Connect hive from remote server

./hive --hiveconf fs.defaultFS=hdfs://c2:8020

./hive --help --service cli

-d,--define <key=value> Variable substitution to apply to Hive

commands. e.g. -d A=B or --define A=B

--database <databasename> Specify the database to use

-e <quoted-query-string> SQL from command line

-f <filename> SQL from files

-H,--help Print help information

--hiveconf <property=value> Use value for given property

--hivevar <key=value> Variable substitution to apply to Hive

commands. e.g. --hivevar A=B

-i <filename> Initialization SQL file

-S,--silent Silent mode in interactive shell

-v,--verbose Verbose mode (echo executed SQL to the

console)

Inside the CLI, variables are displayed and changed using the SET command. For example, the following session shows the value for one variable, in the env namespace, and then all variable definitions!

hive> set hivevar:foo=bar;

hive> set hivevar:foo;

hivevar:foo=bar

hive>

Variable references in queries are replaced in the CLI before the query is sent to the query processor. Consider the following hive CLI session (v0.8.X only):

hive> create table toss1(i int, ${hivevar:foo} string);

OK

Time taken: 1.825 seconds

hive> describe toss1;

FAILED: SemanticException [Error 10001]: Table not found toos1

hive> describe toss1;

OK

i int

bar string

Time taken: 0.36 seconds, Fetched: 2 row(s)

It’s also useful to know about the system namespace, which provides read-write access to Java system properties, and the env namespace, which provides read-only access to environment variables: hive> set system:user.name; system:user.name=myusername

Hive “One Shot” Commands The user may wish to run one or more queries (semicolon separated) and then have the hive CLI exit immediately after completion. The CLI accepts a -e command argument that enables this feature. If mytable has a string and integer column, we might see the following output: $ hive -e "SELECT \* FROM mytable LIMIT 3";

Executing Hive Queries from Files Hive can execute one or more queries that were saved to a file using the -f file argument. By convention, saved Hive query files use the .q or .hql extension. $ hive -f /path/to/file/withqueries.hql If you are already inside the Hive shell you can use the SOURCE command to execute a script file. Here is an example: $ cat /path/to/file/withqueries.hql SELECT x.\* FROM src x; $ hive hive> source /path/to/file/withqueries.hql; ... By the way, we’ll occasionally use the name src (“source”) for tables in queries when the name of the table is irrelevant for the example. This convention is taken from the unit tests in Hive’s source code; first create a src table before all tests.

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. These files are convenient for commands that you run frequently, such as setting system properties (see “Variables and Properties” on page 31) or adding Java archives (JAR files) of custom Hive extensions to Hadoop’s distributed cache (as discussed in Chapter 15). The following shows an example of a typical $HOME/.hiverc file: ADD JAR /path/to/custom\_hive\_extensions.jar; set hive.cli.print.current.db=true; set hive.exec.mode.local.auto=true; The first line adds a JAR file to the Hadoop distributed cache. The second line modifies the CLI prompt to show the current working Hive database, as we described earlier in “Variables and Properties” on page 31. The last line “encourages” Hive to be more aggressive about using local-mode execution when possible, even when Hadoop is running in distributed or pseudo-distributed mode, which speeds up queries for small data sets.

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 This method of accessing hadoop commands is actually more efficient than using the hadoop dfs ... equivalent at the bash shell, because the latter starts up a new JVM instance each time, whereas Hive just runs the same code in its current process. You can see a full listing of help on the options supported by dfs using this command: hive> dfs -help; See also http://hadoop.apache.org/common/docs/r0.20.205.0/file\_system\_shell.html or similar documentation for your Hadoop distribution.

Comments in hive cli

--this is a comment.

=======

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 www.it-ebooks.info

set hive.cli.print.header=true;

(Page 39).

