Hive

Hive is not a full database. The design constraints and limitations of Hadoop and HDFS

impose limits on what Hive can do. The biggest limitation is that Hive does not provide

record-level update, insert, nor delete.

Hive

queries have higher latency, due to the start-up overhead for MapReduce jobs. Queries

that would finish in seconds for a traditional database take longer for Hive, even for

relatively small data sets.1 Finally, Hive does not provide transactions.

So, Hive doesn’t provide crucial features required for OLTP, *Online Transaction Processing*.

It’s closer to being an OLAP tool, *Online Analytic Processing*, but as we’ll see,

Hive isn’t ideal for satisfying the “online” part of OLAP.

So, Hive is best suited for data warehouse applications, where a large data set is maintained

and mined for insights, reports, etc.



Bundled with the Hive distribution is the CLI, a simple web interface called *Hive web*

*interface* (HWI), and programmatic access through JDBC, ODBC, and a Thrift server

(see Chapter 16).

All commands and queries go to the Driver, which compiles the input, optimizes the

computation required, and executes the required steps, usually with MapReduce jobs.

When MapReduce jobs are required, Hive doesn’t generate Java MapReduce programs.

Instead, it uses built-in, generic Mapper and Reducer modules that are driven by an

XML file representing the “job plan.” In other words, these generic modules function

like mini language interpreters and the “language” to drive the computation is encoded

in XML.

Hive communicates with the *JobTracker* to initiate the MapReduce job. Hive does not

have to be running on the same master node with the JobTracker. In larger clusters,

it’s common to have edge nodes where tools like Hive run. They communicate remotely

with the JobTracker on the master node to execute jobs. Usually, the data files to be

processed are in HDFS, which is managed by the *NameNode*.

The Metastore is a separate relational database (usually a MySQL instance) where Hive

persists table schemas and other system metadata.

Local Mode, Pseudodistributed Mode, and Distributed Mode

We mentioned above that the default mode is *local mode*, where filesystem references use the local

filesystem. Also in local mode, when Hadoop jobs are executed (including most Hive

queries), the Map and Reduce tasks are run as part of the same process.

Actual clusters are configured in *distributed mode*, where all filesystem references that

aren’t full URIs default to the distributed filesystem (usually HDFS) and jobs are managed

by the *JobTracker* service, with individual tasks executed in separate processes.

A dilemma for developers working on personal machines is the fact that local mode

doesn’t closely resemble the behavior of a real cluster, which is important to remember

when testing applications. To address this need, a single machine can be configured to

run in *pseudodistributed mode*, where the behavior is identical to distributed mode,

namely filesystem references default to the distributed filesystem and jobs are managed

by the *JobTracker* service, but there is just a single machine. Hence, for example, HDFS

file block replication is limited to one copy. In other words, the behavior is like a singlenode

“cluster.”

Because Hive uses Hadoop jobs for most of its work, its behavior reflects the Hadoop

mode you’re using. However, even when running in distributed mode, Hive can decide

on a per-query basis whether or not it can perform the query using just local mode,

where it reads the data files and manages the MapReduce tasks itself, providing faster

turnaround. Hence, the distinction between the different modes is more of an

*execution* style for Hive than a *deployment* style, as it is for Hadoop.

When working with small data sets, using local mode execution will make Hive queries much faster. Setting the property set hive.exec.mode.local.auto=true; will cause Hive to use this mode more aggressively, even when you are running Hadoop in distributed or pseudodistributed mode. To always use this setting, add the command to your *$HOME/.hiverc* file.

Hive directory in local mode: [*file:///user/hive/warehouse*](file:///user/hive/warehouse),

Hive directory in cluster mode: *hdfs://namenode\_server/user/hive/warehouse*

Variables and Properties

The --define key=value option is effectively equivalent to the --hivevar key=value

option. Both let you define on the command line custom variables that you can reference

in Hive scripts to customize execution. This feature is only supported in Hive

v0.8.0 and later versions.

When you use this feature, Hive puts the key-value pair in the hivevar “namespace” to

distinguish these definitions from three other built-in namespaces, hiveconf, system,

and env.

*Table 2-3. Hive namespaces for variables and properties*

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).

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**;

Let’s look at the --hiveconf option, which is supported in Hive v0.7.X. It is used for

all properties that configure Hive behavior. We’ll use it with a property

hive.cli.print.current.db that was added in Hive v0.8.0. It turns on printing of the

current working database name in the CLI prompt. (See “Databases in

Hive” on page 49 for more on Hive databases.) The default database is named

default. This property is false by default:

$ hive --hiveconf hive.cli.print.current.db=true

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

hive.cli.print.current.db=true

Autocomplete

If you start typing and hit the Tab key, the CLI will autocomplete possible keywords

and function names. For example, if you type SELE and then the Tab key, the CLI will

complete the word SELECT.

If you type the Tab key at the prompt, you’ll get this reply:

hive>

Display **all** 407 possibilities? (y **or** n)

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

Hadoop dfs Commands from Inside Hive

You can run the hadoop dfs ... commands from within the hive CLI; just drop the

hadoop word from the command and add the semicolon at the end:

hive> dfs -ls / ;

**Found** 3 items

drwxr-xr-x - root supergroup 0 2011-08-17 16:27 /etl

drwxr-xr-x - edward supergroup 0 2012-01-18 15:51 /flag

drwxrwxr-x - hadoop supergroup 0 2010-02-03 17:50 /users

Comments in Hive Scripts

As of Hive v0.8.0, you can embed lines of comments that start with the string --, for

example:

*-- Copyright (c) 2012 Megacorp, LLC.*

*-- This is the best Hive script evar!!*

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:

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..

