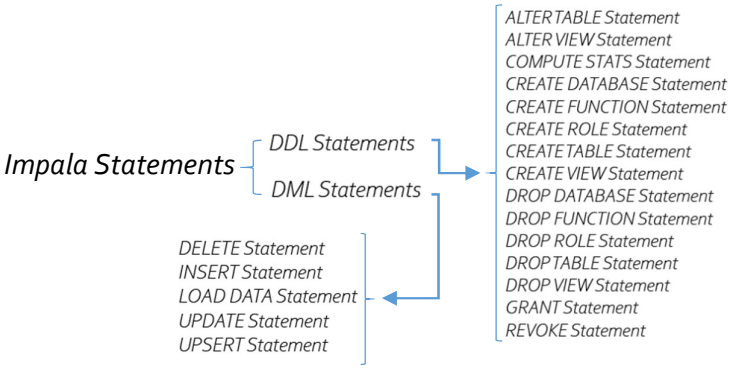
# *A*pache *I*mpala
# *C*heat *S*heet

*By: Aref Gholizadeh, December 2019*

*A Quick guide to Impala **D**DL Statements, **D**ML Statements, **U**seful Clauses and **O**perators*




## ■ Impala DDL Statements

**Description**:

*DDL refers to "**D**ata **D**efinition **L**anguage", a subset of SQL statements that change the structure of the database schema in some way, typically by creating, deleting, or modifying schema objects such as databases, tables, and views.*

### Using **CREATE VIEW**

**Description**:

*The CREATE VIEW statement lets you create a shorthand abbreviation for a more complicated query.*

**Sample**:

```
CREATE VIEW v6 AS SELECT t1.c1, t2.c2 FROM t1
JOIN t2 ON t1.id = t2.id;
```

### Using **CREATE FUNCTION**

**Description**:

*Creates a user-defined function (UDF), which you can use to implement custom logic during SELECT or INSERT operations.*
*To create a persistent scalar C++ UDF with CREATE FUNCTION:*

```
CREATE FUNCTION [IF NOT EXISTS]
[db_name.]function_name([arg_type[,
arg_type...])
  RETURNS return_type
  LOCATION 'hdfs_path_to_dot_so'
  SYMBOL='symbol_name'
```

*To create a persistent Java UDF with CREATE FUNCTION:*

```
CREATE FUNCTION [IF NOT EXISTS]
[db_name.]function_name
  LOCATION 'hdfs_path_to_jar'
  SYMBOL='class_name'
```

**Sample**:

```
insert into bigint_x values (1), (2);
```

```
select testudf(x) from bigint_x;
+-----------------+
| udfs.testudf(x) |
+-----------------+
| 1               |
| 2               |
+-----------------+
```

## ■ Impala DML Statements

**Description**:

*DML refers to "Data Manipulation Language", a subset of SQL statements that modify the data stored in tables. Because Impala focuses on query performance and leverages the append-only nature of HDFS storage, currently Impala only supports a small set of DML statements:*

### **INSERT** Query

```
INSERT OVERWRITE TABLE tab3

SELECT id, col_1, col_2, MONTH (col_3),
DAYOFMONTH (col_3)

FROM tab1 WHERE YEAR (col_3) = 2012;
```

### Aggregation and Join

```
SELECT tab2.*

          FROM tab2,

          (SELECT tab1.col_1, MAX (tab2.col_2)
AS max_col2

          FROM tab2, tab1

          WHERE tab1.id = tab2.id

          GROUP BY col_1) subquery1

          WHERE subquery1.max_col2 =
tab2.col_2;
```

### Subquery, Aggregate and Joins

```
SELECT tab2.*

          FROM tab2,

          (SELECT tab1.col_1, MAX (tab2.col_2)
AS max_col2

          FROM tab2, tab1

          WHERE tab1.id = tab2.id

          GROUP BY col_1) subquery1

          WHERE subquery1.max_col2 =
tab2.col_2;
```

### . Version ()

```
Select version ();

Result: impalad version 2.2.0-cdh5.4.8 RELEASE
```

Description:
*call the version () function to confirm which version of Impala you are running. Version number is important when consulting documentation and dealing with support issues.*

. Show databases

Code: show databases;
Result:
_impala_builtins
corr
cs
default
nethouse
ps
sep_dsl
ssad

. Show tables;

Code: show tables;
Code: show tables in … (mention database name);
Code: show tables like '*view*';
Code: show tables in … (mention database name) like 'customer*';

