Quick look: Apache Hive

Hive is a data-warehousing infrastructure based on Hadoop. Hadoop provides massive scale out and fault tolerance capabilities for data storage and processing (using the map-reduce programming paradigm) on commodity hardware.

Hive is designed to enable easy data summarization, ad-hoc querying and analysis of large volumes of data.

Latency for Hive queries is generally very high (minutes) even when data sets involved are very small (say a few hundred megabytes). As a result it cannot be compared with systems such as Oracle where analyses are conducted on a significantly smaller amount of data but the analyses proceed much more iteratively with the response times between iterations being less than a few minutes.

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

Data Units

Databases: Namespaces that separate tables and other data units from naming confliction.

Tables: Homogeneous units of data which have the same schema.

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 certain criteria.

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.

Data Types

Primitive types:

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

Complex types:

- **Structs:** the elements within the type can be accessed using the DOT (.) notation.
- **Maps** (key-value tuples): The elements are accessed using ['element name'] notation.
- **Arrays** (indexable lists): The elements in the array have to be in the same type.

Built In Operations

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 semi-colon, 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	**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.
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	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

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 e.g. 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 e.g. 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 e.g for struct foobar {int foo, int bar} foobar.foo returns the integer stored in the foo field of the struct.

Built In Functions

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 e.g. substr('foobar', 4, 2) results in 'ba'
string	upper(string A)	returns the string resulting from converting all characters of A to upper case e.g. 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 e.g. 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 e.g. 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> e.g. 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 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

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

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;
```

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;
```

Browsing Tables and Partitions

```
SHOW TABLES;

SHOW TABLES 'page.*';

SHOW PARTITIONS page_view;

DESCRIBE page_view;

DESCRIBE EXTENDED page_view;

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

Altering Tables

```
ALTER TABLE old_table_name RENAME TO new_table_name;

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

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

Dropping Tables and Partitions

```
   DROP TABLE pv_users;

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

Loading Data

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';
```

The system also supports syntax that can load the data from a file in the local files system directly into a Hive table

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

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

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

Querying and Inserting Data

Simple Query

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

Partition Based 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';
```

Joins

```
a) 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';


b) 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

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

b) 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;
```

Multi Table/File Inserts

```
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;
```

Sampling

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