Cloudera Runtime 7.0.3

Impala SQL Reference

Date published: 2019-11-22
Date modified:

https://docs.cloudera.com/

Legal Notice

Contents

Impala SQL

Use Impala SQL to manage and access data in Hadoop storage.

Impala provides a high degree of compatibility with the Hive Query Language. Impala uses the same metadata store as Hive to record information about table structure and properties. Therefore, Impala can access tables defined through the native Impala CREATE TABLE command, or tables created using the Hive data definition language (DDL).

Impala schema objects

Schema objects are logical structures that you use to store and manipulate data. Schema objects include databases, tables, views, functions, etc.

Impala aliases

You can assign an alias to the names of tables, columns, or column expressions in SQL statements and use the alias when referencing the table or column in the same statement. You typically specify aliases that are shorter, easier to remember, or both than the original names. The aliases are printed in the query header, making them useful for self-documenting output.

The following rules apply to aliases:

- To set up an alias, add the AS *alias* clause immediately after any table, column, or expression name in the SELE CT list or FROM list of a query. The AS keyword is optional.

  You can specify the alias immediately after the original name.
- You can specify column aliases with or without the AS keyword, and with no quotation marks, single quotation marks, or double quotation marks.
- Aliases are not case sensitive.
- Aliases can be up to the maximum length of a Java string, 2147483647.
- Aliases can include additional characters such as spaces and dashes when they are quoted using backtick characters (``).
- To use an alias name that matches one of the Impala reserved keywords, surround the identifier with either single or double quotation marks, or backtick characters.
- Aliases are allowed at the top level of the GROUP BY, HAVING, and ORDER BY clauses, e.g. GROUP BY *alias*.
- Aliases are not allowed in subexpressions of the GROUP BY, HAVING, and ORDER BY clauses, e,g. GROUP BY *alias/5*.

Related Information

Impala reserved words

Impala databases

Databases are logical containers for a group of tables, and each database defines a separate namespace.

Creating a database is a lightweight operation. There are minimal database-specific properties to configure, such as LOCATION and COMMENT.

You can change the owner of a database with the ALTER DATABASE statement.

Typically, you create a separate database for each project or application, to avoid naming conflicts between tables and to make clear which tables are related to each other.

The USE statement lets you switch between databases. Unqualified references to tables, views, and functions refer to objects within the current database. You can also refer to objects in other databases by using qualified names of the form *dbname.object_name*.

Each database is physically represented by a directory in HDFS. When you do not specify a LOCATION attribute, the directory is located in the Impala data directory with the associated tables managed by Impala. When you do specify a LOCATION attribute, any read and write operations for tables in that database are relative to the specified HDFS directory.

Impala includes two predefined databases:

- default

  By default, you use the default database when you connect to Impala. Tables created in the default database are physically located at one level higher in HDFS than all the user-created databases.
- _impala_builtins

  All Impala built-in functions reside in this database.

Related Information

CREATE DATABASE statement

Overview of Impala functions

Functions let you apply arithmetic, string, or other computations and transformations to Impala data. You typically use them in SELECT lists and WHERE clauses to filter and format query results so that the result set is exactly what you want, with no further processing needed on the application side.

Scalar functions return a single result for each input row.

```
[localhost:21000] > select name, population from country where continent = '
North America' order by population desc limit 4;
[localhost:21000] > select upper(name), population from country where conti
nent = 'North America' order by population desc limit 4;
+-------------+------------+
| upper(name) | population |
+-------------+------------+
| USA         | 320000000  |
| MEXICO      | 122000000  |
| CANADA      | 25000000   |
| GUATEMALA   | 16000000   |
+-------------+------------+
```

Aggregate functions combine the results from multiple rows: either a single result for the entire table, or a separate result for each group of rows. Aggregate functions are frequently used in combination with GROUP BY and HAVI NG clauses in the SELECT statement.

```
[localhost:21000] > select continent, sum(population) as howmany from cou
ntry group by continent order by howmany desc;
+---------------+------------+
| continent     | howmany    |
+---------------+------------+
| Asia          | 4298723000 |
| Africa        | 1110635000 |
| Europe        | 742452000  |
| North America | 565265000  |
| South America | 406740000  |
| Oceania       | 38304000   |
+---------------+------------+
```

User-defined functions (UDFs) let you code your own logic. They can be either scalar or aggregate functions. UDFs let you implement important business or scientific logic using high-performance code for Impala to automatically parallelize. You can also use UDFs to implement convenience functions to simplify reporting or porting SQL from other database systems.

```
[localhost:21000] > select rot13('Hello world!') as 'Weak obfuscation';
```

```
+------------------+
| weak obfuscation |
+------------------+
| Uryyb jbeyq!     |
+------------------+
[localhost:21000] > select likelihood_of_new_subatomic_particle(sensor1,
 sensor2, sensor3) as probability
                  > from experimental_results group by experiment;
```

Each function is associated with a specific database. For example, if you issue a USE somedb statement followed by CREATE FUNCTION somefunc, the new function is created in the somedb database, and you could refer to it through the fully qualified name somedb.somefunc. You could then issue another USE statement and create a function with the same name in a different database.

Impala built-in functions are associated with a special database named _impala_builtins, which lets you refer to them from any database without qualifying the name.

```
[localhost:21000] > show databases;
+-------------------------+
| name                    |
+-------------------------+
| _impala_builtins        |
| analytic_functions      |
| avro_testing            |
| data_file_size          |
...
[localhost:21000] > show functions in _impala_builtins like '*subs*';
+-------------+-----------------------------------+
| return type | signature                         |
+-------------+-----------------------------------+
| STRING      | substr(STRING, BIGINT)            |
| STRING      | substr(STRING, BIGINT, BIGINT)    |
| STRING      | substring(STRING, BIGINT)         |
| STRING      | substring(STRING, BIGINT, BIGINT) |
+-------------+-----------------------------------+
```

Related Information

Impala built-in functions

User-defined functions (UDFs)

Impala identifiers

Identifiers are the names of databases, tables, or columns that you specify when you create the objects.

The following rules apply to identifiers in Impala:

- The minimum length of an identifier is 1 character.
- The maximum length of an identifier is currently 128 characters, enforced by the Metastore database.
- An identifier must start with an alphanumeric or underscore character. The remainder can contain any combination of alphanumeric characters and underscores.

  Quoting the identifier with backticks has no effect on the allowed characters in the name.
- An identifier can contain only ASCII characters.

- To use an identifier name that matches one of the Impala reserved keywords, surround the identifier with `` characters (backticks). Quote the reserved word even if it is part of a fully qualified name.

  The following example shows how a reserved word can be used as a column name if it is quoted with backticks in the CREATE TABLE statement, and how the column name must also be quoted with backticks in a query:

  Important: Because the list of reserved words grows over time as new SQL syntax is added, consider adopting coding conventions (especially for any automated scripts or in packaged applications) to always quote all identifiers with backticks. Quoting all identifiers protects your SQL from compatibility issues if new reserved words are added in later releases.
- Impala identifiers are case-insensitive.

Related Information

Impala reserved words

Impala tables

Tables are the primary containers for data in Impala.

Logically, each table has a structure based on the definition of its columns, partitions, and other properties.

Physically, each table that uses HDFS storage is associated with a directory in HDFS. The table data consists of all the data files underneath that directory:

- Internal tables are managed by Impala, and use directories inside the designated Impala work area.
- External tables use arbitrary HDFS directories, where the data files are typically shared between different Hadoop components.
- Large-scale data is usually handled by partitioned tables, where the data files are divided among different HDFS subdirectories.

Impala queries ignore files with extensions commonly used for temporary work files by Hadoop tools. Any files with extensions .tmp or .copying are not considered part of the Impala table. The suffix matching is case-insensitive, so for example Impala ignores both .copying and .COPYING suffixes.

When you create a table in Impala, you can create an internal table or an external table.

To see whether a table is internal or external, and its associated HDFS location, issue the statement DESCRIBE FORMATTED *table_name*. The Table Type field displays MANAGED_TABLE for internal tables and EXTERNAL_TABLE for external tables. The Location field displays the path of the table directory as an HDFS URI.

