Tutorial

Hive Tutorial

Concepts

What Is Hive

Hive is a data warehousing infrastructure based on Apache Hadoop. Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing on commodity hardware.

Hive is designed to enable easy data summarization, ad-hoc querying and analysis of large volumes of data. It provides SQL which enables users to do ad-hoc querying, summarization and data analysis easily. At the same time, Hive's SQL gives users multiple places to integrate their own functionality to do custom analysis, such as User Defined Functions (UDFs).

What Hive Is NOT

Hive is not designed for online transaction processing. It is best used for traditional data warehousing tasks.

Getting Started

For details on setting up Hive, HiveServer2, and Beeline, please refer to the GettingStarted guide.

Books about Hive lists some books that may also be helpful for getting started with Hive.

In the following sections we provide a tutorial on the capabilities of the system. We start by describing the concepts of data types, tables, and partitions (which are very similar to what you would find in a traditional relational DBMS) and then illustrate the capabilities of Hive with the help of some examples.

Data Units

In the order of granularity - Hive data is organized into:

- **Databases**: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on. Databases can also be used to enforce security for a user or group of users.
- **Tables**: Homogeneous units of data which have the same schema. An example of a table could be page_views table, where each row could comprise of the following columns (schema):
  - `timestamp`—which is of INT type that corresponds to a UNIX timestamp of when the page was viewed.
  - `userid` —which is of BIGINT type that identifies the user who viewed the page.
  - `page_url`–which is of STRING type that captures the location of the page.
  - `referer_url`–which is of STRING that captures the location of the page from where the user arrived at the current page.
  - `IP`–which is of STRING type that captures the IP address from where the page request was made.
- **Partitions**: Each Table can have one or more partition Keys which determines how the data is stored. Partitions—apart from being storage units—also allow the user to efficiently identify the rows that satisfy a specified criteria; for example, a date_partition of type STRING and country_partition of type STRING. Each unique value of the partition keys defines a partition of the Table. For example, all "US" data from "2009-12-23" is a partition of the page_views table. Therefore, if you run analysis on only the "US" data for 2009-12-23, you can run that query only on the relevant partition of the table, thereby speeding up the analysis significantly. Note however, that just because a partition is named 2009-12-23 does not mean that it contains all or only data from that date; partitions are named after dates for convenience; it is the user's job to guarantee the relationship between partition name and data content! Partition columns are virtual columns, they are not part of the data itself but are derived on load.
- **Buckets** (or **Clusters**): Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table. For example the page_views table may be bucketed by userid, which is one of the columns, other than the partitions columns, of the page_view table. These can be used to efficiently sample the data.

Note that it is not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.

Type System

Hive supports primitive and complex data types, as described below. See Hive Data Types for additional information.

Primitive Types

- Types are associated with the columns in the tables. The following Primitive types are supported:
- Integers
  - TINYINT—1 byte integer
  - SMALLINT—2 byte integer
  - INT—4 byte integer
  - BIGINT—8 byte integer
- Boolean type
  - BOOLEAN—TRUE/FALSE
- Floating point numbers
  - FLOAT—single precision
  - DOUBLE—Double precision
- Fixed point numbers
  - DECIMAL—a fixed point value of user defined scale and precision
- String types
  - STRING—sequence of characters in a specified character set
  - VARCHAR—sequence of characters in a specified character set with a maximum length
  - CHAR—sequence of characters in a specified character set with a defined length
- Date and time types
  - TIMESTAMP — A date and time without a timezone ("LocalDateTime" semantics)
  - TIMESTAMP WITH LOCAL TIME ZONE — A point in time measured down to nanoseconds ("Instant" semantics)
  - DATE—a date
- Binary types
  - BINARY—a sequence of bytes

The Types are organized in the following hierarchy (where the parent is a super type of all the children instances):

- Type
  - Primitive Type
    - Number
      - DOUBLE
        - FLOAT
          - BIGINT
            - INT
              - SMALLINT
                - TINYINT
    - STRING
    - BOOLEAN

This type hierarchy defines how the types are implicitly converted in the query language. Implicit conversion is allowed for types from child to an ancestor. So when a query expression expects type1 and the data is of type2, type2 is implicitly converted to type1 if type1 is an ancestor of type2 in the type hierarchy. Note that the type hierarchy allows the implicit conversion of STRING to DOUBLE.

Explicit type conversion can be done using the cast operator as shown in the #Built In Functions section below.

Complex Types

Complex Types can be built up from primitive types and other composite types using:

- Structs: the elements within the type can be accessed using the DOT (.) notation. For example, for a column c of type STRUCT {a INT; b INT}, the a field is accessed by the expression c.a
- Maps (key-value tuples): The elements are accessed using ['element name'] notation. For example in a map M comprising of a mapping from 'group' -> gid the gid value can be accessed using M['group']
- Arrays (indexable lists): The elements in the array have to be in the same type. Elements can be accessed using the [n] notation where n is an index (zero-based) into the array. For example, for an array A having the elements ['a', 'b', 'c'], A[1] retruns 'b'.

Using the primitive types and the constructs for creating complex types, types with arbitrary levels of nesting can be created. For example, a type User may comprise of the following fields:

- gender—which is a STRING.
- active—which is a BOOLEAN.

Timestamp

Timestamps have been the source of much confusion, so we try to document the intended semantics of Hive.

