project/data1](file:///\\user\hive\project\data1)
  + the load command will try to copy all the files addressed by filepath to the target filesystem. The target file system is inferred by looking at the location attribute of the table. The copied data files will then be moved to the table.
* If the keyword LOCAL is not specified, then Hive will either use the full URI of filepath, if one is specified, or will apply the following rules:
  + If scheme or authority are not specified, Hive will use the scheme and authority from the hadoop configuration variable fs.default.name that specifies the Namenode URI.
  + If the path is not absolute, then Hive will interpret it relative to /user/<username>
  + Hive will move the files addressed by filepath into the table (or partition)
* If the OVERWRITE keyword is used then the contents of the target table (or partition) will be deleted and replaced by the files referred to by filepath; otherwise the files referred by filepath will be added to the table.

##### Notes

* filepath cannot contain subdirectories.
* If the keyword LOCAL is not given, filepath must refer to files within the same filesystem as the table's (or partition's) location.
* Hive does some minimal checks to make sure that the files being loaded match the target table. Currently it checks that if the table is stored in sequencefile format, the files being loaded are also sequencefiles, and vice versa.
* A bug that prevented loading a file when its name includes the "+" character is fixed in release 0.13.0 ([HIVE-6048](https://issues.apache.org/jira/browse/HIVE-6048)).
* Please read [CompressedStorage](https://cwiki.apache.org/confluence/display/Hive/CompressedStorage) if your datafile is compressed.

### Inserting data into Hive Tables from queries

Query Results can be inserted into tables by using the insert clause.

##### Syntax

|  |
| --- |
| Standard syntax:  INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select\_statement1 FROM from\_statement;  INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select\_statement1 FROM from\_statement;    Hive extension (multiple inserts):  FROM from\_statement  INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select\_statement1  [INSERT OVERWRITE TABLE tablename2 [PARTITION ... [IF NOT EXISTS]] select\_statement2]  [INSERT INTO TABLE tablename2 [PARTITION ...] select\_statement2] ...;  FROM from\_statement  INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select\_statement1  [INSERT INTO TABLE tablename2 [PARTITION ...] select\_statement2]  [INSERT OVERWRITE TABLE tablename2 [PARTITION ... [IF NOT EXISTS]] select\_statement2] ...;    Hive extension (dynamic partition inserts):  INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select\_statement FROM from\_statement;  INSERT INTO TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select\_statement FROM from\_statement; |

##### Synopsis

* INSERT OVERWRITE will overwrite any existing data in the table or partition
  + unless IF NOT EXISTS is provided for a partition (as of Hive [0.9.0](https://issues.apache.org/jira/browse/HIVE-2612)).
  + As of Hive 2.3.0 ([HIVE-15880](https://issues.apache.org/jira/browse/HIVE-15880)), if the table has [TBLPROPERTIES](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-listTableProperties) ("auto.purge"="true") the previous data of the table is not moved to Trash when INSERT OVERWRITE query is run against the table. This functionality is applicable only for managed tables (see [managed tables](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-ManagedandExternalTables)) and is turned off when "auto.purge" property is unset or set to false.
* INSERT INTO will append to the table or partition, keeping the existing data intact. (Note: INSERT INTO syntax is only available starting in version 0.8.)
  + As of Hive [0.13.0](https://issues.apache.org/jira/browse/HIVE-6406), a table can be made **immutable** by creating it with [TBLPROPERTIES ("immutable"="true")](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-CreateTable). The default is "immutable"="false".  
    INSERT INTO behavior into an immutable table is disallowed if any data is already present, although INSERT INTO still works if the immutable table is empty. The behavior of INSERT OVERWRITE is not affected by the "immutable" table property.  
    An immutable table is protected against accidental updates due to a script loading data into it being run multiple times by mistake. The first insert into an immutable table succeeds and successive inserts fail, resulting in only one set of data in the table, instead of silently succeeding with multiple copies of the data in the table.
* Inserts can be done to a table or a partition. If the table is partitioned, then one must specify a specific partition of the table by specifying values for all of the partitioning columns. If [hive.typecheck.on.insert](https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties#ConfigurationProperties-hive.typecheck.on.insert) is set to true, these values are validated, converted and normalized to conform to their column types (Hive [0.12.0](https://issues.apache.org/jira/browse/HIVE-5297) onward).
* Multiple insert clauses (also known as Multi Table Insert) can be specified in the same query.
* The output of each of the select statements is written to the chosen table (or partition). Currently the OVERWRITE keyword is mandatory and implies that the contents of the chosen table or partition are replaced with the output of corresponding select statement.
* The output format and serialization class is determined by the table's metadata (as specified via DDL commands on the table).
* As of [Hive 0.14](https://issues.apache.org/jira/browse/HIVE-5317), if a table has an OutputFormat that implements AcidOutputFormat and the system is configured to use a [transaction](https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions) manager that implements ACID, then INSERT OVERWRITE will be disabled for that table.  This is to avoid users unintentionally overwriting transaction history.  The same functionality can be achieved by using [TRUNCATE TABLE](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-TruncateTable) (for non-partitioned tables) or [DROP PARTITION](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-DropPartitions) followed by INSERT INTO.
* As of Hive [1.1.0](https://issues.apache.org/jira/browse/HIVE-9353) the TABLE keyword is optional.
* As of Hive [1.2.0](https://issues.apache.org/jira/browse/HIVE-9481) each INSERT INTO T can take a column list like INSERT INTO T (z, x, c1).  See Description of [HIVE-9481](https://issues.apache.org/jira/browse/HIVE-9481) for examples.