Internal Tables

The default kind of table produced by the CREATE TABLE statement is known as an internal table.

- Impala creates a directory in HDFS to hold the data files.
- You can create data in internal tables by issuing INSERT or LOAD DATA statements.
- If you add or replace data using HDFS operations, issue the REFRESH command in `impala-shell` so that Impala recognizes the changes in data files, block locations, and so on.
- When you issue a DROP TABLE statement, Impala physically removes all the data files from the directory.
- To see whether a table is internal or external, and its associated HDFS location, issue the statement DESCRIBE FORMATTED *table_name*. The Table Type field displays MANAGED_TABLE for internal tables and EXTERNAL_TABLE for external tables. The Location field displays the path of the table directory as an HDFS URI.
- When you issue an ALTER TABLE statement to rename an internal table, all data files are moved into the new HDFS directory for the table. The files are moved even if they were formerly in a directory outside the Impala data directory, for example in an internal table with a LOCATION attribute pointing to an outside HDFS directory.

External Tables

The syntax CREATE EXTERNAL TABLE sets up an Impala table that points at existing data files, potentially in HDFS locations outside the normal Impala data directories.. This operation saves the expense of importing the data into a new table when you already have the data files in a known location in HDFS, in the desired file format.

- You can use Impala to query the data in this table.
- You can create data in external tables by issuing INSERT or LOAD DATA statements.
- If you add or replace data using HDFS operations, issue the REFRESH command in `impala-shell` so that Impala recognizes the changes in data files, block locations, and so on.
- When you issue a DROP TABLE statement in Impala, that removes the connection that Impala has with the associated data files, but does not physically remove the underlying data. You can continue to use the data files with other Hadoop components and HDFS operations.
- When you issue an ALTER TABLE statement to rename an external table, all data files are left in their original locations.
- You can point multiple external tables at the same HDFS directory by using the same LOCA TION attribute for each one. The tables could have different column definitions, as long as the number and types of columns are compatible with the schema evolution considerations for the underlying file type. For example, for text data files, one table might define a certain column as a STRING while another defines the same column as a BIGINT.

You can switch a table between internal to external by using the ALTER TABLE statement with the SET TBLPRO PERTIES clause. For example:

```
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='TRUE');
```

If the Kudu service is integrated with the Hive Metastore, the above operations are not supported.

File formats

Each table has an associated file format, which determines how Impala interprets the associated data files.

You set the file format during the CREATE TABLE statement, or change it later using the ALTER TABLE statement.

Partitioned tables can have a different file format for individual partitions, allowing you to change the file format used in your ETL process for new data without going back and reconverting all the existing data in the same table.

Any INSERT statements produce new data files with the current file format of the table.

For existing data files, changing the file format of the table does not automatically do any data conversion. You must use TRUNCATE TABLE or INSERT OVERWRITE to remove any previous data files that use the old file format. Then you use the LOAD DATA statement, INSERT ... SELECT, or other mechanism to put data files of the correct format into the table.

The default file format, Parquet, offers the highest query performance and uses compression to reduce storage requirements; therefore, where practical, use Parquet for Impala tables with substantial amounts of data. Also, the complex types (ARRAY, STRUCT, and MAP) available in Impala 2.3 and higher are currently only supported with the Parquet file type.

Kudu tables

By default, tables stored in Apache Kudu are treated specially, because Kudu manages its data independently of HDFS files.

All metadata that Impala needs is stored in the HMS.

When Kudu is not integrated with the HMS, when you create a Kudu table through Impala, the table is assigned an internal Kudu table name of the form impala::*db_name.table_name*. You can see the Kudu-assigned name in the output of DESCRIBE FORMATTED, in the kudu.table_name field of the table properties.

For Impala-Kudu managed tables, ALTER TABLE ... RENAME renames both the Impala and the Kudu table.

For Impala-Kudu external tables, ALTER TABLE ... RENAME renames just the Impala table. To change the Kudu table that an Impala external table points to, use ALTER TABLE *impala_name* SET TBLPROPERTIES('kudu.table_name' = '*different_kudu_table_name*'). The underlying Kudu table must already exist.

In practice, external tables are typically used to access underlying Kudu tables that were created outside of Impala, that is, through the Kudu API.

The SHOW TABLE STATS output for a Kudu table shows Kudu-specific details about the layout of the table. Instead of information about the number and sizes of files, the information is divided by the Kudu tablets. For each tablet, the output includes the fields # Rows (although this number is not currently computed), Start Key, Stop Key, Leader Replica, and # Replicas. The output of SHOW COLUMN STATS, illustrating the distribution of values within each column, is the same for Kudu tables as for HDFS-backed tables.

If the Kudu service is not integrated with the Hive Metastore, the distinction between internal and external tables has some special details for Kudu tables. Tables created entirely through Impala are internal tables. The table name as represented within Kudu includes notation such as an impala:: prefix and the Impala database name. External Kudu tables are those created by a non-Impala mechanism, such as a user application calling the Kudu APIs. For these tables, the CREATE EXTERNAL TABLE syntax lets you establish a mapping from Impala to the existing Kudu table:

```
CREATE EXTERNAL TABLE impala_name STORED AS KUDU
  TBLPROPERTIES('kudu.table_name' = 'original_kudu_name');
```

External Kudu tables differ in one important way from other external tables: adding or dropping a column or range partition changes the data in the underlying Kudu table, in contrast to an HDFS-backed external table where existing data files are left untouched.

Related Information

Hadoop file formats supported

Impala views

Views are lightweight logical constructs that act as aliases for queries. You can specify a view name in a query (a SELECT statement or the SELECT portion of an INSERT statement) where you would usually specify a table name.

A view lets you to:

- Issue complicated reporting queries with compact and simple syntax:
- Reduce maintenance by avoiding the duplication of complicated queries across multiple applications in multiple languages:

```
CREATE VIEW v2 AS SELECT t1.c1, t1.c2, t2.c3 FROM t1 JOIN t2 ON (t1.id =
 t2.id);
-- This simple query is safer to embed in reporting applications than the
 longer query above.
-- The view definition can remain stable even if the structure of the und
erlying tables changes.
SELECT c1, c2, c3 FROM v2;
```

- Build a new, more refined query on top of the original query by adding new clauses, select-list expressions, function calls, and so on: This technique lets you build up several more or less granular variations of the same query, and switch between them when appropriate.
- Set up aliases with intuitive names for tables, columns and result sets from joins.
- Avoid coding lengthy subqueries and repeating the same subquery text in many other queries.

To create, alter, or drop views, use the CREATE VIEW, the ALTER VIEW, and the DROP VIEW statements respectively.

You cannot insert into views.

To see the definition of a view, issue a DESCRIBE FORMATTED statement, which shows the query from the original CREATE VIEW statement

Impala SQL data types

Impala supports a set of data types that you can use for table columns, expression values, and function arguments and return values.

Note: Currently, Impala supports only scalar types, not composite or nested types. Accessing a table containing any columns with unsupported types causes an error.

For the notation to write literals of each of these data types, see Literals.

Impala supports a limited set of implicit casts to avoid undesired results from unexpected casting behavior.

- Impala does not implicitly cast between string and numeric or Boolean types. Always use CAST() for these conversions.
- Impala does perform implicit casts among the numeric types, when going from a smaller or less precise type to a larger or more precise one. For example, Impala will implicitly convert a SMALLINT to a BIGINT or FLOAT, but to convert from DOUBLE to FLOAT or INT to TINYINT requires a call to CAST() in the query.
- Impala does perform implicit casts from STRING to TIMESTAMP. Impala has a restricted set of literal formats for the TIMESTAMP data type and the FROM_UNIXTIME() format string; see TIMESTAMP data type on page 40 for details.

See the topics under this section for full details on implicit and explicit casting for each data type, and see Impala type conversion functions on page 308 for details about the CAST() function.