Timestamp ("LocalDateTime" semantics)

Java's "LocalDateTime" timestamps record a date and time as year, month, date, hour, minute, and seconds without a timezone. These timestamps always have those same values regardless of the local time zone.

For example, the timestamp value of "2014-12-12 12:34:56" is decomposed into year, month, day, hour, minute and seconds fields, but with no time zone information available. It does not correspond to any specific instant. It will always be the same value regardless of the local time zone. Unless your application uses UTC consistently, timestamp with local time zone is strongly preferred over timestamp for most applications. When users say an event is at 10:00, it is always in reference to a certain timezone and means a point in time, rather than 10:00 in an arbitrary time zone.

Timestamp with local time zone ("Instant" semantics)

Java's "Instant" timestamps define a point in time that remains constant regardless of where the data is read. Thus, the timestamp will be adjusted by the local time zone to match the original point in time.

Type	Value in America/Los_Angeles	Value in America/New_York
timestamp	2014-12-12 12:34:56	2014-12-12 12:34:56
timestamp with local time zone	2014-12-12 12:34:56	2014-12-12 15:34:56

Comparisons with other tools

	SQL 2003	Oracle	Sybase	Postgres	MySQL	Microsoft SQL	IBM DB2	Presto	Snowflake	Hive >= 3.1	Iceberg	Spark
timestamp	Local	Local	Local	Local	Instant	Other	Local	Local	Local	Local	Local	Instant
timestamp with local time zone		Instant							Instant	Instant		
timestamp with time zone	Offset	Offset	Offset	Instant			Offset	Offset	Offset		Instant	
timestamp without time zone	Local	Local		Local			Local					

Other definitions:

- Offset = Recording a point in time as well as the time zone offset in the writer's time zone.

Built In Operators and Functions

The operators and functions listed below are not necessarily up to date. (Hive Operators and UDFs has more current information.) In Beeline or the Hive CLI, use these commands to show the latest documentation:

```
SHOW FUNCTIONS;
DESCRIBE FUNCTION <function_name>;
DESCRIBE FUNCTION EXTENDED <function_name>;
```

Case-insensitive

All Hive keywords are case-insensitive, including the names of Hive operators and functions.

Built In Operators

- **Relational Operators**—The following operators compare the passed operands and generate a TRUE or FALSE value, depending on whether the comparison between the operands holds or not.

Relational Operator	Operand types	Description
A = B	all primitive types	TRUE if expression A is equivalent to expression B; otherwise FALSE

A != B	all primitive types	TRUE if expression A is *not* equivalent to expression B; otherwise FALSE
A < B	all primitive types	TRUE if expression A is less than expression B; otherwise FALSE
A <= B	all primitive types	TRUE if expression A is less than or equal to expression B; otherwise FALSE
A > B	all primitive types	TRUE if expression A is greater than expression B] otherwise FALSE
A >= B	all primitive types	TRUE if expression A is greater than or equal to expression B otherwise FALSE
A IS NULL	all types	TRUE if expression A evaluates to NULL otherwise FALSE
A IS NOT NULL	all types	FALSE if expression A evaluates to NULL otherwise TRUE
A LIKE B	strings	TRUE if string A matches the SQL simple regular expression B, otherwise FALSE. The comparison is done character by character. The _ character in B matches any character in A (similar to **.** in posix regular expressions), and the % character in B matches an arbitrary number of characters in A (similar to **.*** in posix regular expressions). For example, `'foobar' LIKE 'foo'` evaluates to FALSE where as `'foobar' LIKE 'foo___'` evaluates to TRUE and so does `'foobar' LIKE 'foo%'`. To escape % use \ (% matches one % character). If the data contains a semicolon, and you want to search for it, it needs to be escaped, `columnValue LIKE 'a\;b'`
A RLIKE B	strings	NULL if A or B is NULL, TRUE if any (possibly empty) substring of A matches the Java regular expression B (see Java regular expressions syntax), otherwise FALSE. For example, 'foobar' rlike 'foo' evaluates to TRUE and so does 'foobar' rlike '^f.*r$'.
A REGEXP B	strings	Same as RLIKE

- **Arithmetic Operators**—The following operators support various common arithmetic operations on the operands. All of them return number types.

Arithmetic Operators	Operand types	Description
A + B	all number types	Gives the result of adding A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands, for example, since every integer is a float. Therefore, float is a containing type of integer so the + operator on a float and an int will result in a float.
A - B	all number types	Gives the result of subtracting B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A * B	all number types	Gives the result of multiplying A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. Note that if the multiplication causing overflow, you will have to cast one of the operators to a type higher in the type hierarchy.
A / B	all number types	Gives the result of dividing B from A. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands. If the operands are integer types, then the result is the quotient of the division.
A % B	all number types	Gives the reminder resulting from dividing A by B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A & B	all number types	Gives the result of bitwise AND of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A | B	all number types	Gives the result of bitwise OR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
A ^ B	all number types	Gives the result of bitwise XOR of A and B. The type of the result is the same as the common parent(in the type hierarchy) of the types of the operands.
~A	all number types	Gives the result of bitwise NOT of A. The type of the result is the same as the type of A.

- **Logical Operators** — The following operators provide support for creating logical expressions. All of them return boolean TRUE or FALSE depending upon the boolean values of the operands.

