Hive Cheat Sheet

A quick reference for the most useful *Hive* commands and *HiveQL* features. Note that some of the output shown, etc. will vary depending on which Hadoop distribution and Hive version you are using and how it is configured in your environment.

See the Hive wiki for more details not covered here.

The end of these notes list some differences between *HiveQL*, Oracle SQL, and ANSI SQL.

The Hive Installation on Our VM

In Amazon AWS 5.8.1, Hive lives in `/usr/lib/hive`. This directory contains many files you'll want to know about

Directory	*Important* Contents	Description	
`bin`	`hive`	The driver for all Hive processes: `hive -h` (help) and `hive` (CLI)	
`bin/ext`	`cli.sh`	The CLI process: `hive` or `hive --service cli` (All `bin/ext` scripts are invoked through `hive --service ...`)	
`bin/ext`	`help.sh`	Help: `hive -h`, `hive --service help`, and `hive --service <service> --help` (all different)	
`bin/ext`	`hiveserver2.sh`	Run Hive as a server that permits *Thrift* client connections: `env HIVE_PORT=NNNN hive --service hiveserver`	
`bin/ext`	`metastore.sh`	Run a metastore server: `env HIVE_METASTORE=NNNN hive --service metastore`	
`conf`	`hive-default.xml`	All possible config variables (many inherited from Hadoop).	
`conf`	`hive-site.xml`	Place to override the defaults.	
`conf`	`hive-log4j.properties`	Log4J configuration.	
`examples`	*all*	Many HiveQL examples.	
`lib`	*all*	The core Hive libraries. Sometimes you will want to add custom jars here.	
`scripts`	`metastore/*`	Scripts for upgrading a metastore from an older Hive version.	

If you want documentation for Hive, we recommend referring to the official hive repository at `hive.apache.org`.

Invoking Hive

The Hive CLI

The `hive` script without any options invokes the CLI (command line interpreter). Here are the options for thie "service", shown by running `hive -h` (which is equivalent to `hive --service cli --help`).

```
usage: hive
 -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
 -i <filename>                    Initialization SQL file
 -S,--silent                      Silent mode in interactive shell
 -v,--verbose                     Verbose mode (echo executed SQL to the
                                  console)
```

The `-i <filename>` option is particularly useful for setting up an "initialization" file that does common setup HQL commands. In fact, with or without this option, Hive will automatically look for a `$HOME/.hiverc` file and run its commands, if found. This is a great place to add commands you run all the time, but may not make sense (or be possible

globally), such as system properties, adding jar files of extension code, etc.

The Hiveserver

The `hiveserver2` lets clients connect to Hive using the *Thrift* serialization protocol. Support exists for JDBC connections as well as from any other language for which there is *Thrift* support, such as C++.

Metastore

Normally in production you use a multiuser production database like *MySQL* or *Postgres* for the *metadata store* and you run a single server for the metastore, as opposed to running local instances on each machine running Hive.

A Simple Example Using the Hive CLI

Here is a simple session transcript, with some output removed for clarity, etc. The `$` on the first line is the `bash` prompt. The `hive` prompt is `hive>`. Lines without that prompt are command results. A blank line is added after each command result for clarity and keywords are shown in all caps. Like other SQL dialects, case is ignored for keywords, etc. (File paths are case sensitive.) For convenience, just the commands for this sample session are captured in `session.hql` in the Hive Walkthrough exercise folder.

```
$ hive
Hive history file=/tmp/thinkbig/hive_job_log_thinkbig_201112050148_1266834092.txt

hive> SHOW TABLES;
OK

hive> CREATE TABLE demo1 (id INT, name STRING);
OK

hive> SHOW TABLES;
OK
demo1

hive> DESCRIBE DEMO1;
OK
id  int
name    string

hive> ALTER TABLE demo1 ADD COLUMNS (boss STRING);
OK

hive> DESCRIBE DEMO1;
OK
id  int
name    string
boss    string

hive> DROP TABLE demo1;
OK
Time taken: 9.421 seconds
```

Here's a more sophisticated table that provides a sneak peak of what's to come...

```
hive> CREATE EXTERNAL TABLE shakespeare_wc (word STRING, count INT)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
    > LOCATION '/data/shakespeare/golden/simple-word-count/plain-text';
OK
Time taken: 0.308 seconds

hive> SELECT word FROM shakespeare_wc LIMIT 5;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
lackeys.
lackeys:
lacking
lacking--god
lacks

hive> SELECT * FROM shakespeare_wc WHERE word LIKE 'w%' LIMIT 5;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
w.  1
wad 1
waddled 1
wade    3
waded   2
hive> DROP TABLE shakespeare_wc;
OK
hive> exit;
```

The Hive Query Language (HiveQL or HQL)

While you can access Hive programmatically through JDBC, ODBC, and Thrift, we'll focus exclusively on accessing the Hive Query Language (HiveQL or HQL) through the CLI.

As before, when CLI examples are shown, they will follow the lecture examples and the exercises. Most of the output is suppressed, except where it is particularly informative.

Note: HiveQL doesn't have a comment convention, so there is no way to embed comments in `hql` files that you run with the `hive -f file` option!

