Namespace Access Description

hivevar Read/Write (v0.8.0 and later) User-defined custom variables.

hiveconf Read/Write Hive-specific configuration properties.

system Read/Write Configuration properties defined by Java.

env Read only Environment variables defined by the shell environment (e.g.,

bash).

Hive “One Shot” Commands

$ hive -e "SELECT \* FROM mytable LIMIT 3";

$ hive -S -e "select \* FROM mytable LIMIT 3" > /tmp/myquery

Executing Hive Queries from Files

$ cat /path/to/file/withqueries.hql

SELECT x.\* FROM src x;

$ hive

hive> source /path/to/file/withqueries.hql;

...

The .hiverc File

The last CLI option we’ll discuss is the -i file option, which lets you specify a file of

commands for the CLI to run as it starts, before showing you the prompt. Hive automatically

looks for a file named *.hiverc* in your HOME directory and runs the commands

it contains, if any.

Command History

You can use the up and down arrow keys to scroll through previous commands. Actually,

each previous line of input is shown separately; the CLI does not combine multiline

commands and queries into a single history entry. Hive saves the last 100,00 lines

into a file *$HOME/.hivehistory*.

Shell Execution

You don’t need to leave the hive CLI to run simple bash shell commands. Simply

type ! followed by the command and terminate the line with a semicolon (;):

hive> ! /bin/echo "what up dog";

"what up dog"

hive> ! pwd;

/home/me/hiveplay

Query Column Headers

As a final example that pulls together a few things we’ve learned, let’s tell the CLI to

print column headers, which is disabled by default. We can enable this feature by setting

the hiveconf property hive.cli.print.header to true:

38 | Chapter 2: Getting Started

hive> **set** hive.cli.print.header=**true**;

hive> **SELECT** \* **FROM** system\_logs **LIMIT** 3;

tstamp severity server message

1335667117.337715 ERROR server1 Hard drive hd1 **is** 90% **full**!

1335667117.338012 WARN server1 Slow response **from** server2.

1335667117.339234 WARN server2 Uh, Dude, I'm kinda busy **right** now...

Primitive Data Types

*Table 3-1. Primitive data types*

Type Size Literal syntax examples

TINYINT 1 byte signed integer. 20

SMALLINT 2 byte signed integer. 20

INT 4 byte signed integer. 20

BIGINT 8 byte signed integer. 20

BOOLEAN Boolean true or false. TRUE

FLOAT Single precision floating point. 3.14159

DOUBLE Double precision floating point. 3.14159

41

Type Size Literal syntax examples

STRING Sequence of characters. The character

set can be specified. Single or double

quotes can be used.

'Now is the time', "for all

good men"

TIMESTAMP (v0.8.0+) Integer, float, or string. 1327882394 (Unix epoch seconds),

1327882394.123456789 (Unix epoch

seconds plus nanoseconds), and

'2012-02-03

12:34:56.123456789' (JDBCcompliant

java.sql.Timestamp

format)

BINARY (v0.8.0+) Array of bytes. See discussion below

Note that Hive does not support “character arrays” (strings) with maximum-allowed

lengths, as is common in other SQL dialects. Relational databases offer this feature as

a performance optimization; fixed-length records are easier to index, scan, etc. In the

“looser” world in which Hive lives, where it may not own the data files and has to be

flexible on file format, Hive relies on the presence of delimiters to separate fields. Also,

Hadoop and Hive emphasize optimizing disk reading and writing performance, where

fixing the lengths of column values is relatively unimportant.

What if you run a query that wants to interpret a string column as a number? You can

explicitly cast one type to another as in the following example, where s is a string

column that holds a value representing an integer:

... **cast**(s **AS** INT) ...;

(To be clear, the AS INT are keywords, so lowercase would be fine.)

Collection Data Types

Type Description Literal syntax examples

STRUCT Analogous to a C struct or an “object.” Fields can be accessed

using the “dot” notation. For example, if a column name is of

type STRUCT {first STRING; last STRING}, then

the first name field can be referenced using name.first.

struct('John', 'Doe')

MAP A collection of key-value tuples, where the fields are accessed

using array notation (e.g., ['key']). For example, if a column

name is of type MAP with key→value pairs

'first'→'John' and 'last'→'Doe', then the last

name can be referenced using name['last'].

map('first', 'John',

'last', 'Doe')

ARRAY Ordered sequences of the *same* type that are indexable using

zero-based integers. For example, if a column name is of type

ARRAY of strings with the value ['John', 'Doe'], then

the second element can be referenced using name[1].

array('John', 'Doe')

Here is a table declaration that demonstrates how to use these types, an *employees* table

in a fictitious Human Resources application:

**CREATE TABLE** employees (

name STRING,

salary FLOAT,

subordinates ARRAY<STRING>,

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

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

Schema on Read

When you write data to a traditional database, either through loading external data,

writing the output of a query, doing UPDATE statements, etc., the database has total

control over the storage. The database is the “gatekeeper.” An important implication

of this control is that the database can enforce the schema as data is *written*. This is

called *schema on write*.

Hive has no such control over the underlying storage. There are many ways to create,

modify, and even damage the data that Hive will query. Therefore, Hive can only enforce

queries on *read*. This is called *schema on read*.

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” on page 66), 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 Hive *ownership* of the data. We can

define an *external* table that points to that data, but doesn’t take ownership of it.

External Tables

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.

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:

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

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**;