Logical Operators	Operands types	Description
A AND B	boolean	TRUE if both A and B are TRUE, otherwise FALSE
A && B	boolean	Same as A AND B

A OR B	boolean	TRUE if either A or B or both are TRUE, otherwise FALSE
A || B	boolean	Same as A OR B
NOT A	boolean	TRUE if A is FALSE, otherwise FALSE
!A	boolean	Same as NOT A

- **Operators on Complex Types**—The following operators provide mechanisms to access elements in Complex Types

Operator	Operand types	Description
A[n]	A is an Array and n is an int	returns the nth element in the array A. The first element has index 0, for example, if A is an array comprising of ['foo', 'bar'] then A[0] returns 'foo' and A[1] returns 'bar'
M[key]	M is a Map<K, V> and key has type K	returns the value corresponding to the key in the map for example, if M is a map comprising of {'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'} then M['all'] returns 'foobar'
S.x	S is a struct	returns the x field of S, for example, for struct foobar {int foo, int bar} foobar.foo returns the integer stored in the foo field of the struct.

Built In Functions

- Hive supports the following built in functions:
  (Function list in source code: FunctionRegistry.java)

Return Type	Function Name (Signature)	Description
BIGINT	round(double a)	returns the rounded BIGINT value of the double
BIGINT	floor(double a)	returns the maximum BIGINT value that is equal or less than the double
BIGINT	ceil(double a)	returns the minimum BIGINT value that is equal or greater than the double
double	rand(), rand(int seed)	returns a random number (that changes from row to row). Specifiying the seed will make sure the generated random number sequence is deterministic.
string	concat(string A, string B,...)	returns the string resulting from concatenating B after A. For example, concat('foo', 'bar') results in 'foobar'. This function accepts arbitrary number of arguments and return the concatenation of all of them.
string	substr(string A, int start)	returns the substring of A starting from start position till the end of string A. For example, substr('foobar', 4) results in 'bar'
string	substr(string A, int start, int length)	returns the substring of A starting from start position with the given length, for example, substr('foobar', 4, 2) results in 'ba'
string	upper(string A)	returns the string resulting from converting all characters of A to upper case, for example, upper('fOoBaR') results in 'FOOBAR'
string	ucase(string A)	Same as upper
string	lower(string A)	returns the string resulting from converting all characters of B to lower case, for example, lower('fOoBaR') results in 'foobar'
string	lcase(string A)	Same as lower
string	trim(string A)	returns the string resulting from trimming spaces from both ends of A, for example, trim(' foobar ') results in 'foobar'
string	ltrim(string A)	returns the string resulting from trimming spaces from the beginning(left hand side) of A. For example, ltrim(' foobar ') results in 'foobar '
string	rtrim(string A)	returns the string resulting from trimming spaces from the end(right hand side) of A. For example, rtrim(' foobar ') results in ' foobar'
string	regexp_replace(string A, string B, string C)	returns the string resulting from replacing all substrings in B that match the Java regular expression syntax(See Java regular expressions syntax) with C. For example, regexp_replace('foobar', 'oo|ar',) returns 'fb'
int	size(Map<K.V>)	returns the number of elements in the map type
int	size(Array<T>)	returns the number of elements in the array type
value of <type>	cast(*<expr>* as *<type>*)	converts the results of the expression expr to <type>, for example, cast('1' as BIGINT) will convert the string '1' to it integral representation. A null is returned if the conversion does not succeed.

string	from_unixtime(int unixtime)	convert the number of seconds from the UNIX epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00"
string	to_date(string timestamp)	Return the date part of a timestamp string: to_date("1970-01-01 00:00:00") = "1970-01-01"
int	year(string date)	Return the year part of a date or a timestamp string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970
int	month(string date)	Return the month part of a date or a timestamp string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11
int	day(string date)	Return the day part of a date or a timestamp string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1
string	get_json_object(string json_string, string path)	Extract json object from a json string based on json path specified, and return json string of the extracted json object. It will return null if the input json string is invalid.

- The following built in aggregate functions are supported in Hive:

Return Type	Aggregation Function Name (Signature)	Description
BIGINT	count(*), count(expr), count(DISTINCT expr[, expr_.])	count(*)—Returns the total number of retrieved rows, including rows containing NULL values; count(expr)—Returns the number of rows for which the supplied expression is non-NULL; count(DISTINCT expr[, expr])—Returns the number of rows for which the supplied expression(s) are unique and non-NULL.
DOUBLE	sum(col), sum(DISTINCT col)	returns the sum of the elements in the group or the sum of the distinct values of the column in the group
DOUBLE	avg(col), avg(DISTINCT col)	returns the average of the elements in the group or the average of the distinct values of the column in the group
DOUBLE	min(col)	returns the minimum value of the column in the group
DOUBLE	max(col)	returns the maximum value of the column in the group

Language Capabilities

Hive's SQL provides the basic SQL operations. These operations work on tables or partitions. These operations are:

- Ability to filter rows from a table using a WHERE clause.
- Ability to select certain columns from the table using a SELECT clause.
- Ability to do equi-joins between two tables.
- Ability to evaluate aggregations on multiple "group by" columns for the data stored in a table.
- Ability to store the results of a query into another table.
- Ability to download the contents of a table to a local (for example,, nfs) directory.
- Ability to store the results of a query in a hadoop dfs directory.
- Ability to manage tables and partitions (create, drop and alter).
- Ability to plug in custom scripts in the language of choice for custom map/reduce jobs.