Primitive Data Types

There are the usual built-in data types [1]. As always, case is ignored.

| 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` | `TRUE` | `FALSE` | `TRUE` |
| `FLOAT` | 4-byte single precision | `3.14159` |
| `DOUBLE` | 8-byte double precision | `3.14159` |
| `STRING` | Sequence of characters. The character set can be specified. | 'Now is the time', "for all good men" |
| `TIMESTAMP` | Integer, float or string | (Hive v0.8) `1327882394` (Unix epoch seconds), `1327882394.123456789` (Unix epoch seconds plus nanoseconds), and `'2012-02-03 12:34:56.123456789'` (JDBC-compliant `java.sql.Timestamp` format). |
| `BINARY` | Array of bytes | (Hive v0.8) see discussion below. |

Note that these types reflect the underlying types provided by Java.

`TIMESTAMP` and `BINARY` are new to Hive v0.8.0. For Hive v0.7.X, use an integer value for the epoch seconds or use a `STRING`. If you use the latter, it is best to use a sortable format like the example shown.

The `BINARY` type is a way of saying "ignore the rest of this record"; it is treated as a byte array without interpretation and with no built-in support for conversion to other types. Use it when you only care about the first few fields in a record, e.g.,

```
CREATE TABLE short (s STRING, i INT, b BINARY);
```

When a query mentions a particular type, Hive will implicitly cast any integer type to a larger integer type, cast `FLOAT` to `DOUBLE`, and cast any integer type to `DOUBLE`, as needed.

You can also explicitly interpret a `STRING` as a number type using, for example, `'3.14159' to DOUBLE`.

Complex Data Types

Hive supports columns that are `structs`, `maps`, and `arrays`. Note that the "literal syntax examples" are actually making calls to built-in functions.

| Type | Description | Literal Syntax Examples || | :------- | :---------- | :---------------------- | | `STRUCT` | Analogous to a C `struct` or an "object". Fields can be accessed using the "dot" notation. For example, if a column `name` if 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` | Lists of the same type that are indexable using zero-based integers. For example, if a column `name` is of type `ARRAY` of strings with the value `['John', 'Doe']`, then the second element can be referenced using `name[1]`. | `array('John', 'Doe')` |

File Formats

Hive supports all the Hadoop file formats, plus Thrift encoding, as well as supporting pluggable `SerDe` (serializer/deserializer) classes to support custom formats. Hive defaults to the following record and field delimiters, all of which are non-printable control characters and all of which can be customized.

Delimiter	Name	Use
`\n` or `\r`	Line feed or carraige return	Records, i.e., one per line.
`^A`	Control-A (`\001`)	Separates fields, e.g., the way commas are used in CSV files.
`^B`	Control-B (`\002`)	Separates elements in the "collections" (`STRUCT` , `MAP` , and `ARRAY`)
`^C`	Control-C (`\003`)	Separates the key and value in `MAP` elements.

All of these defaults can be customized when creating tables. See examples below.

There are several file formats supported by Hive. `TEXTFILE` is the easiest to use, but the least space efficient. Hadoop's `SEQUENCEFILE` format is more space efficient. A related format is `MAPFILE` which adds an index to a `SEQUENCEFILE` for faster retrieval of particular records.

Database (Schema) Management

Let's discuss database (a.k.a. schema) management and demonstrate the relationship between databases and tables. First, an example session:

```
hive> SHOW DATABASES;
OK
default

hive> CREATE DATABASE IF NOT EXISTS db1
  COMMENT 'Our database db1';
OK

hive> SHOW DATABASES;
OK
db1
default

hive> DESCRIBE DATABASE db1;
OK
db1     Our database db1    hdfs://localhost/user/hive/warehouse/db1.db

hive> SHOW TABLES;
OK

hive> CREATE TABLE db1.table1 (word STRING, count INT);
OK

hive> SHOW TABLES;
OK
table1

hive> DESCRIBE db1.table1;
FAILED: Parse Error: line 1:0 cannot recognize input near 'describe' 'table' 'db1' in describe statement

hive> DESCRIBE table1;
OK
word    string
count   int

hive> USE db1;
OK

hive> SHOW TABLES;
OK
table1

hive> SELECT * FROM db1.table1;
OK

hive> SELECT * FROM table1;
OK

hive> DROP TABLE table1;
OK

hive> DROP DATABASE db1;
OK

hive> USE default;
OK
```

Unfortunately, there is no built-in command to show the current database you are using! The only workaround for the 0.7.X versions of Hive is to repeat the `USE db1` command to ensure you are where you want to be. However, for version 0.8.X and newer, an alternative is to modify the `hive>` prompt to show the current database. Add the following to your `$HOME/.hiverc` file (or type the command at the hive prompt):

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

Creating and Deleting Tables

By default, created tables are *internal*, where Hive owns and manages the data, as well as the table metadata.

```
hive> CREATE TABLE demo1 (id INT, name STRING);
```

This command creates an internally-managed table with the data storage located in HDFS in the `<hive.metastore.warehouse.dir>/demo1` directory, where the `hive.metastore.warehouse.dir` property is configurable. We are using the default value of `/user/hive/warehouse` in HDFS. This is an otherwise ordinary HDFS directory. You can always use `hadoop -lsr /user/hive/warehouse` to view this directory tree.

However, when sharing data between many tools, it is often convenient to use *external* tables, where the data is managed outside of Hive's control, but Hive owns the table metadata.