*A completely empty Impala instance contains no tables, but still has two databases:*
***default**, where new tables are created when you do not specify any other database.*
***_impala_builtins**, a system database used to hold all the built-in functions.*

**. current_database ();**

Code: select current_database ();
Result: default

**. Use**

Code: use … (database name);
Description: *Once you know what tables and databases are available, you descend into a database with the USE statement.*

**. Count ()**

Code: select count (*) from … (table name);
Code: select count (distinct … (columns)) from customer;
Result: 119
Description: *to know the number of rows in the table, to include null in the results put * in the parenthesis.*

**. ALTER**

Code: alter table t1 rename to experiments.t1;
Description: *The ALTER TABLE statement lets you move the table to the intended database.*

**.COMPUTE INCREMENTAL**

Code: compute incremental stats … (table name);
Code: show table stats … (table name);
Result: 119
Description:
*the COMPUTE INCREMENTAL STATS statement is the way to collect statistics for partitioned tables. Then the SHOW TABLE STATS statement confirms that the statistics are in place for each partition, and illustrates how many files and how much raw data is in each partition.*

. Desc or Describe;

Code: describe city;
Code: desc costumer;
Description:
*To understand the structure of each table, you use the DESCRIBE command.*

## ■ Useful Clauses

Using having clause:

Sample:

```
Select ss_item_sk as Item, count (ss_item_sk) as Times Purchased,
Sum (ss_quantity) as Total_Quantity_Purchased
From store_sales
Group by ss_item_sk
having times_purchased >= 100
Order by sum (ss_quantity) Limit 5;
```

Description:
*it is always used in conjunction with a function such as COUNT (), SUM (), AVG (), MIN (), or MAX (), and typically with the GROUP BY clause also. The HAVING clause lets you filter the results of aggregate functions, because you cannot refer to those expressions in the WHERE clause. For example, to find the 5 lowest-selling items that were included in at least 100 sales transactions, we could use this query.*

Using **offset** clause:
Sample:
Select x from numbers order by x limit 5 offset 5;
Description:
*The OFFSET clause in a SELECT query causes the result set to start some number of rows after the logical first item.*

Using **Group by** Clause:
Sample:

```
Select ss_item_sk as Item, count (ss_item_sk) as Times_Purchased,
Sum (ss_quantity) as Total_Quantity_Purchased
From store_sales
Group by ss_item_sk
Having times_purchased >= 100
Order by sum (ss_quantity)
Limit 5;
```

Description:
*Specify the GROUP BY clause in queries that use aggregation functions, such as COUNT (), SUM (), AVG (), MIN (), and MAX (). Specify in the GROUP BY clause the names of all the columns that do not participate in the aggregation operation.*

Using **WITH** Clause:
Sample:

```
With t1 as (select 1) (with t2 as (select 2) select * from t2)
union all select * from t1;
```

Description:
*A clause that can be added before a SELECT statement, to define aliases for complicated expressions that are referenced multiple times within the body of the SELECT. Similar to CREATE VIEW, except that the table and column names defined in the WITH*

*clause do not persist after the query finishes, and do not conflict with names used in actual tables or views. Also known as "subquery factoring".*

Using **Union** Clause:
**Sample**:
**Union sample:**
query_1 UNION [DISTINCT | ALL] query_2;
**Union All samples:**
**Select x from few_ints union all select x from few_ints;**
**Description**:
*The UNION clause lets you combine the result sets of multiple queries. By default, the result sets are combined as if the DISTINCT operator was applied.*
The **UNION** *keyword by itself is the same as UNION DISTINCT. Because eliminating duplicates can be a memory-intensive process for a large result set, prefer UNION ALL where practical. (That is, when you know* the different queries in the union will not produce any *duplicates, or where the duplicate values are acceptable.*

## ■ Operators

Arithmetic Operators
+ , -, *, /, DIV (integer division), % (Modulo operator), & (Bitwise AND), | (Bitwise OR),~ (Bitwise NOT), ^ (Bitwise XOR)

Using **BETWEEN** Operator:
expression BETWEEN lower_bound AND upper_bound

Using **Comparison** Operator:
=, !=, <>: apply to all scalar types
<, <=, >, >=: apply to all scalar types
for BOOLEAN, **TRUE is considered greater than FALSE**.