Usage and Examples

NOTE: Many of the following examples are out of date. More up to date information can be found in the LanguageManual.

The following examples highlight some salient features of the system. A detailed set of query test cases can be found at Hive Query Test Cases and the corresponding results can be found at Query Test Case Results.

Creating, Showing, Altering, and Dropping Tables

See Hive Data Definition Language for detailed information about creating, showing, altering, and dropping tables.

Creating Tables

An example statement that would create the page_view table mentioned above would be like:

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
STORED AS SEQUENCEFILE;
```

In this example, the columns of the table are specified with the corresponding types. Comments can be attached both at the column level as well as at the table level. Additionally, the partitioned by clause defines the partitioning columns which are different from the data columns and are actually not stored with the data. When specified in this way, the data in the files is assumed to be delimited with ASCII 001(ctrl-A) as the field delimiter and newline as the row delimiter.

The field delimiter can be parametrized if the data is not in the above format as illustrated in the following example:

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '1'
STORED AS SEQUENCEFILE;
```

The row delimintor currently cannot be changed since it is not determined by Hive but Hadoop delimiters.

It is also a good idea to bucket the tables on certain columns so that efficient sampling queries can be executed against the data set. If bucketing is absent, random sampling can still be done on the table but it is not efficient as the query has to scan all the data. The following example illustrates the case of the page_view table that is bucketed on the userid column:

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '1'
        COLLECTION ITEMS TERMINATED BY '2'
        MAP KEYS TERMINATED BY '3'
STORED AS SEQUENCEFILE;
```

In the example above, the table is clustered by a hash function of userid into 32 buckets. Within each bucket the data is sorted in increasing order of viewTime. Such an organization allows the user to do efficient sampling on the clustered column—n this case userid. The sorting property allows internal operators to take advantage of the better-known data structure while evaluating queries with greater efficiency.

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                friends ARRAY<BIGINT>, properties MAP<STRING, STRING>
                ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '1'
        COLLECTION ITEMS TERMINATED BY '2'
        MAP KEYS TERMINATED BY '3'
STORED AS SEQUENCEFILE;
```

In this example, the columns that comprise of the table row are specified in a similar way as the definition of types. Comments can be attached both at the column level as well as at the table level. Additionally, the partitioned by clause defines the partitioning columns which are different from the data columns and are actually not stored with the data. The CLUSTERED BY clause specifies which column to use for bucketing as well as how many buckets to create. The delimited row format specifies how the rows are stored in the hive table. In the case of the delimited format, this specifies how the fields are terminated, how the items within collections (arrays or maps) are terminated, and how the map keys are terminated. STORED AS SEQUENCEFILE indicates that this data is stored in a binary format (using hadoop SequenceFiles) on hdfs. The values shown for the ROW FORMAT and STORED AS clauses in the above, example represent the system defaults.

Table names and column names are case insensitive.

Browsing Tables and Partitions

```
SHOW TABLES;
```

To list existing tables in the warehouse; there are many of these, likely more than you want to browse.

```
SHOW TABLES 'page.*';
```

To list tables with prefix 'page'. The pattern follows Java regular expression syntax (so the period is a wildcard).

```
SHOW PARTITIONS page_view;
```

To list partitions of a table. If the table is not a partitioned table then an error is thrown.

```
DESCRIBE page_view;
```

To list columns and column types of table.

```
DESCRIBE EXTENDED page_view;
```

To list columns and all other properties of table. This prints lot of information and that too not in a pretty format. Usually used for debugging.

```
DESCRIBE EXTENDED page_view PARTITION (ds='2008-08-08');
```

To list columns and all other properties of a partition. This also prints lot of information which is usually used for debugging.

Altering Tables

To rename existing table to a new name. If a table with new name already exists then an error is returned:

```
ALTER TABLE old_table_name RENAME TO new_table_name;
```

To rename the columns of an existing table. Be sure to use the same column types, and to include an entry for each preexisting column:

```
ALTER TABLE old_table_name REPLACE COLUMNS (col1 TYPE, ...);
```

To add columns to an existing table:

```
ALTER TABLE tab1 ADD COLUMNS (c1 INT COMMENT 'a new int column', c2 STRING DEFAULT 'def val');
```

Note that a change in the schema (such as the adding of the columns), preserves the schema for the old partitions of the table in case it is a partitioned table. All the queries that access these columns and run over the old partitions implicitly return a null value or the specified default values for these columns.

In the later versions, we can make the behavior of assuming certain values as opposed to throwing an error in case the column is not found in a particular partition configurable.