Here is an *external* table that points to the *Word Count* results for our previous William Shakespeare exercise. Note that we have to specify the field separator (tab) and the directory location in HDFS. Hive will use all the files in that directory.

```
hive> CREATE EXTERNAL TABLE shakespeare_wc (
      name STRING,
      count INT
      )
  ROW FORMAT
  DELIMITED FIELDS TERMINATED BY '\t'
  LOCATION '/data/shakespeare/golden/simple-word-count/plain-text';
```

The following query runs a MapReduce job and takes ~40 seconds on my VM (sometimes longer...) and returns 64332:

```
hive> SELECT COUNT(*) FROM shakespeare_wc;
```

Here is the same *external* table, but this time, we are using `SequenceFile` storage. Note that location change.

```
hive> SET io.seqfile.compression.type=BLOCK;
hive> CREATE EXTERNAL TABLE shakespeare_wc2 (
      name STRING,
      count INT
      )
  ROW FORMAT
  DELIMITED FIELDS TERMINATED BY '\t'
  STORED AS SEQUENCEFILE
  LOCATION '/data/shakespeare/golden/simple-word-count/sequence-file';
```

If you add the `EXTENDED` keyword to `DESCRIBE`, you get more information:

```
hive> DESCRIBE EXTENDED shakespeare_wc2;
```

How does the change affect the timing of the same query?

```
SELECT COUNT(*) FROM shakespeare_wc2;
```

Let's create another *external* table that demonstrates the support for complex data types (discussed previously) in columns with non-trivial columns and the all the delimiters explicit set. In fact, we are just specifying Hive's default delimiters, where `\001` is Control-A, `\002` is Control-B, and `\003` is Control-C:

```
hive> CREATE EXTERNAL TABLE employees (
      name         STRING,
      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
  LOCATION '/user/thinkbig/employees';
```

(The indentation and breaks between lines are arbitrary; we chose them for clarity.) We have defined keys for indexing here, so we're just using the name as a key. Hence `subordinates` is an array of other `employee` names. An example of a `deduction` is `'Federal Tax' -> .20`, meaning 20% of the employee's salary.

Here is a sample record: Mary SmithBill KingFederal Taxes.2State Taxes.05Insurance.1100 Ontario St.ChicagoIL60601

Can you recognizes the different fields and complex datatype elements?

```
hive> DESCRIBE employees;
OK
name    string
subordinates   array<string>
deductions   map<string,float>
address struct<street:string,city:string,state:string,zip:int>
Time taken: 0.733 seconds

hive> SELECT * FROM employees;
OK
Mary Smith  ["Bill King"]   {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}    {"street":"100 Ontario St.","city":"Chicago","state":"IL","zip":60601}
...

hive> -- Float comparisons have a bug: x > 0.1 will return 0.1 values! Use 0.11.
hive> SELECT name FROM employees WHERE deductions['Federal Taxes'] > 0.11;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
...
OK
Mary Smith
...
hive> SELECT name FROM employees WHERE subordinates[1] = 'Todd Jones';
OK
Mary Smith
...
hive> SELECT name FROM employees WHERE size(subordinates) > 0;
OK
Mary Smith
...
hive> SELECT name FROM employees WHERE address.city = 'Chicago';
OK
Mary Smith
...
```

Here is an example of creating a table with partitions.

```
hive> CREATE TABLE stocks (
    ymd             STRING,
    price_open      FLOAT,
    price_high      FLOAT,
    price_low       FLOAT,
    price_close     FLOAT,
    volume          FLOAT,
    price_adj_close FLOAT
  )
  PARTITIONED BY (exchange STRING, symbol STRING)
  STORED AS SEQUENCEFILE;
```

Where `ymd` is "year-month-day". Using the name `date` causes a parse error!

Once you have data loaded into a partitioned table (see below), you can view the partitions with this command.

```
hive> SHOW PARTITIONS stocks;
```

Here is a different way to organize this data, with separate tables for each exchange and clustering of the data by date (`ymd`), sorted it by `symbol` , and bucketized into 32 buckets. (Read the Hive documentation about clustering and bucketing, as you must be careful how you load data into such tables; Hive doesn't handle the clustering and bucketing for you! These settings only affect queries.)

```
hive> CREATE TABLE nasdaq (
    symbol          STRING,
    ymd             STRING,
    price_open      FLOAT,
    price_high      FLOAT,
    price_low       FLOAT,
    price_close     FLOAT,
    volume          FLOAT,
    price_adj_close FLOAT
  )
  CLUSTERED BY (ymd) SORTED BY (symbol) INTO 32 BUCKETS
  STORED AS SEQUENCEFILE;
```

Altering Tables

You can rename a table, add, modify, or remove columns, add or remove partitions, change the file storage and location, etc. Note that changes to the table schema, e.g., changing columns or adding/removing partitions, has no affect on the underlying data. For example, a specified partition might not even exist! Only when queries are done will the schema changes have an effect. If a partition is missing, it is ignored. If a column type is wrong, it may cause a query error.

