



Chapter 4: HiveQL



In this chapter, you will learn

- What is HiveQL?
- The basic SELECT syntax
- How to use functions and aggregates
- What complex types are supported
- How to use custom map/reduce scripts
- The syntax for subqueries and joins
- How to use the Hive INSERT



What is HiveQL?

- A subset of SQL-92
 - Joins
 - Aggregates
 - Subqueries (in the FROM clause)
 - Etc..
- Plus useful extensions
 - Partitioning and sampling
 - Complex data structures
 - Pluggable map/reduce scripts
 - Multi-table inserts



The Hive Query Language (HiveQL) is a subset of SQL-92 plus useful extensions. Like SQL, HiveQL supports:

- joining tables
- aggregates such as `sum` and `count`
- subqueries (though only in the `FROM` clause)
- sorting
- filtering
- unions

Some of the extensions that Hive offers are:

- partitioning and sampling
- complex data structures (arrays, maps and structs)
- the ability to use custom map and reduce scripts in a query
- multi-table inserts

SQL/OLTP vs. HiveQL/Data Warehouse

	SQL	HiveQL
Update Capabilities	INSERT, UPDATE, and DELETE	INSERT OVERWRITE; no UPDATE or DELETE
Transactions	Supported	Not supported
Latency	Sub-second	Minutes or more
Data types	Int, Double, Char, Varchar, Date, Time,...	Int, Double, String, Map, Array, Struct (no date or time)
Language	Correlated and Non-correlated subqueries in any clause; Inner and outer joins; May vary across vendors	Only supports subqueries in FROM clause; Inner and outer joins; LIMIT supported; No HAVING clause
Extensibility	Stored procedures, User-defined functions	UDF, UDAF, custom MapReduce scripts



HiveQL is not identical to SQL. A major difference is that HiveQL is meant to read data from Hadoop/HDFS. Since HDFS cannot do inline edits to data, UPDATE and DELETE are not supported. In order to change data, it is necessary to transform the data in a table into a new table. Similarly, Hive does not offer transactional support. Due to the time involved to set up a MapReduce job, HiveQL will not respond to simple queries as quickly as an RDBMS. The minimum MapReduce job will take many seconds or minutes depending on the amount of data and the complexity of the job. Hive is optimized for batch processing, not latency.

HiveQL supports many of the data types found in SQL, with the exception of temporal types: date, time, timestamp. In addition to those data types, HiveQL supports three complex types: map, array and struct.

The language that Hive uses is similar to SQL in many ways. At this time, subqueries are supported, but only in the FROM clause. Inner, outer and full outer joins are supported. HiveQL supports the non-standard LIMIT clause that is found in some RDBMSes. The HAVING keyword is not available at this time.

HiveQL offers many built-in functions and it is also possible to add user-defined functions and user-defined aggregate functions. A useful feature that Hive supports is the ability to write custom map or reduce scripts in a variety of scripting languages.

A basic SELECT

- `SELECT expr, expr, ..`
`FROM tablename;`



A basic `SELECT` statement in HiveQL takes the form of:

```
SELECT expr, expr FROM tablename;
```

In the above example, "expr" could be a column or other expression such as a function call. The `FROM` clause is required and indicates which table or tables the query is reading data from. To those familiar with SQL, the syntax of HiveQL will seem similar.

Case sensitivity: Most aspects of the Hive language are case-insensitive (e.g., keywords, identifiers, and functions). Capitalizing keywords is a convention but not required. String comparisons are case-sensitive.

The WHERE clause

- Any boolean expression
- Examples:
 - `countrycode = 'US'`
 - `amount > 42`
 - `url LIKE '%.example.com'`
 - `item IS NULL`
- Expressions can be combined with AND/OR



The WHERE clause is a boolean expression that acts as a filter on the input data.