Related Information

Impala SQL literals

SQL differences between Impala and Hive

ARRAY complex type

A complex data type that can represent an arbitrary number of ordered elements. The elements can be scalars or another complex type (ARRAY, STRUCT, or MAP).

Syntax:

```
column_name ARRAY < type >

type ::= primitive_type | complex_type
```

Usage:

Because complex types are often used in combination, for example an ARRAY of STRUCT elements, if you are unfamiliar with the Impala complex types, start with Complex types on page 49 for background information and usage examples.

The elements of the array have no names. You refer to the value of the array item using the ITEM pseudocolumn, or its position in the array with the POS pseudocolumn.

Each row can have a different number of elements (including none) in the array for that row.

When an array contains items of scalar types, you can use aggregation functions on the array elements without using join notation. For example, you can find the COUNT(), AVG(), SUM(), and so on of numeric array elements, or the MAX() and MIN() of any scalar array elements by referring to *table_name.array_column* in the FROM clause of the query. When you need to cross-reference values from the array with scalar values from the same row, such as by including a GROUP BY clause to produce a separate aggregated result for each row, then the join clause is required.

A common usage pattern with complex types is to have an array as the top-level type for the column: an array of structs, an array of maps, or an array of arrays. For example, you can model a denormalized table by creating a column that is an ARRAY of STRUCT elements; each item in the array represents a row from a table that would normally be used in a join query. This kind of data structure lets you essentially denormalize tables by associating multiple rows from one table with the matching row in another table.

You typically do not create more than one top-level ARRAY column, because if there is some relationship between the elements of multiple arrays, it is convenient to model the data as an array of another complex type element (either STRUCT or MAP).

You can pass a multi-part qualified name to DESCRIBE to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table T1 contains an ARRAY column A1, you could issue the statement DESCRIBE t1.a1. If table T1 contained a STRUCT column S1, and a field F1 within the STRUCT was a MAP, you could issue the statement DESCRIBE t1.s1.f1. An ARRAY is shown as a two-column table, with ITEM and POS columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

Restrictions:

- Columns with this data type can only be used in tables or partitions with the Parquet file format.
- Columns with this data type cannot be used as partition key columns in a partitioned table.
- The COMPUTE STATS statement does not produce any statistics for columns of this data type.
- The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.
- See the *Limitations and restrictions for complex types* topic for a full list of limitations and associated guidelines about complex type columns.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Examples:

Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types for the table definitions.

The following example shows how to construct a table with various kinds of ARRAY columns, both at the top level and nested within other complex types. Whenever the ARRAY consists of a scalar value, such as in the PETS column or the CHILDREN field, you can see that future expansion is limited. For example, you could not easily evolve the schema to record the kind of pet or the child's birthday alongside the name. Therefore, it is more common to use an ARRAY whose elements are of STRUCT type, to associate multiple fields with each array element.

Note: Practice the CREATE TABLE and query notation for complex type columns using empty tables, until you can visualize a complex data structure and construct corresponding SQL statements reliably.

```
CREATE TABLE array_demo
(
  id BIGINT,
  name STRING,
-- An ARRAY of scalar type as a top-level column.
  pets ARRAY <STRING>,

-- An ARRAY with elements of complex type (STRUCT).
  places_lived ARRAY < STRUCT <
    place: STRING,
    start_year: INT
  >>,

-- An ARRAY as a field (CHILDREN) within a STRUCT.
-- (The STRUCT is inside another ARRAY, because it is rare
-- for a STRUCT to be a top-level column.)
  marriages ARRAY < STRUCT <
    spouse: STRING,
    children: ARRAY <STRING>
  >>,

-- An ARRAY as the value part of a MAP.
```

```
-- The first MAP field (the key) would be a value such as
-- 'Parent' or 'Grandparent', and the corresponding array would
-- represent 2 parents, 4 grandparents, and so on.
  ancestors MAP < STRING, ARRAY <STRING> >
)
STORED AS PARQUET;
```

The following example shows how to examine the structure of a table containing one or more ARRAY columns by using the DESCRIBE statement. You can visualize each ARRAY as its own two-column table, with columns ITEM and POS.

```
DESCRIBE array_demo;
+--------------+--------------------------+
| name         | type                     |
+--------------+--------------------------+
| id           | bigint                   |
| name         | string                   |
| pets         | array<string>            |
| marriages    | array<struct<            |
|              |   spouse:string,         |
|              |   children:array<string> |
|              | >>                       |
| places_lived | array<struct<            |
|              |   place:string,          |
|              |   start_year:int         |
|              | >>                       |
| ancestors    | map<string,array<string>> |
+--------------+--------------------------+

DESCRIBE array_demo.pets;
+------+--------+
| name | type   |
+------+--------+
| item | string |
| pos  | bigint |
+------+--------+
DESCRIBE array_demo.marriages;
+------+-------------------------+
| name | type                    |
+------+-------------------------+
| item | struct<                 |
|      |   spouse:string,        |
|      |   children:array<string> |
|      | >                       |
| pos  | bigint                  |
+------+-------------------------+
DESCRIBE array_demo.places_lived;
+------+------------------+
| name | type             |
+------+------------------+
| item | struct<          |
|      |   place:string,  |
|      |   start_year:int |
|      | >                |
| pos  | bigint           |
+------+------------------+

DESCRIBE array_demo.ancestors;
+-------+---------------+
| name  | type          |
+-------+---------------+
| key   | string        |
```

```
| value | array<string> |
+-------+---------------+
```

The following example shows queries involving ARRAY columns containing elements of scalar or complex types. You "unpack" each ARRAY column by referring to it in a join query, as if it were a separate table with ITEM and POS columns. If the array element is a scalar type, you refer to its value using the ITEM pseudocolumn. If the array element is a STRUCT, you refer to the STRUCT fields using dot notation and the field names. If the array element is another ARRAY or a MAP, you use another level of join to unpack the nested collection elements.

```
-- Array of scalar values.
-- Each array element represents a single string, plus we know its position
 in the array.
SELECT id, name, pets.pos, pets.item FROM array_demo, array_demo.pets;
-- Array of structs.
-- Now each array element has named fields, possibly of different types.
-- You can consider an ARRAY of STRUCT to represent a table inside another
 table.
SELECT id, name, places_lived.pos, places_lived.item.place, places_lived.it
em.start_year
FROM array_demo, array_demo.places_lived;
-- The .ITEM name is optional for array elements that are structs.
-- The following query is equivalent to the previous one, with .ITEM
-- removed from the column references.
SELECT id, name, places_lived.pos, places_lived.place, places_lived.start_ye
ar
  FROM array_demo, array_demo.places_lived;

-- To filter specific items from the array, do comparisons against the .POS
or .ITEM
-- pseudocolumns, or names of struct fields, in the WHERE clause.
SELECT id, name, pets.item FROM array_demo, array_demo.pets
  WHERE pets.pos in (0, 1, 3);

SELECT id, name, pets.item FROM array_demo, array_demo.pets
  WHERE pets.item LIKE 'Mr. %';

SELECT id, name, places_lived.pos, places_lived.place, places_lived.start_ye
ar
  FROM array_demo, array_demo.places_lived
WHERE places_lived.place like '%California%';
```

BIGINT data type

An 8-byte integer data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name BIGINT
```

Range: -9223372036854775808 .. 9223372036854775807. There is no UNSIGNED subtype.

Conversions: Impala automatically converts to a floating-point type (FLOAT or DOUBLE) automatically. Use CAST () to convert to TINYINT, SMALLINT, INT, STRING, or TIMESTAMP. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting ##use_local_tz_for_unix_timestam p_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Examples:

```
CREATE TABLE t1 (x BIGINT);
SELECT CAST(1000 AS BIGINT);
```

Usage notes:

BIGINT is a convenient type to use for column declarations because you can use any kind of integer values in INSERT statements and they are promoted to BIGINT where necessary. However, BIGINT also requires the most bytes of any integer type on disk and in memory, meaning your queries are not as efficient and scalable as possible if you overuse this type. Therefore, prefer to use the smallest integer type with sufficient range to hold all input values, and CAST() when necessary to the appropriate type.

For a convenient and automated way to check the bounds of the BIGINT type, call the functions MIN_BIGINT() and MAX_BIGINT().

If an integer value is too large to be represented as a BIGINT, use a DECIMAL instead with sufficient digits of precision.

NULL considerations: Casting any non-numeric value to this type produces a NULL value.

Partitioning: Prefer to use this type for a partition key column. Impala can process the numeric type more efficiently than a STRING representation of the value.

HBase considerations: This data type is fully compatible with HBase tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as an 8-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

BOOLEAN data type

A data type used in CREATE TABLE and ALTER TABLE statements, representing a single true/false choice.

Syntax: In the column definition of a CREATE TABLE statement:

```
column_name BOOLEAN
```

Range: TRUE or FALSE. Do not use quotation marks around the TRUE and FALSE literal values. You can write the literal values in uppercase, lowercase, or mixed case. The values queried from a table are always returned in lowercase, true or false.

Conversions: Impala does not automatically convert any other type to BOOLEAN. All conversions must use an explicit call to the CAST() function.

You can use CAST() to convert any integer or floating-point type to BOOLEAN: a value of 0 represents false, and any non-zero value is converted to true.