Here are some examples, assuming the following `apple` table exists and was created from our `stocks` table.

```
hive> CREATE TABLE apple AS SELECT ymd, price_close
    FROM stocks WHERE symbol = 'AAPL' AND exchange = 'NASDAQ';
```

Change the name of the table:

```
hive> ALTER TABLE apple RENAME TO aapl;
```

Change the `price_close` column to `close`, keep the same type, and add a comment:

```
hive> ALTER TABLE aapl CHANGE COLUMN price_close close STRING COMMENT 'The closing price';
```

Add a new `adj_close` column:

```
hive> ALTER TABLE aapl ADD COLUMNS (adj_close STRING COMMENT 'The adjusted closing price');
```

Note that all the values in this column will be `NULL`, *unless* the underlying storage already has a third column of data (e.g., in the case of an external file *or* a previous `ALTER` statement removed the column from the schema). There is no way to specify a default value for the column.

Loading Data

There are three ways to load data into tables.

1. Define the table to be `EXTERNAL` so you can modify the distributed file system directory to which it points. Recall from the previous section that if the table is partitioned, you must use `ALTER TABLE` to add new partitions, where each partition which be stored in its own directory in the distributed file system.
2. Insert data from a query of one or more other table(s). Here is an example for our `stocks` table:

   hive> INSERT OVERWRITE TABLE stocks PARTITION (exchange='NASDAQ', symbol='AAPL') SELECT ymd, price*open, price*high, price*low, price*close, volume, price*adj*close FROM stocks2 WHERE symbol = 'AAPL' AND exchange = 'NASDAQ';

3. Combine `INSERT` with `CREATE TABLE`:

   hive> CREATE TABLE apple AS SELECT ymd, price_close FROM stocks WHERE symbol = 'AAPL' AND exchange = 'NASDAQ';

Select Statements

Many of the standard features of SQL statements are supported.

Operators and Functions

Here are the supported operators and functions, adapted from the Hive Tutorial.

Relational Operators

All return `TRUE` or `FALSE`.

Operator	Types of Operands	Notes
`A = B`	primitives	`TRUE` if they are equal.
`A != B`	primitives	`TRUE` if they are **not** equal.
`A <> B`	primitives	`TRUE` if they are **not** equal.
`A < B`	primitives	Less than. Buggy for `FLOATS`, `DOUBLES`; behaves like `<=`.
`A <= B`	primitives	Less than or equals to.
`A > B`	primitives	Greater than. Buggy for `FLOATS`, `DOUBLES`; behaves like `>=`.
`A >= B`	primitives	Less than or equals to.
`A IS NULL`	all types	`TRUE` if `A` is `NULL`.
`A IS NOT NULL`	all types	`TRUE` if `A` is **not** `NULL`.
`A LIKE B`	strings	Character by character comparison. `TRUE` if string `A` matches the SQL simple regular expression `B`. The `_` character in `B` matches any character in `A` and the `%` character in `B` matches an arbitrary number of characters in `A`. To match a literal `%` use `\%`.
`A RLIKE B`	strings	`TRUE` if string `A` matches the Java regular expression `B`.
`A REGEXP B`	strings	Same as `RLIKE`.

Arithmetic Operators

All take only number types and all return number types. When mixing number types, e.g., `INT` and `FLOAT`, the usual promotion rules apply.

Operator	Notes
`A + B`	Addition.
`A - B`	Subtraction.
`A * B`	Multiplication. If it would cause overflow, you must cast one of the operands to a "wider" type.
`A / B`	Division. If both operands are integer types, then the result is the quotient of the division.
`A % B`	Modulus; the remainder from division.
`A & B`	Bitwise `AND` of `A` and `B`.
`A | B`	Bitwise `OR` of `A` and `B`.
`A ^ B`	Bitwise `XOR` of `A` and `B`.
`~A`	Bitwise `NOT` of `A`.

Logical Operators

All return boolean `TRUE` or `FALSE`.

Operator	Notes
`A AND B`	`TRUE` if `A` is `TRUE` **and** `B` is `TRUE`.
`A && B`	Same as `A AND B`.
`A OR B`	`TRUE` if `A` is `TRUE` **or** `B` is `TRUE`.
`A | B`	Same as `A OR B`.
`NOT A`	`TRUE` if `A` is `FALSE`, otherwise `FALSE`.
`!A`	Same as `NOT A`.

Operators on Complex Types

The literal syntax for accessing elements in complex types.

Operator	Types of Operands	Notes
`A[n]`	`A` is an `ARRAY` and `n` is an `INT`	Returns the `n`th element in the array `A`, indexed from 0.
`M[key]`	`M` is a `MAP<K, V>` with `key` of type `K`	Returns the value corresponding to the key in the map or `NULL`.
`S.x`	`S` is a `STRUCT`	Returns the `x` field of `S`.

Built-in Functions

The list of available functions is embedded in the source file FunctionRegistry.java. Many of them simply call the corresponding Java JDK functions. The following list is from Hive v0.7.1. Subsequent versions may add additional functions.