Supported operators:

`=, <, >, <=, >=, <>, !=, IS NULL, IS NOT NULL, LIKE, RLIKE`

Multiple expressions can be evaluated such as:

```
WHERE fname = 'Tom' AND lname = 'White'
```

Parenthesis may be used to control operator precedence.

ORDER BY

- Sorts the rows in ascending order (by default)
 - For descending: `ORDER BY expr DESC`
- Multiple expressions may be specified
 - `ORDER BY surname, firstname`
- Useful with `LIMIT` for “top-n” queries
 - `ORDER BY price LIMIT 10`



The `ORDER BY` clause causes the results to be ordered by the column or columns specified. By default the ordering is ascending, unless `DESC` is included for descending order. The sort ordering is based on the type of the data (integers will sort numerically, strings will sort lexicographically). When multiple columns are given, the results are ordered by the first column (either ascending or descending). If two values are equal, they are then sorted by the second column (which can be either ascending or descending).

`ORDER BY` is very commonly used with `LIMIT` to provide “top-n” queries. In order to find the 10 least expensive items, you would order the rows by price (ascending) and then limit the results to 10: `ORDER BY price LIMIT 10;`

It is also valid to use column aliases in the `ORDER BY`: `SELECT column1 AS c FROM tablename ORDER BY c;`

SORT BY

- Like ORDER BY, but sorts per reducer
- Can be used with DISTRIBUTED BY
- `SELECT * FROM purchases`

`DISTRIBUTE BY custid`

`SORT BY cost DESC;`



The ORDER BY requires a single reducer to provide an overall sort. This may not scale in some cases. Hive supports a SORT BY clause (not part of standard SQL). The SORT BY will sort within each reducer. This does not return an overall sorted order, but the results of each reducer could be merged to provide a sorted result.

To control which reducer a particular set of rows goes to, use DISTRIBUTE BY. For example,

```
SELECT * FROM purchases
DISTRIBUTE BY custid
SORT BY cost DESC;
```

Depending on the number of reducers, this will produce a set of sorted files where each one contains some custids and is sorted by cost (descending).

There is also a "CLUSTER BY" clause which is shorthand for "DISTRIBUTE BY" and "SORT BY" the same key.

Built-in functions

- Hive provides many built-in functions, such as:

Numeric functions	String functions	Date functions	Other
round(double)	length(string)	from_unixtime(int)	size(Map)
rand()	reverse(string)	unix_timestamp()	size(Array)
pow(double, double)	concat(string,string..)	year(string)	cast(expr as type)
sqrt(double)	substr(string,int)	month(string)	if
exp(double)	upper(string)	minute(string)	case
ln(double)	lower(string)	weekofyear(string)	get_json_object
log10(double)	trim(string)	datediff(string,string)	parseUrl
log2(double)	split(string,string)	date_add(string,int)	Xpath*



Hive supports a good number of built-in functions (though not as many as most RDBMSs). For a full list of Hive's functions, see http://wiki.apache.org/hadoop/Hive/LanguageManual/UDF#Built-in_Functions or issue `SHOW FUNCTIONS`.

Use `DESCRIBE` to see sample usage information:

```
hive> DESCRIBE FUNCTION length;
```

Note: although Hive does not support temporal types like `DATE`, there are functions for working with integers (Unix epoch) and strings that represent temporal information.

* Xpath functions are coming in Hive 0.6

Aggregate functions

- Special functions that work on a collection of values and return an aggregated result
 - `count(1)`
 - `sum(col)`
 - `avg(col)`
 - `min(col)`
 - `max(col)`
 - `percentile(col)`
- ```
SELECT count(1), sum(cost)
FROM purchases;
```



There are 6 special built-in functions that aggregate values into a single result. The `count` function can be used two ways: `count(1)` returns the number of items in the group; `count(DISTINCT column)` returns the number of distinct values for column. Summing numeric values can be done with `sum`. Likewise `avg` can be used to find an average (mean). `sum` and `avg` can use `DISTINCT` as well. `max` and `min` return the largest and smallest value in the group respectively.