```
SELECT CAST(42 AS BOOLEAN) AS nonzero_int, CAST(99.44 AS BOOLEAN) AS nonzero
_decimal,
  CAST(000 AS BOOLEAN) AS zero_int, CAST(0.0 AS BOOLEAN) AS zero_decimal;
+-------------+-----------------+----------+--------------+
| nonzero_int | nonzero_decimal | zero_int | zero_decimal |
+-------------+-----------------+----------+--------------+
| true        | true            | false    | false        |
+-------------+-----------------+----------+--------------+
```

When you cast the opposite way, from BOOLEAN to a numeric type, the result becomes either 1 or 0:

```
SELECT CAST(true AS INT) AS true_int, CAST(true AS DOUBLE) AS true_double,
  CAST(false AS INT) AS false_int, CAST(false AS DOUBLE) AS false_double;
+----------+-------------+-----------+--------------+
| true_int | true_double | false_int | false_double |
+----------+-------------+-----------+--------------+
| 1        | 1           | 0         | 0            |
+----------+-------------+-----------+--------------+
```

You can cast DECIMAL values to BOOLEAN, with the same treatment of zero and non-zero values as the other numeric types. You cannot cast a BOOLEAN to a DECIMAL.

You cannot cast a STRING value to BOOLEAN, although you can cast a BOOLEAN value to STRING, returning '1' for true values and '0' for false values.

Although you can cast a TIMESTAMP to a BOOLEAN or a BOOLEAN to a TIMESTAMP, the results are unlikely to be useful. Any non-zero TIMESTAMP (that is, any value other than 1970-01-01 00:00:00) becomes TRUE when converted to BOOLEAN, while 1970-01-01 00:00:00 becomes FALSE. A value of FALSE becomes 1970-01-01 00:00:00 when converted to BOOLEAN, and TRUE becomes one second past this epoch date, that is, 1970-01-01 00:00:01.

NULL considerations: An expression of this type produces a NULL value if any argument of the expression is NULL.

Partitioning:

Do not use a BOOLEAN column as a partition key. Although you can create such a table, subsequent operations produce errors:

```
[localhost:21000] > create table truth_table (assertion string) partitioned
by (truth boolean);
[localhost:21000] > insert into truth_table values ('Pigs can fly',false);
ERROR: AnalysisException: INSERT into table with BOOLEAN partition column
 (truth) is not supported: partitioning.truth_table
```

Examples:

```
SELECT 1 < 2;
SELECT 2 = 5;
SELECT 100 < NULL, 100 > NULL;
CREATE TABLE assertions (claim STRING, really BOOLEAN);
INSERT INTO assertions VALUES
  ("1 is less than 2", 1 < 2),
  ("2 is the same as 5", 2 = 5),
  ("Grass is green", true),
  ("The moon is made of green cheese", false);
SELECT claim FROM assertions WHERE really = TRUE;
```

HBase considerations: This data type is fully compatible with HBase tables.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Related Information

BOOLEAN literals

Impala SQL operators
Impala conditional functions

CHAR data type

A fixed-length character type, padded with trailing spaces if necessary to achieve the specified length. If values are longer than the specified length, Impala truncates any trailing characters.

Syntax: In the column definition of a CREATE TABLE statement:

```
column_name CHAR(length)
```

The maximum *length* you can specify is 255.

Semantics of trailing spaces:

- When you store a CHAR value shorter than the specified length in a table, queries return the value padded with trailing spaces if necessary; the resulting value has the same length as specified in the column definition.
- Leading spaces in CHAR are preserved within the data file.
- If you store a CHAR value containing trailing spaces in a table, those trailing spaces are not stored in the data file. When the value is retrieved by a query, the result could have a different number of trailing spaces. That is, the value includes however many spaces are needed to pad it to the specified length of the column.
- If you compare two CHAR values that differ only in the number of trailing spaces, those values are considered identical.
- When comparing or processing CHAR values:
  - CAST() truncates any longer string to fit within the defined length. For example:

    ```
    SELECT CAST('x' AS CHAR(4)) = CAST('x        ' AS CHAR(4)); -- Returns T
    RUE.
    ```

  - If a CHAR value is shorter than the specified length, it is padded on the right with spaces until it matches the specified length.
  - CHAR_LENGTH() returns the length including any trailing spaces.
  - LENGTH() returns the length excluding trailing spaces.
  - CONCAT() returns the length including trailing spaces.

Partitioning: This type can be used for partition key columns. Because of the efficiency advantage of numeric values over character-based values, if the partition key is a string representation of a number, prefer to use an integer type with sufficient range (INT, BIGINT, and so on) where practical.

HBase considerations: This data type cannot be used with HBase tables.

Parquet considerations:

- This type can be read from and written to Parquet files.
- There is no requirement for a particular level of Parquet.
- Parquet files generated by Impala and containing this type can be freely interchanged with other components such as Hive and MapReduce.
- Any trailing spaces, whether implicitly or explicitly specified, are not written to the Parquet data files.
- Parquet data files might contain values that are longer than allowed by the CHAR(*n*) length limit. Impala ignores any extra trailing characters when it processes those values during a query.

Text table considerations: Text data files might contain values that are longer than allowed for a particular CHAR(*n*) column. Any extra trailing characters are ignored when Impala processes those values during a query. Text data files can also contain values that are shorter than the defined length limit, and Impala pads them with trailing spaces up to the specified length. Any text data files produced by Impala INSERT statements do not include any trailing blanks for CHAR columns.

Avro considerations: The Avro specification allows string values up to 2**64 bytes in length. Impala queries for Avro tables use 32-bit integers to hold string lengths. In Impala 2.5 and higher, Impala truncates CHAR and VARCHAR

values in Avro tables to (2**31)-1 bytes. If a query encounters a STRING value longer than (2**31)-1 bytes in an Avro table, the query fails. In earlier releases, encountering such long values in an Avro table could cause a crash.

Compatibility: This type is available using or higher.

Some other database systems make the length specification optional. For Impala, the length is required.

Internal details: Represented in memory as a byte array with the same size as the length specification. Values that are shorter than the specified length are padded on the right with trailing spaces.

Added in: Impala 2.0.0

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

UDF considerations: This type cannot be used for the argument or return type of a user-defined function (UDF) or user-defined aggregate function (UDA).

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Performance consideration: The CHAR type currently does not have the Impala Codegen support, and we recommend using VARCHAR or STRING over CHAR as the performance gain of Codegen outweighs the benefits of fixed width CHAR.

Restrictions: Because the blank-padding behavior requires allocating the maximum length for each value in memory, for scalability reasons, you should avoid declaring CHAR columns that are much longer than typical values in that column.

All data in CHAR and VARCHAR columns must be in a character encoding that is compatible with UTF-8. If you have binary data from another database system (that is, a BLOB type), use a STRING column to hold it.

When an expression compares a CHAR with a STRING or VARCHAR, the CHAR value is implicitly converted to STRING first, with trailing spaces preserved.

This behavior differs from other popular database systems. To get the expected result of TRUE, cast the expressions on both sides to CHAR values of the appropriate length. For example:

```
SELECT CAST("foo  " AS CHAR(5)) = CAST('foo' AS CHAR(3)); -- Returns TRUE.
```

This behavior is subject to change in future releases.

DATE data type

Use the DATE data type to store date values.

Use the DATE data type to store date values. The DATE type is supported for HBase, Text, Avro, Kudu and Parquet.

Range:

0001-01-01 to 9999-12-31

Literals and expressions:

The DATE literals are in the form of DATE'YYYY-MM-DD'. For example, DATE '2013-01-01'

Parquet and Avro considerations:

Parquet and Avro use DATE logical type for dates. The DATE logical type annotates an INT32 that stores the number of days from the Unix epoch, January 1, 1970. This representation introduces an interoperability issue between Impala and older versions of Hive:

If Hive versions lower than 3.1 wrote dates earlier than 1582-10-15 to a Parquet or Avro table, those dates will be read back incorrectly by Impala and vice versa. In Hive 3.1 and higher, this is no longer an issue.

Explicit casting between DATE and other data types:

DATE type can only be converted to/from DATE, TIMESTAMP, or STRING types as described below.

Cast from	Cast to	Result
TIMESTAMP	DATE	The date component of the TIMESTAMP is returned, and the time of the day component of the TIMESTAMP is ignored.
STRING	DATE	The DATE value of yyyy-MM-dd is returned. The STRING value must be in the yyyy-MM-dd or yyyy-MM-dd HH:mm:ss.SSSS SSSSS pattern. If the time component is present in STRING, it is silently ignored. If the STRING value does not match the above formats, an error is returned.
DATE	TIMESTAMP	The year, month, and day of the DATE is returned along with the time of day component set to 00:00:00.
DATE	STRING	The STRING value, 'yyyy-MM-dd', is returned.

Implicit casting between DATE and other types:

Implicit casting is supported:

- From STRING to DATE if the source STRING value is in the yyyy-MM-dd or yyyy-MM-dd HH:mm:ss.SSSS SSSSS pattern.
- From DATE to TIMESTAMP.

Added in:

The DATE type is available in Impala 3.3 and higher.

Kudu considerations:

You can read and write DATE values to Kudu tables.

DECIMAL data type

The DECIMAL data type is a numeric data type with fixed scale and precision. The data type is useful for storing and doing operations on precise decimal values.

Syntax:

```
DECIMAL[(precision[, scale])]
```

Precision:

precision represents the total number of digits that can be represented regardless of the location of the decimal point.

This value must be between 1 and 38, specified as an integer literal.

The default precision is 9.

Scale:

scale represents the number of fractional digits.

This value must be less than or equal to the precision, specified as an integer literal.

The default scale is 0.

When the precision and the scale are omitted, a DECIMAL is treated as DECIMAL(9, 0).

Range:

The range of DECIMAL type is -10^38 +1 through 10^38 –1.

The largest value is represented by DECIMAL(38, 0).

The most precise fractional value (between 0 and 1, or 0 and -1) is represented by DECIMAL(38, 38), with 38 digits to the right of the decimal point. The value closest to 0 would be .0000...1 (37 zeros and the final 1). The value closest to 1 would be .999... (9 repeated 38 times).

Memory and disk storage:

Only the precision determines the storage size for DECIMAL values, and the scale setting has no effect on the storage size. The following table describes the in-memory storage once the values are loaded into memory.

Precision	In-memory storage
1 - 9	4 bytes
10 - 18	8 bytes
19 - 38	10 bytes

The on-disk representation varies depending on the file format of the table.

Text, RCFile, and SequenceFile tables use ASCII-based formats as below:

- Leading zeros are not stored.
- Trailing zeros are stored.
- Each DECIMAL value takes up as many bytes as the precision of the value, plus:
  - One extra byte if the decimal point is present.
  - One extra byte for negative values.

Parquet and Avro tables use binary formats and offer more compact storage for DECIMAL values. In these tables, Impala stores each value in fewer bytes where possible depending on the precision specified for the DECIMAL column. To conserve space in large tables, use the smallest-precision DECIMAL type.

Precision and scale in arithmetic operations:

For all arithmetic operations, the resulting precision is at most 38.

If the resulting precision would be greater than 38, Impala truncates the result from the back, but keeps at least 6 fractional digits in scale and rounds.

For example, DECIMAL(38, 20) * DECIMAL(38, 20) returns DECIMAL(38, 6). According to the table below, the resulting precision and scale would be (77, 40), but they are higher than the maximum precision and scale for DECI MAL. So, Impala sets the precision to the maximum allowed 38, and truncates the scale to 6.

When you use DECIMAL values in arithmetic operations, the precision and scale of the result value are determined as follows. For better readability, the following terms are used in the table below:

- P1, P2: Input precisions
- S1, S2: Input scales
- L1, L2: Leading digits in input DECIMALs, i.e., L1 = P1 - S1 and L2 = P2 - S2

Operation	Resulting precision	Resulting scale
Addition and Subtraction	max (L1, L2) + max (S1, S2) + 1 1 is for carry-over.	max (S1, S2)
Multiplication	P1 + P2 + 1	S1 + S2
Division	L1 + S2 + max (S1 + P2 + 1, 6)	max (S1 + P2 + 1, 6)
Modulo	min (L1, L2) + max (S1, S2)	max (S1, S2)

Precision and scale in functions:

When you use DECIMAL values in built-in functions, the precision and scale of the result value are determined as follows:

- The result of the SUM aggregate function on a DECIMAL value is:
  - Precision: 38
  - Scale: The same scale as the input column
- The result of AVG aggregate function on a DECIMAL value is:
  - Precision: 38
  - Scale: max(Scale of input column, 6)

Implicit conversions in DECIMAL assignments:

Impala enforces strict conversion rules in decimal assignments like in INSERT and UNION statements, or in functions like COALESCE.

If there is not enough precision and scale in the destination, Impala fails with an error.

Impala performs implicit conversions between DECIMAL and other numeric types as below:

- DECIMAL is implicitly converted to DOUBLE or FLOAT when necessary even with a loss of precision. It can be necessary, for example when inserting a DECIMAL value into a DOUBLE column. For example:

  ```
  CREATE TABLE flt(c FLOAT);
  INSERT INTO flt SELECT CAST(1e37 AS DECIMAL(38, 0));
  SELECT CAST(c AS DECIMAL(38, 0)) FROM flt;

  Result: 99999999338158125107115063762579619 84
  ```

  The result has a loss of information due to implicit casting. This is why we discourage using the DOUBLE and FLOAT types in general.
- DOUBLE and FLOAT cannot be implicitly converted to DECIMAL. An error is returned.
- DECIMAL is implicitly converted to DECIMAL if all digits fit in the resulting DECIMAL.

  For example, the following query returns an error because the resulting type that guarantees that all digits fit cannot be determined .

  ```
  SELECT GREATEST (CAST(1 AS DECIMAL(38, 0)), CAST(2 AS DECIMAL(38, 37)));
  ```
- Integer values can be implicitly converted to DECIMAL when there is enough room in the DECIMAL to guarantee that all digits fit. The integer types require the following numbers of digits to the left of the decimal point when converted to DECIMAL:
  - BIGINT: 19 digits
  - INT: 10 digits
  - SMALLINT: 5 digits
  - TINYINT: 3 digits

  For example:

  ```
  CREATE TABLE decimals_10_8 (x DECIMAL(10, 8));
  INSERT INTO decimals_10_8 VALUES (CAST(1 AS TINYINT));
  ```

  The above INSERT statement fails because TINYINT requires room for 3 digits to the left of the decimal point in the DECIMAL.