Signature	Description
`BIGINT round(DOUBLE d)`	
`BIGINT floor(DOUBLE d)`	Returns the *maximum* `BIGINT` value that is equal to or less than `d`.
`BIGINT ceil(DOUBLE d)`	Returns the *minimum* `BIGINT` value that is equal to or greater than `d`.
`DOUBLE rand()`, `DOUBLE rand(int seed)`	Returns a random number (differen from row to row). Specifying the seed makes the generated random number sequence deterministic.
`STRING concat(STRING s1, STRING s2,...)`	Concatenates all string arguments.
`STRING substr(STRING s, int start)`	Returns the substring of `s` starting from the `start` position till the end of the string.
`STRING substr(STRING s, int start, int length)`	Returns the substring of `s` starting from the `start` position with the given `length`.
`STRING upper(STRING s)`	Returns the string that results from converting all characters of `s` to upper case.
`STRING ucase(STRING s)`	Same as `upper`.
`STRING lower(STRING s)`	Returns the string that results from converting all characters of `s` to lower case.
`STRING lcase(STRING s)`	Same as `lower`.
`STRING trim(STRING s)`	Returns the string that results from trimming whitespace from both ends of `s`.
`STRING ltrim(STRING s)`	Returns the string that results from trimming whitespace from the beginning (left hand side) of `s`.
`STRING rtrim(STRING s)`	Returns the string that results from trimming whitespace from the end (right hand side) of `s`.
`STRING regexp_replace(STRING s, STRING regex, STRING replacement)`	Returns the string that results from replacing all substrings in `s` that match the Java regular expression `regex` with the `replacement` string, which can be left blank, implying "". For example, `regexp_replace('foobar', 'oo|ar', )` returns `"fb"`.
`int size(Map<K.V>)`	Returns the number of elements in the `MAP`.
`int size(Array<T>)`	Returns the number of elements in the `ARRAY`.
`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 `BIGINT` value of 1. A `NULL` is returned if the conversion fails.
`STRING from_unixtime(int unixtime)`	Converts 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)`	Returns the date part of a timestamp string, e.g., `to_date("1970-01-01 00:00:00") = "1970-01-01"`.
`INT year(STRING date)`	Returns the year part of a date or a timestamp string, e.g., `year("1970-01-01 00:00:00") = 1970`, `year("1970-01-01") = 1970`.
`INT month(STRING date)`	Returns the month part of a date or a timestamp string, e.g., `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, e.g., `day("1970-11-01 00:00:00") = 1`, `day("1970-11-01") = 1`.
`STRING get_json_object(STRING json_string, STRING path)`	Extracts a JSON object from a JSON string `json_string` based on the JSON `path` specified, then returns the corresponding string representation of the extracted JSON object. Returns null if the input JSON string is invalid.

Aggregate Functions

"Aggregate" functions take a collection of things and return a computation over them.

Signature	Description	
`BIGINT count(*)`	Returns the total number of retrieved rows, including rows containing `NULL` values.
`BIGINT count(expr)`	Returns the number of rows for which the supplied expression is non-`NULL`.
`BIGINT count(DISTINCT expr[, expr_.])`	Returns the number of rows for which the supplied expressions are unique and non-`NULL`.
`DOUBLE sum(col)`	Returns the sum of the values in the column, including duplicates.
`DOUBLE sum(DISTINCT col)`	Returns the sum of the distinct values in the column.
`DOUBLE avg(col)`	Returns the average of the values in the column, including duplicates.
`DOUBLE avg(DISTINCT col)`	Returns the average of the distinct values in the column.
`DOUBLE min(col)`	Returns the minimum value in the column.
`DOUBLE max(col)`	Returns the maximum value in the column.

Example Select Statements

Using the `stocks` table we created previously, first notice that the first of these two queries does *not* invoke MapReduce, so it returns much faster than the second query.

```
hive> SELECT * FROM stocks s WHERE s.exchange = 'NASDAQ' s.symbol = 'AAPL' LIMIT 20;
hive> SELECT s.ymd, s.price_open FROM stocks s WHERE s.exchange = 'NASDAQ' AND s.symbol = 'AAPL' LIMIT 20;
```

Another key point about both queries is that they filter on both partitions defined for this table, the `exchange` and the `symbol`. That means that Hive skips all the directories that don't match the `NASDAQ` and `AAPL` partitions, speeding up the queries, even though the query includes a filtering `WHERE` clause. In fact, *no* filtering is actually done since all data in the selected partition combination matches `NASDAQ` and `AAPL`. From a logical standpoint, you could drop the `s.exchange = 'NASDAQ'` clause, which is a superset filter compared to the `s.symbol = 'AAPL'` clause, but leaving it in prevents Hive from searching all the NYSE partitions!

```
hive> EXPLAIN SELECT * FROM stocks s WHERE s.exchange = 'NASDAQ' AND s.symbol = 'AAPL' LIMIT 20;
```

Compute the average closing price for `AAPL` over the whole dataset ($51.75)

```
hive> SELECT avg(s.price_close) FROM stocks s WHERE s.symbol = 'AAPL' AND s.exchange = 'NASDAQ';
```

It's perhaps more interesting to compute the average over blocks of time, such as yearly, which uses the `GROUP BY` clause.

```
hive> SELECT year(s.ymd), avg(s.price_close) FROM stocks s WHERE s.symbol = 'AAPL' AND s.exchange = 'NASDAQ' GROUP BY year(s.ymd);
```

If you only care about the years where the average was within a certain range of values, then add a `HAVING` clause.

```
hive> SELECT year(s.ymd), avg(s.price_close) FROM stocks s WHERE s.symbol = 'AAPL' AND s.exchange = 'NASDAQ' GROUP BY year(s.ymd)
      HAVING avg(s.price_close) > 50.0 AND avg(s.price_close) < 100.0;
```