##### Notes

* Multi Table Inserts minimize the number of data scans required. Hive can insert data into multiple tables by scanning the input data just once (and applying different query operators) to the input data.
* Starting with [Hive 0.13.0](https://issues.apache.org/jira/browse/HIVE-1180), the select statement can include one or more common table expressions (CTEs) as shown in the [SELECT syntax](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Select#LanguageManualSelect-SelectSyntax). For an example, see [Common Table Expression](https://cwiki.apache.org/confluence/display/Hive/Common+Table+Expression#CommonTableExpression-CTEinViews,CTAS,andInsertStatements).

##### Dynamic Partition Inserts

**Version information**

This information reflects the situation in Hive 0.12; dynamic partition inserts were added in Hive 0.6.

In the dynamic partition inserts, users can give partial partition specifications, which means just specifying the list of partition column names in the PARTITION clause. The column values are optional. If a partition column value is given, we call this a static partition, otherwise it is a dynamic partition. Each dynamic partition column has a corresponding input column from the select statement. This means that the dynamic partition creation is determined by the value of the input column. The dynamic partition columns must be **specified last** among the columns in the SELECT statement and **in the same order** in which they appear in the PARTITION() clause.

Dynamic partition inserts are disabled by default prior to Hive 0.9.0 and enabled by default in Hive [0.9.0](https://issues.apache.org/jira/browse/HIVE-2835) and later. These are the relevant configuration properties for dynamic partition inserts:

| **Configuration property** | **Default** | **Note** |
| --- | --- | --- |
| hive.exec.dynamic.partition | true | Needs to be set to true to enable dynamic partition inserts |
| hive.exec.dynamic.partition.mode | strict | In strict mode, the user must specify at least one static partition in case the user accidentally overwrites all partitions, in nonstrict mode all partitions are allowed to be dynamic |
| hive.exec.max.dynamic.partitions.pernode | 100 | Maximum number of dynamic partitions allowed to be created in each mapper/reducer node |
| hive.exec.max.dynamic.partitions | 1000 | Maximum number of dynamic partitions allowed to be created in total |
| hive.exec.max.created.files | 100000 | Maximum number of HDFS files created by all mappers/reducers in a MapReduce job |
| hive.error.on.empty.partition | false | Whether to throw an exception if dynamic partition insert generates empty results |

###### Example

|  |
| --- |
| FROM page\_view\_stg pvs  INSERT OVERWRITE TABLE page\_view PARTITION(dt='2008-06-08', country)         SELECT pvs.viewTime, pvs.userid, pvs.page\_url, pvs.referrer\_url, null, null, pvs.ip, pvs.cnt |

Here the country partition will be dynamically created by the last column from the SELECT clause (i.e. pvs.cnt). Note that the name is not used. In nonstrict mode the dt partition could also be dynamically created.