Text File Encoding of Data Values

*Table 3-3. Hive’s default record and field delimiters*

Delimiter Description

\n For text files, each line is a record, so the line feed character separates records.

^A (“control” A) Separates all fields (columns). Written using the octal code \001 when explicitly

specified in CREATE TABLE statements.

^B Separate the elements in an ARRAY or STRUCT, or the key-value pairs in a MAP.

Written using the octal code \002 when explicitly specified in CREATE TABLE

statements.

^C Separate the key from the corresponding value in MAP key-value pairs. Written using

the octal code \003 when explicitly specified in CREATE TABLE statements..

**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;

Chapter-4 :HiveQL: Data Definition

hive> **DESCRIBE DATABASE** financials;

financials Holds **all** financial tables

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

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.

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;

Customizing Table Storage Formats

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.

**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 SERDEPRO

PERTIES 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.

However, if staged\_employees is very large and you run 65 of these statements to cover

all states, then it means you are scanning staged\_employees 65 times! Hive offers an

alternative INSERT syntax that allows you to scan the input data once and split it multiple

ways. The following example shows this feature for creating the employees partitions

for three states:

**FROM** staged\_employees se

**INSERT** OVERWRITE **TABLE** employees

PARTITION (country = 'US', **state** = 'OR')

**SELECT** \* **WHERE** se.cnty = 'US' **AND** se.st = 'OR'

**INSERT** OVERWRITE **TABLE** employees

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

**SELECT** \* **WHERE** se.cnty = 'US' **AND** se.st = 'CA'

**INSERT** OVERWRITE **TABLE** employees

PARTITION (country = 'US', **state** = 'IL')

**SELECT** \* **WHERE** se.cnty = 'US' **AND** se.st = 'IL';

You can also mix *dynamic* and *static* partitions. This variation of the previous query

specifies a *static* value for the country (US) and a *dynamic* value for the state:

**INSERT** OVERWRITE **TABLE** employees

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

**SELECT** ..., se.cnty, se.st

**FROM** staged\_employees se

**WHERE** se.cnty = 'US';

The static partition keys must come before the dynamic partition keys.

*Table 5-1. Dynamic partitions properties*

Name Default Description

hive.exec.dynamic.parti

tion

false Set to true to enable dynamic partitioning.

hive.exec.dynamic.parti

tion.mode

strict Set to nonstrict to enable all partitions to be determined

dynamically.

hive.exec.max.dynamic.par

titions.pernode

100 The maximum number of dynamic partitions that can be created

by each mapper or reducer. Raises a fatal error if one

mapper or reducer attempts to create more than the threshold.

hive.exec.max.dynamic.par

titions

+1000 The total number of dynamic partitions that can be created by

one statement with dynamic partitioning. Raises a fatal error

if the limit is exceeded.

hive.exec.max.cre

ated.files

100000 The maximum total number of files that can be created globally.

A Hadoop counter is used to track the number of files created.

Raises a fatal error if the limit is exceeded.

ORDER BY and SORT BY

The ORDER BY clause is familiar from other SQL dialects. It performs a *total ordering* of

the query result set. This means that *all* the data is passed through a single reducer,

which may take an unacceptably long time to execute for larger data sets.

Hive adds an alternative, SORT BY, that orders the data only within each reducer, thereby

performing a *local ordering*, where each reducer’s output will be sorted. Better performance

is traded for total ordering.

DISTRIBUTE BY with SORT BY

DISTRIBUTE BY controls how map output is divided among reducers. All data that flows

through a MapReduce job is organized into key-value pairs. Hive must use this feature

internally when it converts your queries to MapReduce jobs.

Using DISTRIBUTE BY ... SORT BY or the shorthand CLUSTER BY clauses is a way to exploit

the parallelism of SORT BY, yet achieve a total ordering across the output files.

Queries that Sample Data

CLUSTER BY

In the previous example, the s.symbol column was used in the DISTRIBUTE BY clause,

and the s.symbol and the s.ymd columns in the SORT BY clause. Suppose that the same

columns are used in both clauses *and* all columns are sorted by ascending order (the

default). In this case, the CLUSTER BY clause is a shor-hand way of expressing the same

query.

hive> **SELECT** s.ymd, s.symbol, s.price\_close

> **FROM** stocks s

> **CLUSTER BY** s.symbol;

Queries that Sample Data

hive> **SELECT** \* **from** numbers TABLESAMPLE(BUCKET 3 **OUT OF** 10 **ON** rand()) s;

If we bucket on a column instead of rand(), then identical results are returned on multiple

runs:

hive> **SELECT** \* **from** numbers TABLESAMPLE(BUCKET 3 **OUT OF** 10 **ON** number) s;

Making Multiple Passes over the Same Data

hive> **FROM** history

> **INSERT** OVERWRITE sales **SELECT** \* **WHERE** action='purchased'

> **INSERT** OVERWRITE credits **SELECT** \* **WHERE** action='returned';

Other File Formats and Compression

One of Hive’s unique features is that Hive does not force data to be converted to a

specific format. Hive leverages Hadoop’s InputFormat APIs to read data from a variety

of sources, such as text files, sequence files, or even custom formats. Likewise, the

OutputFormat API is used to write data to various formats.