This example would count the number of purchases in the `purchases` table as well as calculate a total cost:

```
SELECT count(1), sum(cost)
FROM purchases;
```

**Note:** while `count`, `sum` and `avg` can use `DISTINCT`, a query cannot contain two aggregate functions that apply `DISTINCT` to different columns. E.g., this is allowed:

```
SELECT count(DISTINCT foo_col), sum(DISTINCT foo_col)...
```

But this is not allowed:

```
SELECT count(DISTINCT foo_col), sum(DISTINCT bar_col)...
```

**Note:** `count(*)` is not supported in HiveQL

## Grouping rows

- The GROUP BY clause groups records with a similar value
- Aggregate functions may be applied to the groups:

```
SELECT custid, count(1), sum(cost)
FROM purchases
GROUP BY custid;
```



The GROUP BY clause groups records with a similar value, such as grouping purchases of each unique customer. Aggregate functions may then be applied to all records in the group.

Once rows are grouped, the only meaningful operations are aggregates. It is important to realize that other columns cannot be selected. For example, this query would return an **error** because there are many order\_date values within a particular group:

```
SELECT custid, count(1), sum(cost), order_date
FROM purchases
GROUP BY custid;
```

The GROUP BY comes after the WHERE clause, but before an ORDER BY.

Note: HAVING is not supported yet (see Subquery section for a workaround).

## Using complex data types

- `CREATE TABLE t (id INT, letters ARRAY<STRING>)...`
- `SELECT * FROM t;`

| id | letters           |
|----|-------------------|
| 1  | ["a","b","c"]     |
| 2  | ["d","e"]         |
| 3  | ["f","g","h","i"] |



Querying a table that contains an array will return the data in a flattened column for all the elements in the array.

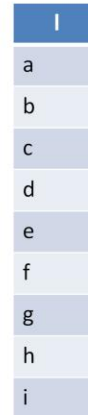
If the table has a map structure, a query would return something like this:

```
{"key1":"foo","key2":"bar"}
```

```
{"key1":"baz"}
```

## explode(array)

- Explode takes an array and makes each element its own row
- ```
SELECT explode(letters) AS l
FROM t;
```



l
a
b
c
d
e
f
g
h
i

cloudera

The explode function takes a single argument, an array, and outputs a row for each element of the array. A column alias is required.

This is an example of a user-defined table-generating function (UDTF). Unlike standard UDFs, UDTFs take a single input and produce multiple outputs.

Type	Input	Output	Example
UDF	single row	single row	SELECT length(firstname)
UDAF	multiple rows	single row	SELECT count(custid)
UDTF	single row	multiple rows	SELECT explode(my_array)

Currently `explode` is the only UDTF in Hive. It is possible to write UDFs, UDAFs or UDTFs in Java and use them in Hive.

LATERAL VIEW

- `SELECT id, l FROM t LATERAL VIEW
explode(letters) lets AS l;`

Original table:

id	letters
1	["a","b","c"]
2	["d","e"]
3	["f","g","h","i"]

id	l
1	a
1	b
1	c
2	d
2	e
3	f
3	g
3	h
3	i

cloudera

The explode function is useful along with `LATERAL VIEW`, which joins the rows from the base table to the output of the UDTF. It applies the function to each row of the base table and then does the join.

Think of the output of the `LATERAL VIEW` as a table (the alias "lets" is a table alias and "l" is a column alias). In the case of `explode`, it returns one column. Each result of the `LATERAL VIEW` is then joined to the original rows of the table.

Custom map/reduce scripts

- Pluggable map/reduce scripts similar to Hadoop Streaming
- Read rows from `stdin` with tabs between columns
- Write results to `stdout` with tabs between columns
- Distribute the script to the cluster with `ADD FILE`



Hive allows custom map or reduce scripts to be plugged into a query. Hive invokes these scripts similar to how Hadoop Streaming works. The script should read rows from standard input (`stdin`) as tab-separated column values. Likewise the script should write rows to standard output (`stdout`) as tab-separated column values.