You can also select elements in complex data types. Recall our `employees` table.

```
hive> CREATE EXTERNAL TABLE employees (
  name         STRING,
  salary       FLOAT,
  subordinates ARRAY<STRING>,
  deductions   MAP<STRING, FLOAT>,
  address      STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION '/user/thinkbig/employees';
```

Here are some sample queries that probe elements of the `ARRAY`, `MAP`, and `STRUCT` columns.

```
hive> SELECT e.name, e.subordinates[0] FROM employees e;
hive> SELECT e.name, e.subordinates FROM employees e WHERE size(e.subordinates) > 0;

hive> SELECT e.name, e.deductions['Federal Taxes'] FROM employees e;
hive> SELECT e.name, e.deductions FROM employees e WHERE size(e.deductions) != 3;

hive> SELECT e.name, e.address from employees e where e.address.city != 'Chicago';
```

Hive supports the SQL `LIKE` statement. It also supports matching on Java-style regular expressions:

```
hive> SELECT e.name, e.address from employees e where e.address.city LIKE 'C%';
hive> SELECT e.name, e.address from employees e where e.address.zip RLIKE '60[56][0-9]{2}';
```

Be careful about known bugs in comparisons of `FLOAT` and `DOUBLE` values:

```
hive> SELECT name, deductions['Federal Taxes'] FROM employees WHERE deductions['Federal Taxes'] > 0.2;
```

Example Joins

You can do equi-joins on multiple tables. Note that hive only support *equality* join conditions, due to difficulties in translating other conditions into MapReduce jobs.

For the purposes of doing joins, consider this `dividends` table for quarterly dividend payments.

```
CREATE TABLE dividends (
  exchange        STRING,
  symbol          STRING,
  ymd             STRING,
  dividend        FLOAT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

Here is an example `EQUI-JOIN` on the closing prices for `AAPL` and the dividend values on the days that dividends were paid.

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
FROM stocks s
JOIN dividends d ON (s.ymd = d.ymd AND s.symbol = d.symbol)
WHERE s.symbol = 'AAPL';
OK
1987-05-11  AAPL    77.0    0.015
1987-08-10  AAPL    48.25   0.015
...
```

Here a similar `LEFT OUTER JOIN` (limited to a time range around a known dividend payment):

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
  FROM stocks s
  LEFT OUTER JOIN dividends d ON (s.ymd = d.ymd AND s.symbol = d.symbol)
  WHERE s.symbol = 'AAPL'     AND
        s.ymd >= '1995-11-15' AND
        s.ymd <= '1995-11-25';
OK
1995-11-15  AAPL    41.0    NULL
1995-11-16  AAPL    39.94   NULL
1995-11-17  AAPL    40.13   NULL
1995-11-20  AAPL    38.63   NULL
...
```

Notice that this `RIGHT OUTER JOIN` is structured to give the same output.

```
hive> SELECT s.ymd, s.symbol, s.price_close, d.dividend
  FROM dividends d
  RIGHT OUTER JOIN stocks s ON (s.ymd = d.ymd AND s.symbol = d.symbol)
  WHERE s.symbol = 'AAPL'     AND
        s.ymd >= '1995-11-15' AND
        s.ymd <= '1995-11-25';
OK
1995-11-15  AAPL    41.0    NULL
1995-11-16  AAPL    39.94   NULL
1995-11-17  AAPL    40.13   NULL
1995-11-20  AAPL    38.63   NULL
...
```

To do a `FULL OUTER JOIN`, simply replace `RIGHT` in the previous query with `FULL`.

A `LEFT SEMI-JOIN` is a workaround for the missing SQL "IN ... EXISTS" clause. A Hive limitation is that you can't reference the table used in the semi-join clause in the `SELECT` or `WHERE` clauses. The following query selects all closing prices for days when a dividend was payed.

```
hive> SELECT s.ymd, s.symbol, s.price_close
  FROM stocks s
  LEFT SEMI JOIN dividends d
  ON (s.ymd = d.ymd AND s.symbol = d.symbol)
  WHERE s.symbol = 'AAPL' AND s.exchange = 'NASDAQ';
```

A Map-side `LEFT JOIN` is an optimization that eliminates a reduce step, if one of several conditions are true. Either all but one table must be small enough to fit into an in-memory Hash or the data is already sorted and *bucketized*.

```
hive> SELECT /*+ MAPJOIN(d) */ s.ymd, s.symbol, s.price_close, d.dividend
  FROM stocks s
  JOIN dividends d ON (s.ymd = d.ymd AND s.symbol = d.symbol)
  WHERE s.symbol = 'AAPL';
```

Note that Hive v0.8.0 automatically performs this optimization, if you set the `hive.auto.convert.join` to `true`.

Ordering of Output Data

The output can be sorted. Hive supports the SQL `ORDER BY` clause, which does a *total ordering*. However, this has the major drawback of forcing all map output to go a single reducer, which might run for an unacceptably long time. Hence, when `hive.mapred.mode` is set to `strict`, Hive only allows the `ORDER BY` clause to be used if a `LIMIT` clause is also used. This restriction can be overridden by setting `hive.mapred.mode` to `nonstrict`.