  ```
  CREATE TABLE decimals_11_8(x DECIMAL(11, 8));
  INSERT INTO decimals_11_8 VALUES (CAST(1 AS TINYINT));
  ```

  The above INSERT statement succeeds because there is enough room for 3 digits to the left of the decimal point that TINYINT requires.

In UNION, the resulting precision and scales are determined as follows.

- Precision: max (L1, L2) + max (S1, S2)

  If the resulting type does not fit in the DECIMAL type, an error is returned. See the first example below.
- Scale: max (S1, S2)

Examples for UNION:

- DECIMAL(20, 0) UNION DECIMAL(20, 20) would require a DECIMAL(40, 20) to fit all the digits. Since this is larger than the max precision for DECIMAL, Impala returns an error. One way to fix the error is to cast both operands to the desired type, for example DECIMAL(38, 18).
- DECIMAL(20, 2) UNION DECIMAL(8, 6) returns DECIMAL(24, 6).
- INT UNION DECIMAL(9, 4) returns DECIMAL(14, 4).

  INT has the precision 10 and the scale 0, so it is treated as DECIMAL(10, 0) UNION DECIMAL(9. 4).

Casting between DECIMAL and other data types:

To avoid potential conversion errors, use CAST to explicitly convert between DECIMAL and other types in decimal assignments like in INSERT and UNION statements, or in functions like COALESCE:

- You can cast the following types to DECIMAL: FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING
- You can cast DECIMAL to the following types: FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING, BOOLEAN, TIMESTAMP

Impala performs CAST between DECIMAL and other numeric types as below:

- Precision: If you cast a value with bigger precision than the precision of the destination type, Impala returns an error. For example, CAST(123456 AS DECIMAL(3,0)) returns an error because all digits do not fit into DECIMAL(3, 0)
- Scale: If you cast a value with more fractional digits than the scale of the destination type, the fractional digits are rounded. For example, CAST(1.239 AS DECIMAL(3, 2)) returns 1.24.

Casting STRING to DECIMAL:

You can cast STRING of numeric characters in columns, literals, or expressions to DECIMAL as long as number fits within the specified target DECIMAL type without overflow.

- If scale in STRING > scale in DECIMAL, the fractional digits are rounded to the DECIMAL scale.

  For example, CAST('98.678912' AS DECIMAL(15, 1)) returns 98.7.
- If # leading digits in STRING > # leading digits in DECIMAL, an error is returned.

  For example, CAST('123.45' AS DECIMAL(2, 2)) returns an error.

Exponential notation is supported when casting from STRING.

For example, CAST('1.0e6' AS DECIMAL(32, 0)) returns 1000000.

Casting any non-numeric value, such as 'ABC' to the DECIMAL type returns an error.

Casting DECIMAL to TIMESTAMP:

Casting a DECIMAL value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970).

DECIMAL vs FLOAT consideration:

The FLOAT and DOUBLE types can cause problems or unexpected behavior due to inability to precisely represent certain fractional values, for example dollar and cents values for currency. You might find output values slightly different than you inserted, equality tests that do not match precisely, or unexpected values for GROUP BY columns. The DECIMAL type can help reduce unexpected behavior and rounding errors, but at the expense of some performance overhead for assignments and comparisons.

Literals and expressions:

- Numeric literals without a decimal point
  - The literals are treated as the smallest integer that would fit the literal. For example, 111 is a TINYINT, and 1111 is a SMALLINT.
  - Large literals that do not fit into any integer type are treated as DECIMAL.
  - The literals too large to fit into a DECIMAL(38, 0) are treated as DOUBLE.
- Numeric literals with a decimal point
  - The literal with less than 38 digits are treated as DECIMAL.
  - The literals with 38 or more digits are treated as a DOUBLE.
- Exponential notation is supported in DECIMAL literals.
- To represent a very large or precise DECIMAL value as a literal, for example one that contains more digits than can be represented by a BIGINT literal, use a quoted string or a floating-point value for the number and CAST the string to the desired DECIMAL type.

  For example: CAST('999999999999999999999999999999' AS DECIMAL(38, 5)))

File format considerations:

The DECIMAL data type can be stored in any of the file formats supported by Impala:

- Impala can query Avro, RCFile, or SequenceFile tables that contain DECIMAL columns, created by other Hadoop components.
- Impala can query and insert into Kudu tables that contain DECIMAL columns. Kudu supports the DECIMAL type.
- The DECIMAL data type is fully compatible with HBase tables.
- The DECIMAL data type is fully compatible with Parquet tables.
- Values of the DECIMAL data type are potentially larger in text tables than in tables using Parquet or other binary formats.

UDF consideration:

When writing a C++ UDF, use the DecimalVal data type defined in /usr/include/impala_udf/udf.h.

Changing precision and scale:

You can issue an ALTER TABLE ... REPLACE COLUMNS statement to change the precision and scale of an existing DECIMAL column.

- For text-based formats (text, RCFile, and SequenceFile tables)
  - If the values in the column fit within the new precision and scale, they are returned correctly by a query.
  - If any values that do not fit within the new precision and scale:
    - Impala returns an error if the query option ABORT_ON_ERROR is set to true.
    - Impala returns a NULL and warning that conversion failed if the query option ABORT_ON_ERROR is set to false.
  - Leading zeros do not count against the precision value, but trailing zeros after the decimal point do.
- For binary formats (Parquet and Avro tables)
  - Although an ALTER TABLE ... REPLACE COLUMNS statement that changes the precision or scale of a DECIMAL column succeeds, any subsequent attempt to query the changed column results in a fatal error. This is because the metadata about the columns is stored in the data files themselves, and ALTER TABLE does not actually make any updates to the data files. The other unaltered columns can still be queried successfully.
  - If the metadata in the data files disagrees with the metadata in the metastore database, Impala cancels the query.

Partitioning:

Using a DECIMAL column as a partition key provides you a better match between the partition key values and the HDFS directory names than using a DOUBLE or FLOAT partitioning column.

Column statistics considerations:

Because the DECIMAL type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Compatibility with older version of DECIMAL:

This version of DECIMAL type is the default in and higher. The key differences between this version of DECIMAL and the previous DECIMAL V1 in Impala 2.x include the following:

Property	Impala 3.0 or higher	Impala 2.12 or lower
Overall behavior	Returns either the result or an error.	Returns either the result or NULL with a warning.
Overflow behavior	Aborts with an error.	Issues a warning and returns NULL.
Truncation / rounding behavior in arithmetic	Truncates and rounds digits from the back.	Truncates digits from the front.
String cast	Truncates from the back and rounds.	Truncates from the back.

If you need to continue using the first version of the DECIMAL type for the backward compatibility of your queries, set the DECIMAL_V2 query option to FALSE:

```
SET DECIMAL_V2=FALSE;
```

Compatibility with other databases:

Use the DECIMAL data type in Impala for applications where you used the NUMBER data type in Oracle.

The Impala DECIMAL type does not support the Oracle idioms of * for scale.

The Impala DECIMAL type does not support negative values for precision.

DOUBLE data type

A double precision floating-point data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name DOUBLE
```

Range: 4.94065645841246544e-324d .. 1.79769313486231570e+308, positive or negative

Precision: 15 to 17 significant digits, depending on usage. The number of significant digits does not depend on the position of the decimal point.

Representation: The values are stored in 8 bytes, using the IEEE 754 Double Precision Binary Floating Point format.

Conversions: Impala does not automatically convert DOUBLE to any other type. You can use CAST() to convert DOUBLE values to FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING, TIMESTAMP, or BOOLEAN. You can use exponential notation in DOUBLE literals or when casting from STRING, for example 1.0e6 to represent one million. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting ##use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

The data type REAL is an alias for DOUBLE.

Impala does not evaluate NaN (not a number) as equal to any other numeric values, including other NaN values. For example, the following statement, which evaluates equality between two NaN values, returns false:

```
SELECT CAST('nan' AS DOUBLE)=CAST('nan' AS DOUBLE);
```

Examples:

```
CREATE TABLE t1 (x DOUBLE);
SELECT CAST(1000.5 AS DOUBLE);
```

Partitioning: Because fractional values of this type are not always represented precisely, when this type is used for a partition key column, the underlying HDFS directories might not be named exactly as you expect. Prefer to partition on a DECIMAL column instead.