DATATYPES::

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

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

What if you run a query that wants to compare a float column to a double column or compare a value of one integer type with a value of a different integer type? Hive will implicitly cast any integer to the larger of the two integer types, cast FLOAT to DOUBLE, and cast any integer value to DOUBLE, as needed, so it is comparing identical types. 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 type::

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]

As for simple types, the case of the type name is ignored. Most relational databases don’t support such collection types, because using them tends to break normal form. For example, in traditional data models, structs might be captured in separate tables, with foreign key relations between the tables, as appropriate. A practical problem with breaking normal form is the greater risk of data duplication, leading to unnecessary disk space consumption and potential data inconsistencies, as duplicate copies can grow out of sync as changes are made.

However, in Big Data systems, a benefit of sacrificing normal form is higher processing throughput. Scanning data off hard disks with minimal “head seeks” is essential when processing terabytes to petabytes of data. Embedding collections in records makes retrieval faster with minimal seeks. Navigating each foreign key relationship requires seeking across the disk, with significant performance overhead.

HIVE default and field defaulters:

\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.

John Doe^A100000.0^AMary Smith^BTodd Jones^AFederal Taxes^C.2^BState Taxes^C.05^BInsurance^C.1^A1 Michigan Ave.^BChicago^BIL^B60600 Mary Smith^A80000.0^ABill King^AFederal Taxes^C.2^BState Taxes^C. 05^BInsurance^C.1^A100 Ontario St.^BChicago^BIL^B60601

{ "name": "John Doe", "salary": 100000.0, "subordinates": ["Mary Smith", "Todd Jones"], "deductions": { "Federal Taxes": .2, "State Taxes": .05, "Insurance": .1 }, "address": { "street": "1 Michigan Ave.", "city": "Chicago", "state": "IL", "zip": 60600 } }

create TABLE employees(

> name STRING,

> salary FLOAT,

> subordinates ARRAY<STRING>,

> deductions map<STRING, FLOAT>,

> address STRUCT<street: STRING, city:STRING, state:STRING, pin: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;

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. So what if the schema doesn’t match the file contents? Hive does the best that it can to read the data. You will get lots of null values if there aren’t enough fields in each record to match the schema. If some fields are numbers and Hive encounters nonnumeric strings, it will return nulls for those fields. Above all else, Hive tries to recover from all errors as best it can.

HIVEQL –data definition

CREATE DATABASE::

hive> create database FINANCIALS;

OK

Time taken: 0.261 seconds

hive> create database IF NOT EXISTS FINANCIALS;

OK

Time taken: 0.052 seconds

hive> create database FINANCIALS;

FAILED: Execution Error, return code 1 from org.apache.hadoop.hive.ql.exec.DDLTask. Database FINANCIALS already exists

hive> create database human\_resources;

OK

Time taken: 0.265 seconds

hive> show databases like 'h.\*';

OK

human\_resources

Time taken: 0.064 seconds, Fetched: 1 row(s)

hive>

create database FINDIIFLOC LOCATION '/FINDIFFLOC';

CREATE DATABASE financials COMMENT 'HOLD ALL FINANCIAL TABLES';

==

hive> create database checkprop

> WITH DBPROPERTIES('creator'= 'saurabhkumar', 'date' = '2010-07-05');

OK

Time taken: 0.431 seconds

describe database EXTENDED checkprop;

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” on page 31 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;

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

hive (classicmodels)>

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

ALTER DATABASE NAME …..;

CREATE TABLE

CREATE TABLE If NOT EXISTS financials.employees(

> name STRING COMMENT 'EMPLOYEE NAME',

> salary FLOAT COMMENT 'EMPLOYEE SALARY',

> subordinates ARRAY<STRING> COMMENT 'EMPLOYEES WORKING UNDER current employee',

> deductions map<STRING, FLOAT> COMMENT 'keys are deductions names, values are percentages',

> address STRUCT<street: STRING,city: STRING ,state: STRING ,pincode: INT> )

> COMMENT 'description of table'

> TBLPROPERTIES ('creator' = 'saurabh', 'created\_at' = '2019-07-08' );

hive (financials)> describe employees;

OK

name string EMPLOYEE NAME

salary float EMPLOYEE SALARY

subordinates array<string> EMPLOYEES WORKING UNDER current employee

deductions map<string,float> keys are deductions names, values are percentages

address struct<street:string,city:string,state:string,pincode:int>

Time taken: 0.285 seconds, Fetched: 5 row(s)

hive (financials)> describe EXTENDED employees;