```
hive> SELECT s.ymd, s.symbol, s.price_close
  FROM stocks s
  ORDER BY s.ymd ASC, s.symbol DESC
  LIMIT 50;
```

Note that you can specify `ASC` (ascending) order, the default, or `DESC` (descending) order.

To avoid funneling large data sets through a single reducer, Hive also provides a `SORT BY` clause that orders the output *locally* in *each* reducer, not *globally*. However, a clever partitioning of keys can result in reducer output files that provide a global ordering of the data if the files are concatenated appropriately.

```
hive> SELECT s.ymd, s.symbol, s.price_close
  FROM stocks s
  SORT BY s.ymd ASC, s.symbol DESC;
```

There are two other ordering options; `DISTRIBUTE BY` specifies columns for which output records with identical values are sent to the same `reducers. However, the column values won't be sorted that arrive at a particular reducer (like map output keys would be sorted). So, an addi` BY `clause is required if you want the column values sorted.` CLUSTER BY `is a short hand for` DISTRIBUTE BY `and` SORT BY` for the columns specified.

```
hive> SELECT s.ymd, s.symbol, s.price_close
  FROM stocks s
  DISTRIBUTE BY s.symbol
  SORT BY s.symbol ASC, s.ymd DESC;

hive> SELECT s.ymd, s.symbol, s.price_close
  FROM stocks s
  CLUSTER BY s.symbol;
```

The Select Transform or Map-Reduce Syntax

The `SELECT TRANSFORM` or `MAP` and `REDUCE` clauses let you call out to external programs to do map and or reduce tasks. This is a useful way to integrate 3rd-party tools into Hive queries. It's also a useful way to implement a calculation that's difficult to do in SQL, but much easier in a *Turing-complete* language. Essentially, we are exploiting Hadoop's *Streaming* API.

As an example, we compute the *Word Count* of Shakespeare's plays using the following Python *map* and *reduce* scripts, called `mapper.py` and `reducer.py`, respectively. We will assume these scripts are in `/usr/local/bin`. Here is `/usr/local/bin/mapper.py`.

```
#!/usr/bin/env python
# The mapper for the WordCount algorithm using Streaming.

import sys

for line in sys.stdin:
    words = line.strip().split()
    for word in words:
        print "%s\t%d" % (word.lower(), 1)
```

Here is `/usr/local/bin/reducer.py`.

```
#!/usr/bin/env python
# The reducer for the WordCount algorithm using Streaming.

import sys

# In the Java API the reducer receives:
#  key1 [value11 value12 ...]
#  key2 [value21 value22 ...]
#  ...
# In the streaming API, the reducer receives:
#  key1 value11
#  key1 value12
#  ...
#  key2 value21
#  key2 value22
#  ...

(last_seen_key, count_for_key) = (None, 0)

for line in sys.stdin:
    (key, value) = line.strip().split("\t")
    if last_seen_key and last_seen_key != key:
        print "%s\t%d" % (last_seen_key, count_for_key)
        (last_seen_key, count_for_key) = (key, int(value))
    else:
        (last_seen_key, count_for_key) = (key, count_for_key + int(value))

if last_seen_key:
    print "%s\t%d" % (last_seen_key, count_for_key)
```

Now, let's suppose we have one or more text files of Shakespeare's plays in HDFS in the directory `/data/shakespeare/input`. We first create an `EXTERNAL` table that points to this file. It treats each line as a record and each record as having a single field, the line's text.

```
hive> CREATE EXTERNAL TABLE shakespeare_line (line STRING)
  LOCATION '/data/shakespeare/input';
```

Next, crate the output table for the Word Count results.

```
hive> CREATE TABLE shakespeare_line_wc (word STRING, count INT)
  ROW FORMAT
    DELIMITED FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;
```

Finally, here is the query that puts them all together.

```
FROM (
  FROM shakespeare_line
    MAP line
    USING '/usr/local/bin/mapper.py'
    AS word, count
    CLUSTER BY word) wc
  INSERT OVERWRITE TABLE shakespeare_line_wc
    REDUCE wc.word, wc.count
    USING '/usr/local/bin/reducer.py'
    AS word, count;
```

Note that `CLUSTER BY` is also used. The `MAP` and `REDUCE` keywords are a bit misleading, because they may not actually map to underlying map and reduce steps, respectively. The alternative `SELECT TRANSFORM` syntax makes the technique more abstract and uniform.

```
FROM (
  FROM shakespeare_line
    SELECT TRANSFORM (line)
    USING '/usr/local/bin/mapper.py'
    AS word, count
    CLUSTER BY word) wc
  INSERT OVERWRITE TABLE shakespeare_line_wc
    SELECT TRANSFORM (wc.word, wc.count)
    USING '/usr/local/bin/reducer.py'
    AS word, count;
```

User Defined Functions (UDFs), User Defined Aggregate Functions (UDAFs), User Defined Table-Generating Functions (UDTFs)

User Defined Functions (UDFs) take a row (which could be a single value) and return a new row (or value). UDFs are one-to-one mappings.

The following `dividends` query uses the `year()` and `month()` UDFs to extract the year and month from a trading day date stamp, respectively, and the `lower()` UDF to convert the symbol to lower case.

```
hive> SELECT ymd, year(ymd), month(ymd), lower(symbol) FROM dividends LIMIT 20;
OK
2006-01-25  2006    1   amtd
2009-11-09  2009    11  ahgp
2009-08-10  2009    8   ahgp
...
```

User Defined Aggregate Functions (UDAFs) take multiple rows and return an *aggregate* of them as a new row. UDAFs are many-to-one mappings.

The following `dividends` query uses the `avg()` and `count()` UDAFs to average and sum the dividends payed by Apple, respectively.