HBase considerations: This data type is fully compatible with HBase tables.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as an 8-byte value.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM() and AVG() for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

The inability to exactly represent certain floating-point values means that DECIMAL is sometimes a better choice than DOUBLE or FLOAT when precision is critical, particularly when transferring data from other database systems that use different representations or file formats.

Kudu considerations:

Currently, the data types BOOLEAN, FLOAT, and DOUBLE cannot be used for primary key columns in Kudu tables.

Related Information

Impala mathematical functions

FLOAT data type

Impala SQL literals

FLOAT data type

A single precision floating-point data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name FLOAT
```

Range: 1.40129846432481707e-45 .. 3.40282346638528860e+38, positive or negative

Precision: 6 to 9 significant digits, depending on usage. The number of significant digits does not depend on the position of the decimal point.

Representation: The values are stored in 4 bytes, using the *IEEE 754 Single Precision Binary Floating Point* format.

Conversions: Impala automatically converts FLOAT to more precise DOUBLE values, but not the other way around. You can use CAST() to convert FLOAT values to TINYINT, SMALLINT, INT, BIGINT, STRING, TIMESTAMP, or BOOLEAN. You can use exponential notation in FLOAT literals or when casting from STRING, for example 1.0e6 to represent one million. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time

in the UTC time zone. If the setting ##use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

Impala does not evaluate NaN (not a number) as equal to any other numeric values, including other NaN values. For example, the following statement, which evaluates equality between two NaN values, returns false:

```
SELECT CAST('nan' AS FLOAT)=CAST('nan' AS FLOAT);
```

Examples:

```
CREATE TABLE t1 (x FLOAT);
SELECT CAST(1000.5 AS FLOAT);
```

Partitioning: Because fractional values of this type are not always represented precisely, when this type is used for a partition key column, the underlying HDFS directories might not be named exactly as you expect. Prefer to partition on a DECIMAL column instead.

HBase considerations: This data type is fully compatible with HBase tables.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 4-byte value.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM() and AVG() for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

The inability to exactly represent certain floating-point values means that DECIMAL is sometimes a better choice than DOUBLE or FLOAT when precision is critical, particularly when transferring data from other database systems that use different representations or file formats.

Kudu considerations:

Currently, the data types BOOLEAN, FLOAT, and DOUBLE cannot be used for primary key columns in Kudu tables.

Related Information

IEEE 754 Single Precision Binary Floating Point

Impala mathematical functions

Impala SQL literals

INT data type

A 4-byte integer data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name INT
```

Range: -2147483648 .. 2147483647. There is no UNSIGNED subtype.

Conversions: Impala automatically converts to a larger integer type (BIGINT) or a floating-point type (FLOAT or DOUBLE) automatically. Use CAST() to convert to TINYINT, SMALLINT, STRING, or TIMESTAMP. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting ##use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

The data type INTEGER is an alias for INT.

For a convenient and automated way to check the bounds of the INT type, call the functions MIN_INT() and MAX_INT().

If an integer value is too large to be represented as a INT, use a BIGINT instead.

NULL considerations: Casting any non-numeric value to this type produces a NULL value.

Examples:

```
CREATE TABLE t1 (x INT);
SELECT CAST(1000 AS INT);
```

Partitioning: Prefer to use this type for a partition key column. Impala can process the numeric type more efficiently than a STRING representation of the value.

HBase considerations: This data type is fully compatible with HBase tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 4-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

MAP complex type

A complex data type representing an arbitrary set of key-value pairs. The key part is a scalar type, while the value part can be a scalar or another complex type (ARRAY, STRUCT, or MAP).

Syntax:

```
column_name MAP < primitive_type, type >

type ::= primitive_type | complex_type
```

Usage notes:

Because complex types are often used in combination, for example an ARRAY of STRUCT elements, if you are unfamiliar with the Impala complex types, start with Complex types on page 49 for background information and usage examples.

The MAP complex data type represents a set of key-value pairs. Each element of the map is indexed by a primitive type such as BIGINT or STRING, letting you define sequences that are not continuous or categories with arbitrary names. You might find it convenient for modelling data produced in other languages, such as a Python dictionary or Java HashMap, where a single scalar value serves as the lookup key.

In a big data context, the keys in a map column might represent a numeric sequence of events during a manufacturing process, or TIMESTAMP values corresponding to sensor observations. The map itself is inherently unordered, so you

choose whether to make the key values significant (such as a recorded TIMESTAMP) or synthetic (such as a random global universal ID).

Note: Behind the scenes, the MAP type is implemented in a similar way as the ARRAY type. Impala does not enforce any uniqueness constraint on the KEY values, and the KEY values are processed by looping through the elements of the MAP rather than by a constant-time lookup. Therefore, this type is primarily for ease of understanding when importing data and algorithms from non-SQL contexts, rather than optimizing the performance of key lookups.

You can pass a multi-part qualified name to DESCRIBE to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table T1 contains an ARRAY column A1, you could issue the statement DESCRIBE t1.a1. If table T1 contained a STRUCT column S1, and a field F1 within the STRUCT was a MAP, you could issue the statement DESCRIBE t1.s1.f1. An ARRAY is shown as a two-column table, with ITEM and POS columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

Added in: Impala 2.3.0

Restrictions:

- Columns with this data type can only be used in tables or partitions with the Parquet file format.
- Columns with this data type cannot be used as partition key columns in a partitioned table.
- The COMPUTE STATS statement does not produce any statistics for columns of this data type.
- The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.
- See the *Limitations and restrictions for complex types* topic for a full list of limitations and associated guidelines about complex type columns.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Examples:

Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types for the table definitions.

The following example shows a table with various kinds of MAP columns, both at the top level and nested within other complex types. Each row represents information about a specific country, with complex type fields of various levels of nesting to represent different information associated with the country: factual measurements such as area and population, notable people in different categories, geographic features such as cities, points of interest within each city, and mountains with associated facts. Practice the CREATE TABLE and query notation for complex type columns using empty tables, until you can visualize a complex data structure and construct corresponding SQL statements reliably.

```
create TABLE map_demo
(
  country_id BIGINT,

-- Numeric facts about each country, looked up by name.
-- For example, 'Area':1000, 'Population':999999.
-- Using a MAP instead of a STRUCT because there could be
-- a different set of facts for each country.
  metrics MAP <STRING, BIGINT>,
-- MAP whose value part is an ARRAY.
-- For example, the key 'Famous Politicians' could represent an array of 10
 elements,
-- while the key 'Famous Actors' could represent an array of 20 elements.
  notables MAP <STRING, ARRAY <STRING>>,

-- MAP that is a field within a STRUCT.
-- (The STRUCT is inside another ARRAY, because it is rare
```

```
-- for a STRUCT to be a top-level column.)
-- For example, city #1 might have points of interest with key 'Zoo',
-- representing an array of 3 different zoos.
-- City #2 might have completely different kinds of points of interest.
-- Because the set of field names is potentially large, and most entries c
ould be blank,
-- a MAP makes more sense than a STRUCT to represent such a sparse data s
tructure.
  cities ARRAY < STRUCT <
    name: STRING,
    points_of_interest: MAP <STRING, ARRAY <STRING>>
  >>,

-- MAP that is an element within an ARRAY. The MAP is inside a STRUCT field
 to associate
-- the mountain name with all the facts about the mountain.
-- The "key" of the map (the first STRING field) represents the name of s
ome fact whose value
-- can be expressed as an integer, such as 'Height', 'Year First Climbed',
and so on.
  mountains ARRAY < STRUCT < name: STRING, facts: MAP <STRING, INT > > >
)
STORED AS PARQUET;
```

```
DESCRIBE map_demo;
+------------+----------------------------------------------------+
| name       | type                                               |
+------------+----------------------------------------------------+
| country_id | bigint                                             |
| metrics    | map<string,bigint>                                 |
| notables   | map<string,array<string>>                          |
| cities     | array<struct<                                      |
|            |   name:string,                                     |
|            |   points_of_interest:map<string,array<string>>     |
|            | >>                                                 |
| mountains  | array<struct<                                      |
|            |   name:string,                                     |
|            |   facts:map<string,int>                            |
|            | >>                                                 |
+------------+----------------------------------------------------+

DESCRIBE map_demo.metrics;
+-------+--------+
| name  | type   |
+-------+--------+
| key   | string |
| value | bigint |
+-------+--------+

DESCRIBE map_demo.notables;
+-------+---------------+
| name  | type          |
+-------+---------------+
| key   | string        |
| value | array<string> |
+-------+---------------+

DESCRIBE map_demo.notables.value;
+------+--------+
| name | type   |
+------+--------+
| item | string |
```

```
| pos  | bigint |
+------+--------+

DESCRIBE map_demo.cities;
+------+-------------------------------------------------+
| name | type                                            |
+------+-------------------------------------------------+
| item | struct<                                         |
|      |   name:string,                                  |
|      |   points_of_interest:map<string,array<string>> |
|      | >                                               |
| pos  | bigint                                          |
+------+-------------------------------------------------+

DESCRIBE map_demo.cities.item.points_of_interest;
+-------+---------------+
| name  | type          |
+-------+---------------+
| key   | string        |
| value | array<string> |
+-------+---------------+

DESCRIBE map_demo.cities.item.points_of_interest.value;
+------+--------+
| name | type   |
+------+--------+
| item | string |
| pos  | bigint |
+------+--------+

DESCRIBE map_demo.mountains;
+------+-------------------------+
| name | type                    |
+------+-------------------------+
| item | struct<                 |
|      |   name:string,          |
|      |   facts:map<string,int> |
|      | >                       |
| pos  | bigint                  |
+------+-------------------------+

DESCRIBE map_demo.mountains.item.facts;
+-------+--------+
| name  | type   |
+-------+--------+
| key   | string |
| value | int    |
+-------+--------+
```