OK

name string EMPLOYEE NAME

salary float EMPLOYEE SALARY

subordinates array<string> EMPLOYEES WORKING UNDER current employee

deductions map<string,float> keys are deductions names, values are percentages

address struct<street:string,city:string,state:string,pincode:int>

Detailed Table Information Table(tableName:employees, dbName:financials, owner:hadoop, createTime:1562606196, lastAccessTime:0, retention:0, sd:StorageDescriptor(cols:[FieldSchema(name:name, type:string, comment:EMPLOYEE NAME), FieldSchema(name:salary, type:float, comment:EMPLOYEE SALARY), FieldSchema(name:subordinates, type:array<string>, comment:EMPLOYEES WORKING UNDER current employee), FieldSchema(name:deductions, type:map<string,float>, comment:keys are deductions names, values are percentages), FieldSchema(name:address, type:struct<street:string,city:string,state:string,pincode:int>, comment:null)], location:hdfs://procurementCluster/user/hive/warehouse/financials.db/employees, inputFormat:org.apache.hadoop.mapred.TextInputFormat, outputFormat:org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat, compressed:false, numBuckets:-1, serdeInfo:SerDeInfo(name:null, serializationLib:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, parameters:{serialization.format=1}), bucketCols:[], sortCols:[], parameters:{}, skewedInfo:SkewedInfo(skewedColNames:[], skewedColValues:[], skewedColValueLocationMaps:{}), storedAsSubDirectories:false), partitionKeys:[], parameters:{numFiles=0, created\_at=2019-07-08, COLUMN\_STATS\_ACCURATE={"BASIC\_STATS":"true"}, transient\_lastDdlTime=1562606196, comment=description of table, numRows=0, totalSize=0, rawDataSize=0, creator=saurabh}, viewOriginalText:null, viewExpandedText:null, tableType:MANAGED\_TABLE, rewriteEnabled:false)

You can also copy the schema (but not the data) of an existing table: CREATE TABLE IF NOT EXISTS mydb.employees2 LIKE mydb.employees;

(Page 54).

hive (financials)> show tables in financials;

OK

employee2

employees

hive (financials)> describe FORMATTED employees;

OK

# col\_name data\_type comment

name string EMPLOYEE NAME

salary float EMPLOYEE SALARY

subordinates array<string> EMPLOYEES WORKING UNDER current employee

deductions map<string,float> keys are deductions names, values are percentages

address struct<street:string,city:string,state:string,pincode:int>

# Detailed Table Information

Database: financials

Owner: hadoop

CreateTime: Mon Jul 08 22:46:36 IST 2019

LastAccessTime: UNKNOWN

Retention: 0

Location: hdfs://procurementCluster/user/hive/warehouse/financials.db/employees

Table Type: MANAGED\_TABLE

Table Parameters:

COLUMN\_STATS\_ACCURATE {\"BASIC\_STATS\":\"true\"}

comment description of table

created\_at 2019-07-08

creator saurabh

numFiles 0

numRows 0

rawDataSize 0

totalSize 0

transient\_lastDdlTime 1562606196

# Storage Information

SerDe Library: org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

InputFormat: org.apache.hadoop.mapred.TextInputFormat

OutputFormat: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

Compressed: No

Num Buckets: -1

Bucket Columns: []

Sort Columns: []

Storage Desc Params:

serialization.format 1

Time taken: 0.327 seconds, Fetched: 37 row(s)

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 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/da tasets) and we want to study this data with many tools. (See the data sets named infochimps\_dataset\_4777\_download\_16185 and 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

(Page 56).

=

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'; 56 | Chapter 4: HiveQL: Data Definition

(Page 56).

CREATE EXTERNAL TABLE IF NOT EXISTS mydb.employees3 LIKE mydb.employees LOCATION '/path/to/data';

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 employees table 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.common datahub.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 the address. 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

CREATE TABLE employee( 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;

OK=============

PARTITIONED ,MANAGED TABLE