```
hive> SELECT avg(dividend), count(dividend) FROM dividends WHERE symbol = 'AAPL';
OK
0.027142856695822306    35
```

User Defined Table-Generating Functions (UDTFs) are the least well known. They take a single row and return multiple new rows, effectively generating a new table. UDTFs are one-to-many mappings.

The following `employees` query uses the `explode()` UDTF to expand the `subordinates` `ARRAY` in the `employees` table rows. Note that `explode()` requires the data to be in an `ARRAY`. Also, an `AS new_col` is required, even if it is not subsequently used.

```
hive> SELECT explode(subordinates) AS subs FROM employees;
OK
Mary Smith
Todd Jones
Bill King
John Doe
Fred Finance
Stacy Accountant
```

If you have a custom UDF implemented in Java, you can use the `ADD JAR` command to make it visible across the Hadoop cluster, so every task process that Hive generates will be able to use it. This is the main difference between this command and adding a jar file to `HADOOP_CLASSPATH`. The latter does not propogate the jar file to all nodes.

Once the jar is added, then you create a "temporary" function that calls to your UDF. The following example, assumes you have implemented a custom `ROT13` cipher in a class `com.example.cipher.Rot13` and built a jar file `rot13-007.jar` that contains it. If the jar file is not in the current directory, then add the appropriate path to it in the `ADD JAR` command.

```
ADD JAR rot13-007.jar;

CREATE TEMPORARY FUNCTION rot13 AS 'com.example.cipher.Rot13';

SELECT name, spy_agency, rot13(message) FROM cloaks_n_daggers;
```

A *SerDe* (Serializer/Deserializer) is used to parse a record in a byte stream that uses a custom format. An `INPUTFORMAT` and `OUTPUTFORMAT` class defines how such records are stored in files. Specifically, the `INPUTFORMAT` defines how the files backing the tables are formatted, while the `OUTPUTFORMAT` defines the format that Hive will use when returning query results. The former must be a subclass of `org.apache.hadoop.mapreduce.lib.input.InputFormat`, while the latter is restricted to subclassing `org.apache.hadoop.hive.ql.io.HiveOutputFormat`.

The following table exposes several fields in Twitter messages encoded in JSON.

```
CREATE EXTERNAL TABLE twitter2 (
    tweet_id         BIGINT,
    created_at       STRING,
    text             STRING,
    user_id          BIGINT,
    user_screen_name STRING,
    user_lang        STRING
  )
  ROW FORMAT SERDE "org.apache.hadoop.hive.contrib.serde2.JsonSerde"
    WITH SERDEPROPERTIES (
      "tweet_id"="$.id",
      "created_at"="$.created_at",
      "text"="$.text",
      "user_id"="$.user.id",
      "user_screen_name"="$.user.screen_name",
      "user_lang"="$.user.lang"
    )
  STORED AS
    INPUTFORMAT  'org.apache.hadoop.mapreduce.lib.input.TextInputFormat'
    OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.HiveBinaryOutputFormat'
  LOCATION '/data/twitter/input';
```

Note that a *SerDe* is used that was contributed to the Hive project. Actually, it's an enhanced version of this *SerDe* found here. The enhancements include the `SERDEPROPERTIES` that allow the user to define the mapping between internal JSON fields and the table columns, among other things.

Using a *SerDe* doesn't require the `STORED AS INPUTFORMAT ... OUTPUTFORMAT ...` clause. Each can be used without the other, as needed.

Differences Between HiveQL, Oracle SQL, and ANSI SQL

HiveQL is another SQL dialect, but it is farther away from the ANSI standard SQL than most other SQL implementations. This section discusses some differences that will be apparent to something already very familiar with SQL, especially Oracle's dialect. Some differences are relatively minor and may be resolved over time. Others reflect more fundamental differences, like the limitations of HDFS.

Here we discuss a few of the differences.

No Transactions or Updates

These are planned and may appear soon, especially through the new HBase integration. Also, appending to HDFS files will finally be available in the Next Generation Hadoop (v.23).

NOT IN Query

In Oracle, you would write something like this:

```
SELECT * FROM employees
  WHERE NOT IN
    (SELECT dept_number FROM departments);
```

Here's how you would have to write it in HiveQL:

```
SELECT * FROM employees
  LEFT OUTER JOIN departments
  ON (employees.dept_number = departments.dept_number)
  WHERE departments.dept_number IS NULL;
```

JOIN Syntax

You may have already noticed that the equi-join syntax is different.

In Oracle.

```
SELECT * FROM employees, departments
  WHERE employees.dept_number = departments.dept_number;
```

Here's how you would have to write it in HiveQL:

```
SELECT * FROM employees
  JOIN departments
  ON (employees.dept_number = departments.dept_number);
```

More Flexible File Storage, etc.

On the other hand, Hive has a lot more flexbility about how you store the data, including the file and record formats, locations, etc. You can also redirect query output directory to a local or HDFS directory, as in this example.

```
INSERT OVERWRITE LOCAL DIRECTORY '/home/me/bosses'
  SELECT * FROM employees WHERE length(employees.subordinates) > 0;
```