Each machine in the cluster will need access to the script. This can be accomplished with `ADD FILE`, which loads the script to the distributed cache.

Custom map script

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
    (col1, col2) = line.strip().split("\t")
    # your logic here
    print output1 + "\t" + output2
```



A custom map or reduce script could be written in nearly any language (the requirement is that it can read stdin and write to stdout). The input and output is a tab-separated string.

An example could be a table that stored HTML documents and a custom map script could extract all links from each document. For another example of using custom map/reduce scripts with Hive, see

<http://www.cloudera.com/blog/2009/09/grouping-related-trends-with-hadoop-and-hive/>

Invoking custom map script

- `ADD FILE /tmp/mapper.py;`
- ```
INSERT OVERWRITE TABLE result
 SELECT transform(t.*)
 USING './mapper.py' AS(col1, col2)
FROM
 (SELECT ...) t;
```



To invoke a custom map script, use the `map`, `reduce` or `transform` function. The script must be available to the nodes invoking the map and reduce tasks, so put the file in the distributed cache:

```
ADD FILE /tmp/mapper.py;
```

The input to the script can be any `SELECT` output (given table alias "t" in the example). The `transform` function specifies which columns should be sent to the script. The `USING` clause specifies the name of the map script and the columns to receive the output.

## Subqueries

- Currently supported in the FROM clause only
- Useful for implementing “HAVING” functionality

| Not Supported                                                                                      | Supported                                                                                                                                  |
|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>SELECT custid, sum(cost) AS total FROM purchases GROUP BY custid HAVING total &gt; 100;</pre> | <pre>SELECT custid, total FROM   (SELECT custid, sum(cost) AS total    FROM purchases    GROUP BY custid) subq WHERE total &gt; 100;</pre> |



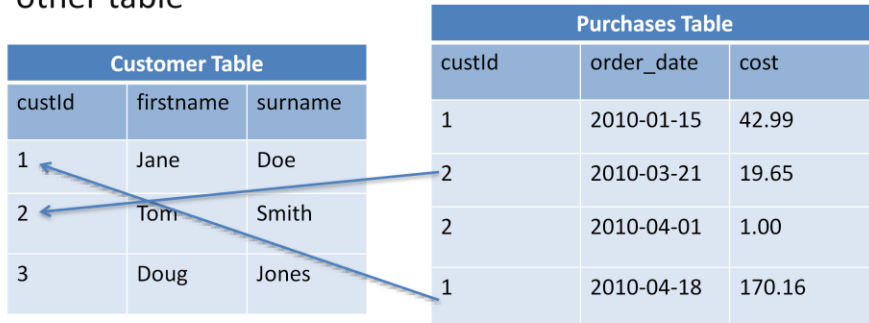
Hive has support for a subquery, or nested `SELECT`, in the `FROM` clause of a query. This is sometimes called an inline view. This can be very useful for implementing a filter after a `GROUP BY`. Since a `WHERE` clause filters results before they are grouped, it is not useful for filtering the aggregated results. In standard SQL this would be accomplished with a `HAVING` clause. `HAVING` is not available in Hive yet, but the same effect can be achieved with a subquery:

```
SELECT custid, total
FROM
 (SELECT custid, sum(cost) AS total
 FROM purchases
 GROUP BY custid) subq
WHERE total > 100;
```

**Note:** the table alias ("subq") is required.

## Joining tables

- To query data from more than one table, the rows need to be “joined”
- Typically some value in one table refers to data in the other table



| Customer Table |           |         |
|----------------|-----------|---------|
| custId         | firstname | surname |
| 1              | Jane      | Doe     |
| 2              | Tom       | Smith   |
| 3              | Doug      | Jones   |

| Purchases Table |            |        |
|-----------------|------------|--------|
| custId          | order_date | cost   |
| 1               | 2010-01-15 | 42.99  |
| 2               | 2010-03-21 | 19.65  |
| 2               | 2010-04-01 | 1.00   |
| 1               | 2010-04-18 | 170.16 |

cloudera

A join is the concept of querying data from more than one table, where the rows from one table are “joined” to the rows of another table. Typically there is a meaningful relationship that defines how the rows should be joined. In the above example, the Purchases table is referencing the customer id (custid) in the Customer table.

## Inner joins

- An inner join returns results if there is a match
- Find customers who are also in the purchases table:

```
SELECT *
FROM customer JOIN purchases
ON (customer.custid = purchases.custid);
```

| custid | firstname | surname | custid | order_date | cost   |
|--------|-----------|---------|--------|------------|--------|
| 1      | Jane      | Doe     | 1      | 2010-01-15 | 42.99  |
| 1      | Jane      | Doe     | 1      | 2010-04-18 | 170.16 |
| 2      | Tom       | Smith   | 2      | 2010-03-21 | 19.65  |
| 2      | Tom       | Smith   | 2      | 2010-04-01 | 1.00   |



An inner join only returns rows that have a match in both tables. For example, Jane, whose custid is 1, has a couple of records in the purchases table. An inner join would return Jane “joined” with her purchases. If a customer, Doug, had not made any purchases, he would not be returned.

It is necessary to qualify column names that are ambiguous, such as custid, since it exists in both tables. Table aliases can also be used.

It is possible to join more than two tables. The keywords JOIN and ON are required for each additional table:

```
SELECT columns
FROM a JOIN b ON (a.col = b.col)
JOIN c ON (c.col = a.col);
```

**Note:** only equi-joins are supported. Non-equality comparisons cannot be used in the ON clause.

Comma join syntax (available in some RDBMSes) is not supported.

## Outer joins

- An outer join includes rows in the result that did not have a match
- ```
SELECT C.custid, firstname, surname, order_date
FROM customer C LEFT OUTER JOIN purchases P
ON (C.custid = P.custid);
```

custid	firstname	surname	order_date
1	Jane	Doe	2010-01-15
1	Jane	Doe	2010-04-18
2	Tom	Smith	2010-03-21
2	Tom	Smith	2010-04-01
3	Doug	Jones	NULL



An outer join is like an inner join, but also includes the rows that did not have a match. For example, Doug did not have any purchase records.

- `LEFT OUTER JOIN` will return the non-matching rows from the left table
- `RIGHT OUTER JOIN` returns the non-matching rows from the right table
- `FULL OUTER JOIN` returns both

For rows that did not have a match in the joined table, there will be NULLs in all columns of the non-matching table.

Note: This example uses table aliases. “C” is an alias for the customer table as declared in the `FROM` clause. An `AS` keyword is not allowed.

Identifying unmatched records

- Outer joins are useful for finding unmatched records
- Example: find customers who have not made purchases