Using **EXIST** Operator:
*Typically use it to find values from one table that have corresponding values in another table.*

**Select** y from t2 where exists (select x from empty);
**Select** y from t2 where not exists (select x from empty);

Using **ILIKE** Operator:
**Description**:
String expression ILIKE wildcard expression:
**Sample**:
Select 'fooBar' ilike 'FOOBAR';
Alternatively:
Select 'fooBar' ilike 'f%'
**Result**: true
String expression NOT ILIKE wildcard_expression
**Sample**:
Select 'ABCXYZ' not ilike 'ab_xyz';
**Result**: false

Using **IN** Operator
**Sample**:
**Select** 1 in (1, null, 2, 3);
**Result**: true
**Select** 5 not in (1, null, 2, 3);
**Result**: true

Using **IREGEXP** Operator:
**Sample**:
**Select** 'abcABCaabbcc' iregexp '^[a-c]+$';
**Result**: true

Using **IS DISTINCT FROM** Operator:
**Sample**:
Select 1 is not distinct from 1;
Result: true

Using **REGEXP** Operator:
**Description**:
string_expression **REGEXP** regular_expression
*The RLIKE operator is a synonym for REGEXP*
Using **DISTINCT** Operator:
**Description**:
*The DISTINCT operator in a SELECT statement filters the result set to remove duplicates.*
**Sample**:
Select DISTINCT c_birth_country FROM customer;

## ■ Impala Built-In Functions:

Impala Mathematical Functions
Impala Type Conversion Functions
Impala Date and Time Functions
Impala Conditional Functions
Impala String Functions
Impala Aggregate Functions.
Impala Analytic Functions
Impala Bit Functions
Impala Miscellaneous Functions

Using **VARIANCE**:
**Description**:
*An aggregate function that returns the variance of a set of numbers. This mathematical property signifies how far the values spread apart from the mean.*
**Sample**:

```
Select variance(score) from test_scores;
```

Result:

```
812.25
```

Using **APPX_MEDIAN:**
**Description**:
*Returns a value that is approximately the median (midpoint) of values in the set of input values.*
**Sample**:

```
Select appx_median(x) from million_numbers;
```

**Result**:

```
24721.6
```

Using **AVG:**
**Description**:
*An aggregate function that returns the average value from a set of numbers or TIMESTAMP values.*

```
AVG([DISTINCT | ALL] expression) [OVER
(analytic_clause)]
```

**Sample**:

```
Select x, property,

  avg(x) over (partition by property order by
x) as 'cumulative average'

  from int_t where property in ('odd','even');
```

Using **GROUP_CONCAT:**

**Description**:

*An aggregate function that returns a single string representing the argument value concatenated together for each row of the result set. If the optional separator string is specified, the separator is added between each pair of concatenated values. The default separator is a comma followed by a space.*

**Sample**:

```
select group_concat(distinct s) from t1;
```

Using **MAX:**

**Sample**:

```
select x, property, max(x) over (partition by
property) as max from int_t where property in
('odd','even');
```

Using **STDDEV**:

**Description**:

An aggregate function that returns the standard deviation of a set of numbers.