Dropping Tables and Partitions

Dropping tables is fairly trivial. A drop on the table would implicitly drop any indexes(this is a future feature) that would have been built on the table. The associated command is:

```
DROP TABLE pv_users;
```

To dropping a partition. Alter the table to drop the partition.

```
ALTER TABLE pv_users DROP PARTITION (ds='2008-08-08')
```

- Note that any data for this table or partitions will be dropped and may not be recoverable. *

Loading Data

There are multiple ways to load data into Hive tables. The user can create an external table that points to a specified location within HDFS. In this particular usage, the user can copy a file into the specified location using the HDFS put or copy commands and create a table pointing to this location with all the relevant row format information. Once this is done, the user can transform the data and insert them into any other Hive table. For example, if the file /tmp/pv_2008-06-08.txt contains comma separated page views served on 2008-06-08, and this needs to be loaded into the page_view table in the appropriate partition, the following sequence of commands can achieve this:

```
CREATE EXTERNAL TABLE page_view_stg(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                ip STRING COMMENT 'IP Address of the User',
                country STRING COMMENT 'country of origination')
COMMENT 'This is the staging page view table'
ROW FORMAT DELIMITED FIELDS TERMINATED BY '44' LINES TERMINATED BY '12'
STORED AS TEXTFILE
LOCATION '/user/data/staging/page_view';

hadoop dfs -put /tmp/pv_2008-06-08.txt /user/data/staging/page_view

FROM page_view_stg pvs
INSERT OVERWRITE TABLE page_view PARTITION(dt='2008-06-08', country='US')
SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip
WHERE pvs.country = 'US';
```

* This code results in an error due to LINES TERMINATED BY limitation

FAILED: SemanticException 6:67 LINES TERMINATED BY only supports newline '\n' right now. Error encountered near token ''12''

See HIVE-5999 - Getting issue details... STATUS HIVE-11996 - Getting issue details... STATUS

In the example above, nulls are inserted for the array and map types in the destination tables but potentially these can also come from the external table if the proper row formats are specified.

This method is useful if there is already legacy data in HDFS on which the user wants to put some metadata so that the data can be queried and manipulated using Hive.

Additionally, the system also supports syntax that can load the data from a file in the local files system directly into a Hive table where the input data format is the same as the table format. If /tmp/pv_2008-06-08_us.txt already contains the data for US, then we do not need any additional filtering as shown in the previous example. The load in this case can be done using the following syntax:

```
   LOAD DATA LOCAL INPATH /tmp/pv_2008-06-08_us.txt INTO TABLE page_view PARTITION(date='2008-06-08',
country='US')
```

The path argument can take a directory (in which case all the files in the directory are loaded), a single file name, or a wildcard (in which case all the matching files are uploaded). If the argument is a directory, it cannot contain subdirectories. Similarly, the wildcard must match file names only.

In the case that the input file /tmp/pv_2008-06-08_us.txt is very large, the user may decide to do a parallel load of the data (using tools that are external to Hive). Once the file is in HDFS - the following syntax can be used to load the data into a Hive table:

```
   LOAD DATA INPATH '/user/data/pv_2008-06-08_us.txt' INTO TABLE page_view PARTITION(date='2008-06-08',
country='US')
```

It is assumed that the array and map fields in the input.txt files are null fields for these examples.

See Hive Data Manipulation Language for more information about loading data into Hive tables, and see External Tables for another example of creating an external table.

Querying and Inserting Data

The Hive query operations are documented in Select, and the insert operations are documented in Inserting data into Hive Tables from queries and Writing data into the filesystem from queries.

Simple Query

For all the active users, one can use the query of the following form:

```
INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active = 1;
```

Note that unlike SQL, we always insert the results into a table. We will illustrate later how the user can inspect these results and even dump them to a local file. You can also run the following query in Beeline or the Hive CLI:

```
SELECT user.*
FROM user
WHERE user.active = 1;
```

This will be internally rewritten to some temporary file and displayed to the Hive client side.

Partition Based Query

What partitions to use in a query is determined automatically by the system on the basis of where clause conditions on partition columns. For example, in order to get all the page_views in the month of 03/2008 referred from domain xyz.com, one could write the following query:

```
INSERT OVERWRITE TABLE xyz_com_page_views
SELECT page_views.*
FROM page_views
WHERE page_views.date >= '2008-03-01' AND page_views.date <= '2008-03-31' AND
      page_views.referrer_url like '%xyz.com';
```

Note that page_views.date is used here because the table (above) was defined with PARTITIONED BY(date DATETIME, country STRING) ; if you name your partition something different, don't expect .date to do what you think!

Joins

In order to get a demographic breakdown (by gender) of page_view of 2008-03-03 one would need to join the page_view table and the user table on the userid column. This can be accomplished with a join as shown in the following query:

```
INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
FROM user u JOIN page_view pv ON (pv.userid = u.id)
WHERE pv.date = '2008-03-03';
```

In order to do outer joins the user can qualify the join with LEFT OUTER, RIGHT OUTER or FULL OUTER keywords in order to indicate the kind of outer join (left preserved, right preserved or both sides preserved). For example, in order to do a full outer join in the query above, the corresponding syntax would look like the following query:

```
INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
FROM user u FULL OUTER JOIN page_view pv ON (pv.userid = u.id)
WHERE pv.date = '2008-03-03';
```

In order check the existence of a key in another table, the user can use LEFT SEMI JOIN as illustrated by the following example.

```
INSERT OVERWRITE TABLE pv_users
SELECT u.*
FROM user u LEFT SEMI JOIN page_view pv ON (pv.userid = u.id)
WHERE pv.date = '2008-03-03';
```

In order to join more than one tables, the user can use the following syntax:

```
INSERT OVERWRITE TABLE pv_friends
SELECT pv.*, u.gender, u.age, f.friends
FROM page_view pv JOIN user u ON (pv.userid = u.id) JOIN friend_list f ON (u.id = f.uid)
WHERE pv.date = '2008-03-03';
```