###### Additional Documentation

* [Design Document](https://cwiki.apache.org/confluence/display/Hive/DynamicPartitions)
  + [Original design doc](https://issues.apache.org/jira/secure/attachment/12437909/dp_design.txt)
  + [HIVE-936](https://issues.apache.org/jira/browse/HIVE-936)
* [Tutorial: Dynamic-Partition Insert](https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-Dynamic-PartitionInsert)
* [HCatalog Dynamic Partitioning](https://cwiki.apache.org/confluence/display/Hive/HCatalog+DynamicPartitions)
  + [Usage with Pig](https://cwiki.apache.org/confluence/display/Hive/HCatalog+DynamicPartitions#HCatalogDynamicPartitions-UsagewithPig)
  + [Usage from MapReduce](https://cwiki.apache.org/confluence/display/Hive/HCatalog+DynamicPartitions#HCatalogDynamicPartitions-UsagefromMapReduce)

### Writing data into the filesystem from queries

Query results can be inserted into filesystem directories by using a slight variation of the syntax above:

##### Syntax

|  |
| --- |
| Standard syntax:  INSERT OVERWRITE [LOCAL] DIRECTORY directory1    [ROW FORMAT row\_format] [STORED AS file\_format] (Note: Only available starting with Hive 0.11.0)    SELECT ... FROM ...    Hive extension (multiple inserts):  FROM from\_statement  INSERT OVERWRITE [LOCAL] DIRECTORY directory1 select\_statement1  [INSERT OVERWRITE [LOCAL] DIRECTORY directory2 select\_statement2] ...      row\_format    : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS TERMINATED BY char]          [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]          [NULL DEFINED AS char] (Note: Only available starting with Hive 0.13) |

##### Synopsis

* Directory can be a full URI. If scheme or authority are not specified, Hive will use the scheme and authority from the hadoop configuration variable fs.default.name that specifies the Namenode URI.
* If LOCAL keyword is used, Hive will write data to the directory on the local file system.
* Data written to the filesystem is serialized as text with columns separated by ^A and rows separated by newlines. If any of the columns are not of primitive type, then those columns are serialized to JSON format.

##### Notes

* INSERT OVERWRITE statements to directories, local directories, and tables (or partitions) can all be used together within the same query.
* INSERT OVERWRITE statements to HDFS filesystem directories are the best way to extract large amounts of data from Hive. Hive can write to HDFS directories in parallel from within a map-reduce job.
* The directory is, as you would expect, OVERWRITten; in other words, if the specified path exists, it is clobbered and replaced with the output.
* As of Hive [0.11.0](https://issues.apache.org/jira/browse/HIVE-3682) the separator used can be specified; in earlier versions it was always the ^A character (\001). However, custom separators are only supported for LOCAL writes in Hive versions 0.11.0 to 1.1.0 – this bug is fixed in version 1.2.0 (see [HIVE-5672](https://issues.apache.org/jira/browse/HIVE-5672)).
* In [Hive 0.14](https://issues.apache.org/jira/browse/HIVE-5317), inserts into [ACID](https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions) compliant tables will deactivate vectorization for the duration of the select and insert.  This will be done automatically.  ACID tables that have data inserted into them can still be queried using vectorization.

### Inserting values into tables from SQL

The INSERT...VALUES statement can be used to insert data into tables directly from SQL.

**Version Information**

INSERT...VALUES is available starting in [Hive 0.14](https://issues.apache.org/jira/browse/HIVE-5317).

##### Syntax

|  |
| --- |
| Standard Syntax:  INSERT INTO TABLE tablename [PARTITION (partcol1[=val1], partcol2[=val2] ...)] VALUES values\_row [, values\_row ...]    Where values\_row is:  ( value [, value ...] )  where a value is either null or any valid SQL literal |

##### Synopsis

* Each row listed in the VALUES clause is inserted into table tablename.
* Values must be provided for every column in the table. The standard SQL syntax that allows the user to insert values into only some columns is not yet supported. To mimic the standard SQL, nulls can be provided for columns the user does not wish to assign a value to.
* Dynamic partitioning is supported in the same way as for [INSERT...SELECT](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML#LanguageManualDML-DynamicPartitionInserts).
* If the table being inserted into supports [ACID](https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions) and a transaction manager that supports ACID is in use, this operation will be auto-committed upon successful completion.
* Hive does not support literals for complex types (array, map, struct, union), so it is not possible to use them in INSERT INTO...VALUES clauses. This means that the user cannot insert data into a complex datatype column using the INSERT INTO...VALUES clause.