**Sample**:

```
Select stddev(score) from test_scores;
```

**For more Information, please visit:**

*https://docs.cloudera.com/documentation/enterprise/latest/topics/impala.html*

## ■ *Complete List of built-In Functions:*

*ABS*
*ACOS*
*ADD_MONTHS*
*ADDDATE*
*APPX_MEDIAN*
*ASCII*
*ASIN*
*ATAN*
*ATAN2*
*AVG*
*AVG - Analytic Function*
*BASE64DECODE*
*BASE64ENCODE*
*BITAND*
*BIN*
*BITNOT*
*BITOR*
*BITXOR*
*BTRIM*
*CASE*
*CASE WHEN*
*CAST*
*CEIL, CEILING, DCEIL*
*CHAR_LENGTH*
*CHR*
*COALESCE*
*CONCAT*
*CONCAT_WS*
*CONV*
*COS*
*COSH*
*COT*
*COUNT*
*COUNT - Analytic Function*
*COUNTSET*
*CUME_DIST*
*CURRENT_DATABASE*
*CURRENT_TIMESTAMP*
*DATE_ADD*
*DATE_PART*
*DATE_SUB*
*DATE_TRUNC*
*DATEDIFF*
*DAY*
*DAYNAME*
*DAYOFWEEK*
*DAYOFYEAR*
*DAYS_ADD*
*DAYS_SUB*
*DECODE*
*DEGREES*
*DENSE_RANK*
*E*
*EFFECTIVE_USER*
*EXP*
*EXTRACT*
*FACTORIAL*
*FIND_IN_SET*
*FIRST_VALUE*
*FLOOR, DFLOOR*
*FMOD*
*FNV_HASH*
*GET_JSON_OBJECT*
*FROM_UNIXTIME*
*FROM_TIMESTAMP*
*FROM_UTC_TIMESTAMP*
*GETBIT*
*GREATEST*
*GROUP_CONCAT*
*GROUP_CONCAT - Analytic Function*
*HEX*
*HOUR*
*HOURS_ADD*
*HOURS_SUB*
*IF*
*IFNULL*
*INITCAP*
*INSTR*
*INT_MONTHS_BETWEEN*
*IS_INF*
*IS_NAN*
*ISFALSE*
*ISNOTFALSE*
*ISNOTTRUE*
*ISNULL*
*ISTRUE*
*JARO_DISTANCE, JARO_DIST*
*JARO_SIMILARITY, JARO_SIM*
*JARO_WINKER_DISTANCE, JW_DST*
*JARO_WINKER_SIMILARITY, JW_SIM*
*LAG*
*LAST_VALUE*
*LEAD*
*LEAST*
*LEFT*
*LENGTH*
*LN*
*LOCATE*
*LOG*
*LOG10*
*LOG2*
*LOWER, LCASE*
*LPAD*
*LTRIM*
*MAX*
*MAX - Analytic Function*
*MAX_INT, MAX_TINYINT, MAX_SMALLINT,*
*MAX_BIGINT*
*MICROSECONDS_ADD*
*MICROSECONDS_SUB*
*MILLISECOND*
*MILLISECONDS_ADD*
*MILLISECONDS_SUB*
*MIN*
*MIN - Analytic Function*
*MIN_INT, MIN_TINYINT, MIN_SMALLINT,*
*MIN_BIGINT*
*MINUTE*
*MINUTES_ADD*
*MINUTES_SUB*
*MOD*
*MONTH*
*MONTHNAME*
*MONTHS_ADD*
*MONTHS_BETWEEN*
*MONTHS_SUB*
*MURMUR_HASH*
*NANOSECONDS_ADD*
*NANOSECONDS_SUB*
*NDV*
*NEGATIVE*
*NEXT_DAY*
*NONNULLVALUE*
*NOW*
*NTILE*
*NULLIF*
*NULLIFZERO*
*NULLVALUE*
*NVL*
*NVL2*
*OVER Clause*
*PARSE_URL*
*PERCENT_RANK*
*PI*
*PID*
*PMOD*
*POSITIVE*
*POW, POWER, DPOW, FPOW*
*PRECISION*
*QUARTER*
*QUOTIENT*
*RADIANS*
*RAND, RANDOM*
*RANK*
*REGEXP_ESCAPE*
*REGEXP_EXTRACT*
*REGEXP_LIKE*
*REGEXP_REPLACE*
*REPEAT*
*REPLACE*
*REVERSE*
*RIGHT*
*ROTATELEFT*
*ROTATERIGHT*
*ROUND, DROUND*
*ROW_NUMBER*
*RPAD*
*RTRIM*
*SCALE*
*SECOND*
*SECONDS_ADD*
*SECONDS_SUB*
*SETBIT*
*SHIFTLEFT*
*SHIFTRIGHT*
*SIGN*
*SIN*
*SINH*
*SLEEP*
*SPACE*
*SPLIT_PART*
*SQRT*
*STDDEV, STDDEV_SAMP, STDDEV_POP*
*STRLEFT*
*STRRIGHT*
*SUBDATE*
*SUBSTR, SUBSTRING*
*SUM*
*SUM - Analytic Function*
*TAN*
*TANH*
*TIMEOFDAY*
*TIMESTAMP_CMP*
*TO_DATE*
*TO_TIMESTAMP*
*TO_UTC_TIMESTAMP*
*TRANSLATE*
*TRIM*
*TRUNC*
*TRUNCATE, DTRUNC, TRUNC*
*TYPEOF*
*UNHEX*
*UNIX_TIMESTAMP*
*UPPER, UCASE*
*USER*
*UTC_TIMESTAMP*
*UUID*
*VARIANCE, VARIANCE_SAMP, VARIANCE_POP,*
*VAR_SAMP, VAR_POP*
*VERSION*
*WEEKOFYEAR*
*WEEKS_ADD*
*WEEKS_SUB*
*WIDTH_BUCKET*
*YEAR*
*YEARS_ADD*
*YEARS_SUB*
*ZEROIFNULL*