Aggregations

In order to count the number of distinct users by gender one could write the following query:

```
INSERT OVERWRITE TABLE pv_gender_sum
SELECT pv_users.gender, count (DISTINCT pv_users.userid)
FROM pv_users
GROUP BY pv_users.gender;
```

Multiple aggregations can be done at the same time, however, no two aggregations can have different DISTINCT columns .e.g while the following is possible

```
INSERT OVERWRITE TABLE pv_gender_agg
SELECT pv_users.gender, count(DISTINCT pv_users.userid), count(*), sum(DISTINCT pv_users.userid)
FROM pv_users
GROUP BY pv_users.gender;
```

however, the following query is not allowed

```
INSERT OVERWRITE TABLE pv_gender_agg
SELECT pv_users.gender, count(DISTINCT pv_users.userid), count(DISTINCT pv_users.ip)
FROM pv_users
GROUP BY pv_users.gender;
```

Multi Table/File Inserts

The output of the aggregations or simple selects can be further sent into multiple tables or even to hadoop dfs files (which can then be manipulated using hdfs utilities). For example, if along with the gender breakdown, one needed to find the breakdown of unique page views by age, one could accomplish that with the following query:

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_sum
    SELECT pv_users.gender, count_distinct(pv_users.userid)
    GROUP BY pv_users.gender

INSERT OVERWRITE DIRECTORY '/user/data/tmp/pv_age_sum'
    SELECT pv_users.age, count_distinct(pv_users.userid)
    GROUP BY pv_users.age;
```

The first insert clause sends the results of the first group by to a Hive table while the second one sends the results to a hadoop dfs files.

Dynamic-Partition Insert

In the previous examples, the user has to know which partition to insert into and only one partition can be inserted in one insert statement. If you want to load into multiple partitions, you have to use multi-insert statement as illustrated below.

```
    FROM page_view_stg pvs
    INSERT OVERWRITE TABLE page_view PARTITION(dt='2008-06-08', country='US')
           SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip WHERE pvs.
country = 'US'
    INSERT OVERWRITE TABLE page_view PARTITION(dt='2008-06-08', country='CA')
           SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip WHERE pvs.
country = 'CA'
    INSERT OVERWRITE TABLE page_view PARTITION(dt='2008-06-08', country='UK')
           SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip WHERE pvs.
country = 'UK';
```

In order to load data into all country partitions in a particular day, you have to add an insert statement for each country in the input data. This is very inconvenient since you have to have the priori knowledge of the list of countries exist in the input data and create the partitions beforehand. If the list changed for another day, you have to modify your insert DML as well as the partition creation DDLs. It is also inefficient since each insert statement may be turned into a MapReduce Job.

Dynamic-partition insert (or multi-partition insert) is designed to solve this problem by dynamically determining which partitions should be created and populated while scanning the input table. This is a newly added feature that is only available from version 0.6.0. In the dynamic partition insert, the input column values are evaluated to determine which partition this row should be inserted into. If that partition has not been created, it will create that partition automatically. Using this feature you need only one insert statement to create and populate all necessary partitions. In addition, since there is only one insert statement, there is only one corresponding MapReduce job. This significantly improves performance and reduce the Hadoop cluster workload comparing to the multiple insert case.

Below is an example of loading data to all country partitions using one insert statement:

```
    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.country
```

There are several syntactic differences from the multi-insert statement:

- country appears in the PARTITION specification, but with no value associated. In this case, country is a *dynamic partition column*. On the other hand, ds has a value associated with it, which means it is a *static partition column*. If a column is dynamic partition column, its value will be coming from the input column. Currently we only allow dynamic partition columns to be the last column(s) in the partition clause because the partition column order indicates its hierarchical order (meaning dt is the root partition, and country is the child partition). You cannot specify a partition clause with (dt, country='US') because that means you need to update all partitions with any date and its country sub-partition is 'US'.
- An additional pvs.country column is added in the select statement. This is the corresponding input column for the dynamic partition column. Note that you do not need to add an input column for the static partition column because its value is already known in the PARTITION clause. Note that the dynamic partition values are selected by ordering, not name, and taken as the last columns from the select clause.

Semantics of the dynamic partition insert statement:

- When there are already non-empty partitions exists for the dynamic partition columns, (for example, country='CA' exists under some ds root partition), it will be overwritten if the dynamic partition insert saw the same value (say 'CA') in the input data. This is in line with the 'insert overwrite' semantics. However, if the partition value 'CA' does not appear in the input data, the existing partition will not be overwritten.
- Since a Hive partition corresponds to a directory in HDFS, the partition value has to conform to the HDFS path format (URI in Java). Any character having a special meaning in URI (for example, '%', ':', '/', '#') will be escaped with '%' followed by 2 bytes of its ASCII value.
- If the input column is a type different than STRING, its value will be first converted to STRING to be used to construct the HDFS path.
- If the input column value is NULL or empty string, the row will be put into a special partition, whose name is controlled by the hive parameter hive.exec.default.partition.name. The default value is `HIVE_DEFAULT_PARTITION{}`. Basically this partition will contain all "bad" rows whose value are not valid partition names. The caveat of this approach is that the bad value will be lost and is replaced by `HIVE_DEFAULT_PARTITION{}` if you select them Hive. JIRA HIVE-1309 is a solution to let user specify "bad file" to retain the input partition column values as well.
- Dynamic partition insert could potentially be a resource hog in that it could generate a large number of partitions in a short time. To get yourself buckled, we define three parameters:
    - **hive.exec.max.dynamic.partitions.pernode** (default value being 100) is the maximum dynamic partitions that can be created by each mapper or reducer. If one mapper or reducer created more than that the threshold, a fatal error will be raised from the mapper/reducer (through counter) and the whole job will be killed.
    - **hive.exec.max.dynamic.partitions** (default value being 1000) is the total number of dynamic partitions could be created by one DML. If each mapper/reducer did not exceed the limit but the total number of dynamic partitions does, then an exception is raised at the end of the job before the intermediate data are moved to the final destination.
    - **hive.exec.max.created.files** (default value being 100000) is the maximum total number of files created by all mappers and reducers. This is implemented by updating a Hadoop counter by each mapper/reducer whenever a new file is created. If the total number is exceeding hive.exec.max.created.files, a fatal error will be thrown and the job will be killed.

- Another situation we want to protect against dynamic partition insert is that the user may accidentally specify all partitions to be dynamic partitions without specifying one static partition, while the original intention is to just overwrite the sub-partitions of one root partition. We define another parameter hive.exec.dynamic.partition.mode=strict to prevent the all-dynamic partition case. In the strict mode, you have to specify at least one static partition. The default mode is strict. In addition, we have a parameter hive.exec.dynamic.partition=true/false to control whether to allow dynamic partition at all. The default value is false prior to Hive 0.9.0 and true in Hive 0.9.0 and later.

- In Hive 0.6, dynamic partition insert does not work with hive.merge.mapfiles=true or hive.merge.mapredfiles=true, so it internally turns off the merge parameters. Merging files in dynamic partition inserts are supported in Hive 0.7 (see JIRA HIVE-1307 for details).

Troubleshooting and best practices:

- As stated above, there are too many dynamic partitions created by a particular mapper/reducer, a fatal error could be raised and the job will be killed. The error message looks something like:

```
    beeline> set hive.exec.dynamic.partition.mode=nonstrict;
    beeline> FROM page_view_stg pvs
          INSERT OVERWRITE TABLE page_view PARTITION(dt, country)
                 SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip,
                        from_unixtimestamp(pvs.viewTime, 'yyyy-MM-dd') ds, pvs.country;
...
2010-05-07 11:10:19,816 Stage-1 map = 0%,  reduce = 0%
[Fatal Error] Operator FS_28 (id=41): fatal error. Killing the job.
Ended Job = job_201005052204_28178 with errors
...
```

  The problem of this that one mapper will take a random set of rows and it is very likely that the number of distinct (dt, country) pairs will exceed the limit of hive.exec.max.dynamic.partitions.pernode. One way around it is to group the rows by the dynamic partition columns in the mapper and distribute them to the reducers where the dynamic partitions will be created. In this case the number of distinct dynamic partitions will be significantly reduced. The above example query could be rewritten to:

```
    beeline> set hive.exec.dynamic.partition.mode=nonstrict;
    beeline> FROM page_view_stg pvs
          INSERT OVERWRITE TABLE page_view PARTITION(dt, country)
                 SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip,
                        from_unixtimestamp(pvs.viewTime, 'yyyy-MM-dd') ds, pvs.country
                 DISTRIBUTE BY ds, country;
```

  This query will generate a MapReduce job rather than Map-only job. The SELECT-clause will be converted to a plan to the mappers and the output will be distributed to the reducers based on the value of (ds, country) pairs. The INSERT-clause will be converted to the plan in the reducer which writes to the dynamic partitions.

Additional documentation:

- Design Document for Dynamic Partitions
  - Original design doc
  - HIVE-936
- Hive DML: Dynamic Partition Inserts
- HCatalog Dynamic Partitioning
  - Usage with Pig
  - Usage from MapReduce

Inserting into Local Files

In certain situations you would want to write the output into a local file so that you could load it into an excel spreadsheet. This can be accomplished with the following command:

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/pv_gender_sum'
SELECT pv_gender_sum.*
FROM pv_gender_sum;
```

Sampling

The sampling clause allows the users to write queries for samples of the data instead of the whole table. Currently the sampling is done on the columns that are specified in the CLUSTERED BY clause of the CREATE TABLE statement. In the following example we choose 3rd bucket out of the 32 buckets of the pv_gender_sum table:

```
INSERT OVERWRITE TABLE pv_gender_sum_sample
SELECT pv_gender_sum.*
FROM pv_gender_sum TABLESAMPLE(BUCKET 3 OUT OF 32);
```

In general the TABLESAMPLE syntax looks like:

```
TABLESAMPLE(BUCKET x OUT OF y)
```

y has to be a multiple or divisor of the number of buckets in that table as specified at the table creation time. The buckets chosen are determined if bucket_number module y is equal to x. So in the above example the following tablesample clause

```
TABLESAMPLE(BUCKET 3 OUT OF 16)
```

would pick out the 3rd and 19th buckets. The buckets are numbered starting from 0.

On the other hand the tablesample clause

```
TABLESAMPLE(BUCKET 3 OUT OF 64 ON userid)
```

would pick out half of the 3rd bucket.