```
SELECT C.custid, firstname, surname
FROM customer C LEFT OUTER JOIN purchases P
ON (C.custid = P.custid)
WHERE P.custid IS NULL;
```

custid	firstname	surname
3	Doug	Jones



In the previous example, Doug was included in the result set despite not having any purchases. It is common to use an outer join for identifying the unmatched records. If you want to return *only* the rows that did not have a match in the joined table, use a `WHERE` to look for the `NULL`s. The `WHERE` clause is processed after the join.

Note: Be sure to use the `"IS NULL"` operator, not `"= NULL"`

INSERT OVERWRITE

- Various uses:
 - Insert data into a Hive table
 - Extract data by inserting into an external table
 - Extract data by writing to a directory



The INSERT OVERWRITE command has various uses in Hive. It can:

- Insert the results of a query into a Hive table. The table must already exist, otherwise use `CREATE TABLE .. AS SELECT`
- Extract data from Hive table(s) into an `EXTERNAL` table whose data is in HDFS
- Extract data from Hive table(s) into files in a directory (either HDFS or the local filesystem)

Inserting data into a table

- `INSERT OVERWRITE TABLE table_foo`
`SELECT * FROM table_bar;`
- `OVERWRITE` is required
- Optionally specify a partition



It is possible to insert rows into a table by selecting from an existing table. However, the `OVERWRITE` keyword is required. The inserted rows replace any prior data in this table. The `SELECT` can be any valid `SELECT` statement in Hive.

A partition expression can be included in which case only that partition is written to. This is a convenient way to append data to a table.

Extract data by inserting into **EXTERNAL** table

- Create an EXTERNAL table with any storage format:
- ```
CREATE EXTERNAL TABLE table (columns)
 ROW FORMAT DELIMITED FIELDS
 TERMINATED BY ','
 STORED AS TEXTFILE
 LOCATION '/hdfs_path';
```
- ```
INSERT OVERWRITE TABLE table SELECT ...
```



The `INSERT OVERWRITE` command can be used to extract data from Hive tables. One technique uses an `EXTERNAL` table. An advantage to this approach is the ability to control the storage format (e.g., `ROW FORMAT` and `STORED AS` clauses). The `INSERT OVERWRITE` can use any legal `SELECT` statement such as joins and functions.

Remember that dropping external tables does not remove the underlying files in HDFS.

Extracting data from Hive

- `INSERT OVERWRITE [LOCAL] DIRECTORY /path/dir
SELECT...`
 - Output format uses text files with ^A column separator



Hive also has the capability to write to the filesystem. The `INSERT . . SELECT` syntax is the same as before, but use the keyword `DIRECTORY` instead of `TABLE`. Like before, the `OVERWRITE` keyword is required (an error will result if the Hive does not have privileges to overwrite this directory). If the keyword `LOCAL` is added, then the data is transferred to the client's local filesystem, otherwise the path is interpreted as an HDFS location. The output format will be a text file(s) with ^A (ctrl-A) as a column separator and newline as a row separator. If the query returns non-primitive data types such as map or arrays, then those are stored in JSON format.

Extension to SQL - “multi-table insert”

- Multiple inserts that share a FROM clause
- Efficiently stream source data a single time

- FROM *tablename alias*

```
INSERT OVERWRITE TABLE t2 SELECT alias.col1
```

```
INSERT OVERWRITE TABLE t3 SELECT alias.col2;
```

- FROM (SELECT...) *alias*

```
INSERT OVERWRITE TABLE..
```

```
INSERT OVERWRITE TABLE..;
```



Hive extends the SQL standard to support multi-table insert statements. This allows data to be scanned a single time and used multiple times which is much more efficient than streaming the data each time it is used. The FROM clause can be a `tablename` or a `SELECT` statement. A table alias is required and is used in the later queries to reference the source data. The `INSERT` statements can either output to tables or directories. There may be any number of `INSERT` statements. They do not use any separator.

Example: list and count documentary movies

```
FROM (  
  SELECT movie_name  
    FROM Movies WHERE documentary = 1) m  
INSERT OVERWRITE DIRECTORY  
  'count_doc_movies' SELECT count(1)  
INSERT OVERWRITE DIRECTORY  
  'documentary_movies' SELECT m.*;
```



This example queries the documentary movies and processes them in two ways: counting them and listing them. The input data (movies which belong to category "Documentary") is scanned a single time and processed into two outputs.

The `SELECT` produces a set of rows which are given a table alias "m". The subsequent `INSERT` statements may use any of the data from "m".

Conclusion

In this chapter, you have learned:

- What is HiveQL?
- The basic SELECT syntax
- How to use functions and aggregates
- What complex types are supported
- How to use custom map/reduce scripts
- The syntax for subqueries and joins
- How to use the Hive INSERT