##### Examples

|  |
| --- |
| CREATE TABLE students (name VARCHAR(64), age INT, gpa DECIMAL(3, 2))    CLUSTERED BY (age) INTO 2 BUCKETS STORED AS ORC;    INSERT INTO TABLE students    VALUES ('fred flintstone', 35, 1.28), ('barney rubble', 32, 2.32);      CREATE TABLE pageviews (userid VARCHAR(64), link STRING, came\_from STRING)    PARTITIONED BY (datestamp STRING) CLUSTERED BY (userid) INTO 256 BUCKETS STORED AS ORC;    INSERT INTO TABLE pageviews PARTITION (datestamp = '2014-09-23')    VALUES ('jsmith', 'mail.com', 'sports.com'), ('jdoe', 'mail.com', null);    INSERT INTO TABLE pageviews PARTITION (datestamp)    VALUES ('tjohnson', 'sports.com', 'finance.com', '2014-09-23'), ('tlee', 'finance.com', null, '2014-09-21'); |

### Update

**Version Information**

UPDATE is available starting in [Hive 0.14](https://issues.apache.org/jira/browse/HIVE-5317).

Updates can only be performed on tables that support ACID. See [Hive Transactions](https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions) for details.

##### Syntax

|  |
| --- |
| Standard Syntax:  UPDATE tablename SET column = value [, column = value ...] [WHERE expression] |

##### Synopsis

* The referenced column must be a column of the table being updated.
* The value assigned must be an expression that Hive supports in the select clause.  Thus arithmetic operators, UDFs, casts, literals, etc. are supported.  Subqueries are not supported.
* Only rows that match the WHERE clause will be updated.
* Partitioning columns cannot be updated.
* Bucketing columns cannot be updated.
* In Hive 0.14, upon successful completion of this operation the changes will be auto-committed.

##### Notes

* Vectorization will be turned off for update operations.  This is automatic and requires no action on the part of the user.  Non-update operations are not affected.  Updated tables can still be queried using vectorization.
* In version 0.14 it is recommended that you set [hive.optimize.sort.dynamic.partition](https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties#ConfigurationProperties-hive.optimize.sort.dynamic.partition)=false when doing updates, as this produces more efficient execution plans.

### Delete

**Version Information**

DELETE is available starting in [Hive 0.14](https://issues.apache.org/jira/browse/HIVE-5317).

Deletes can only be performed on tables that support ACID. See [Hive Transactions](https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions) for details.

##### Syntax

|  |
| --- |
| Standard Syntax:  DELETE FROM tablename [WHERE expression] |

##### Synopsis

* Only rows that match the WHERE clause will be deleted.
* In Hive 0.14, upon successful completion of this operation the changes will be auto-committed.

##### Notes

* Vectorization will be turned off for delete operations.  This is automatic and requires no action on the part of the user.  Non-delete operations are not affected.  Tables with deleted data can still be queried using vectorization.
* In version 0.14 it is recommended that you set [hive.optimize.sort.dynamic.partition](https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties#ConfigurationProperties-hive.optimize.sort.dynamic.partition)=false when doing deletes, as this produces more efficient execution plans.

### Merge

**Version Information**

MERGE is available starting in [Hive 2.2](https://issues.apache.org/jira/browse/HIVE-10924).

Merge can only be performed on tables that support ACID. See [Hive Transactions](https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions) for details.

##### Syntax

|  |
| --- |
| Standard Syntax:  MERGE INTO <target table> AS T USING <source expression/table> AS S  ON <boolean expression1>  WHEN MATCHED [AND <boolean expression2>] THEN UPDATE SET <set clause list>  WHEN MATCHED [AND <boolean expression3>] THEN DELETE  WHEN NOT MATCHED [AND <boolean expression4>] THEN INSERT VALUES<value list> |

##### Synopsis

* [Merge](https://en.wikipedia.org/wiki/Merge_(SQL)) allows actions to be performed on a target table based on the results of a join with a source table.
* In Hive 2.2, upon successful completion of this operation the changes will be auto-committed.

##### Performance Note

SQL Standard requires that an error is raised if the ON clause is such that more than 1 row in source matches a row in target.  This check is computationally expensive and may affect the overall runtime of a MERGE statement significantly.  [hive.merge.cardinality.check](https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties#ConfigurationProperties-hive.merge.cardinality.check)=false may be used to disable the check at your own risk.  If the check is disabled, but the statement has such a cross join effect, it may lead to data corruption.

##### Notes

* 1, 2, or 3 WHEN clauses may be present; at most 1 of each type:  UPDATE/DELETE/INSERT.
* WHEN NOT MATCHED must be the last WHEN clause.
* If both UPDATE and DELETE clauses are present, the first one in the statement must include [AND <boolean expression>].
* Vectorization will be turned off for merge operations.  This is automatic and requires no action on the part of the user.  Non-delete operations are not affected.  Tables with deleted data can still be queried using vectorization.