Union All

The language also supports union all, for example, if we suppose there are two different tables that track which user has published a video and which user has published a comment, the following query joins the results of a union all with the user table to create a single annotated stream for all the video publishing and comment publishing events:

```
INSERT OVERWRITE TABLE actions_users
SELECT u.id, actions.date
FROM (
    SELECT av.uid AS uid
    FROM action_video av
    WHERE av.date = '2008-06-03'

    UNION ALL

    SELECT ac.uid AS uid
    FROM action_comment ac
    WHERE ac.date = '2008-06-03'
    ) actions JOIN users u ON(u.id = actions.uid);
```

Array Operations

Array columns in tables can be as follows:

```
CREATE TABLE array_table (int_array_column ARRAY<INT>);
```

Assuming that pv.friends is of the type ARRAY<INT> (i.e. it is an array of integers), the user can get a specific element in the array by its index as shown in the following command:

```
SELECT pv.friends[2]
FROM page_views pv;
```

The select expression gets the third item in the pv.friends array.

The user can also get the length of the array using the size function as shown below:

```
SELECT pv.userid, size(pv.friends)
FROM page_view pv;
```

Map (Associative Arrays) Operations

Maps provide collections similar to associative arrays. Such structures can only be created programmatically currently. We will be extending this soon. For the purpose of the current example assume that pv.properties is of the type map<String, String> i.e. it is an associative array from strings to string. Accordingly, the following query:

```
 INSERT OVERWRITE page_views_map
 SELECT pv.userid, pv.properties['page type']
 FROM page_views pv;
```

can be used to select the 'page_type' property from the page_views table.

Similar to arrays, the size function can also be used to get the number of elements in a map as shown in the following query:

```
SELECT size(pv.properties)
FROM page_view pv;
```

Custom Map/Reduce Scripts

Users can also plug in their own custom mappers and reducers in the data stream by using features natively supported in the Hive language. for example, in order to run a custom mapper script - map_script - and a custom reducer script - reduce_script - the user can issue the following command which uses the TRANSFORM clause to embed the mapper and the reducer scripts.

Note that columns will be transformed to string and delimited by TAB before feeding to the user script, and the standard output of the user script will be treated as TAB-separated string columns. User scripts can output debug information to standard error which will be shown on the task detail page on hadoop.

```
FROM (
     FROM pv_users
     MAP pv_users.userid, pv_users.date
     USING 'map_script'
     AS dt, uid
     CLUSTER BY dt) map_output

 INSERT OVERWRITE TABLE pv_users_reduced
     REDUCE map_output.dt, map_output.uid
     USING 'reduce_script'
     AS date, count;
```

Sample map script (weekday_mapper.py)

```
import sys
import datetime

for line in sys.stdin:
  line = line.strip()
  userid, unixtime = line.split('\t')
  weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()
  print ','.join([userid, str(weekday)])
```

Of course, both MAP and REDUCE are "syntactic sugar" for the more general select transform. The inner query could also have been written as such:

```
 SELECT TRANSFORM(pv_users.userid, pv_users.date) USING 'map_script' AS dt, uid CLUSTER BY dt FROM pv_users;
```

Schema-less map/reduce: If there is no "AS" clause after "USING map_script", Hive assumes the output of the script contains 2 parts: key which is before the first tab, and value which is the rest after the first tab. Note that this is different from specifying "AS key, value" because in that case value will only contains the portion between the first tab and the second tab if there are multiple tabs.

In this way, we allow users to migrate old map/reduce scripts without knowing the schema of the map output. User still needs to know the reduce output schema because that has to match what is in the table that we are inserting to.

```
FROM (
    FROM pv_users
    MAP pv_users.userid, pv_users.date
    USING 'map_script'
    CLUSTER BY key) map_output

INSERT OVERWRITE TABLE pv_users_reduced

    REDUCE map_output.dt, map_output.uid
    USING 'reduce_script'
    AS date, count;
```

Distribute By and Sort By: Instead of specifying "cluster by", the user can specify "distribute by" and "sort by", so the partition columns and sort columns can be different. The usual case is that the partition columns are a prefix of sort columns, but that is not required.

```
FROM (
    FROM pv_users
    MAP pv_users.userid, pv_users.date
    USING 'map_script'
    AS c1, c2, c3
    DISTRIBUTE BY c2
    SORT BY c2, c1) map_output

INSERT OVERWRITE TABLE pv_users_reduced

    REDUCE map_output.c1, map_output.c2, map_output.c3
    USING 'reduce_script'
    AS date, count;
```

Co-Groups

Amongst the user community using map/reduce, cogroup is a fairly common operation wherein the data from multiple tables are sent to a custom reducer such that the rows are grouped by the values of certain columns on the tables. With the UNION ALL operator and the CLUSTER BY specification, this can be achieved in the Hive query language in the following way. Suppose we wanted to cogroup the rows from the actions_video and action_comments table on the uid column and send them to the 'reduce_script' custom reducer, the following syntax can be used by the user:

```
FROM (
     FROM (
             FROM action_video av
             SELECT av.uid AS uid, av.id AS id, av.date AS date

            UNION ALL

             FROM action_comment ac
             SELECT ac.uid AS uid, ac.id AS id, ac.date AS date
     ) union_actions
     SELECT union_actions.uid, union_actions.id, union_actions.date
     CLUSTER BY union_actions.uid) map

 INSERT OVERWRITE TABLE actions_reduced
     SELECT TRANSFORM(map.uid, map.id, map.date) USING 'reduce_script' AS (uid, id, reduced_val);
```