The following example shows a table that uses a variety of data types for the MAP “key” field. Typically, you use BIGINT or STRING to use numeric or character-based keys without worrying about exceeding any size or length constraints.

```
CREATE TABLE map_demo_obscure
(
  id BIGINT,
  m1 MAP <INT, INT>,
  m2 MAP <SMALLINT, INT>,
  m3 MAP <TINYINT, INT>,
  m4 MAP <TIMESTAMP, INT>,
  m5 MAP <BOOLEAN, INT>,
  m6 MAP <CHAR(5), INT>,
```

```
  m7 MAP <VARCHAR(25), INT>,
  m8 MAP <FLOAT, INT>,
  m9 MAP <DOUBLE, INT>,
  m10 MAP <DECIMAL(12,2), INT>
)
STORED AS PARQUET;
```

```
CREATE TABLE celebrities (name STRING, birth_year MAP < STRING, SMALLINT >)
STORED AS PARQUET;
-- A typical row might represent values with 2 different birth years, such
 as:
-- ("Joe Movie Star", { "real": 1972, "claimed": 1977 })

CREATE TABLE countries (name STRING, famous_leaders MAP < INT, STRING >) STO
RED AS PARQUET;
-- A typical row might represent values with different leaders, with key val
ues corresponding to their numeric sequence, such as:
-- ("United States", { 1: "George Washington", 3: "Thomas Jefferson", 16:
"Abraham Lincoln" })

CREATE TABLE airlines (name STRING, special_meals MAP < STRING, MAP < STRI
NG, STRING > >) STORED AS PARQUET;
-- A typical row might represent values with multiple kinds of meals, each
 with several components:
-- ("Elegant Airlines",
--   {
--     "vegetarian": { "breakfast": "pancakes", "snack": "cookies", "dinne
r": "rice pilaf" },
--     "gluten free": { "breakfast": "oatmeal", "snack": "fruit", "dinner":
"chicken" }
--   } )
```

REAL data type

An alias for the DOUBLE data type.

Examples:

These examples show how you can use the type names REAL and DOUBLE interchangeably, and behind the scenes Impala treats them always as DOUBLE.

```
[localhost:21000] > create table r1 (x real);
[localhost:21000] > describe r1;
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | double |         |
+------+--------+---------+
[localhost:21000] > insert into r1 values (1.5), (cast (2.2 as double));
[localhost:21000] > select cast (1e6 as real);
+---------------------------+
| cast(1000000.0 as double) |
+---------------------------+
| 1000000                   |
+---------------------------+
```

Related Information

DOUBLE data type

SMALLINT data type

A 2-byte integer data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name SMALLINT
```

Range: -32768 .. 32767. There is no UNSIGNED subtype.

Conversions: Impala automatically converts to a larger integer type (INT or BIGINT) or a floating-point type (FLOAT or DOUBLE) automatically. Use CAST() to convert to TINYINT, STRING, or TIMESTAMP. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting ##use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Usage notes:

For a convenient and automated way to check the bounds of the SMALLINT type, call the functions MIN_SMALLINT() and MAX_SMALLINT().

If an integer value is too large to be represented as a SMALLINT, use an INT instead.

NULL considerations: Casting any non-numeric value to this type produces a NULL value.

Examples:

```
CREATE TABLE t1 (x SMALLINT);
SELECT CAST(1000 AS SMALLINT);
```

Parquet considerations:

Physically, Parquet files represent TINYINT and SMALLINT values as 32-bit integers. Although Impala rejects attempts to insert out-of-range values into such columns, if you create a new table with the CREATE TABLE ... LIKE PARQUET syntax, any TINYINT or SMALLINT columns in the original table turn into INT columns in the new table.

Partitioning: Prefer to use this type for a partition key column. Impala can process the numeric type more efficiently than a STRING representation of the value.

HBase considerations: This data type is fully compatible with HBase tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 2-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

STRING data type

A data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE and ALTER TABLE statements:

```
column_name STRING
```

Length:

If you need to manipulate string values with precise or maximum lengths, in Impala 2.0 and higher you can declare columns as VARCHAR(*max_length*) or CHAR(*length*), but for best performance use STRING where practical.

Take the following considerations for STRING lengths:

- The hard limit on the size of a STRING and the total size of a row is 2 GB.

  If a query tries to process or create a string larger than this limit, it will return an error to the user.
- The limit is 1 GB on STRING when writing to Parquet files.
- Queries operating on strings with 32 KB or less will work reliably and will not hit significant performance or memory problems (unless you have very complex queries, very many columns, etc.)
- Performance and memory consumption may degrade with strings larger than 32 KB.
- The row size, i.e. the total size of all string and other columns, is subject to lower limits at various points in query execution that support spill-to-disk. A few examples for lower row size limits are:
  - Rows coming from the right side of any hash join
  - Rows coming from either side of a hash join that spills to disk
  - Rows being sorted by the SORT operator without a limit
  - Rows in a grouping aggregation

  The max row size is configurable on a per-query basis with the MAX_ROW_SIZE query option. Rows up to MAX_ROW_SIZE (which defaults to 512 KB) can always be processed in the above cases. Rows larger than MAX_ROW_SIZE are processed on a best-effort basis.

Character sets:

For full support in all Impala subsystems, restrict string values to the ASCII character set. Although some UTF-8 character data can be stored in Impala and retrieved through queries, UTF-8 strings containing non-ASCII characters are not guaranteed to work properly in combination with many SQL aspects, including but not limited to:

- String manipulation functions.
- Comparison operators.
- The ORDER BY clause.
- Values in partition key columns.

For any national language aspects such as collation order or interpreting extended ASCII variants such as ISO-8859-1 or ISO-8859-2 encodings, Impala does not include such metadata with the table definition. If you need to sort, manipulate, or display data depending on those national language characteristics of string data, use logic on the application side.

Conversions:

- Impala does not automatically convert STRING to any numeric type. Impala does automatically convert STRING to TIMESTAMP if the value matches one of the accepted TIMESTAMP formats.
- You can use CAST() to convert STRING values to TINYINT, SMALLINT, INT, BIGINT, FLOAT, DOUBLE, or TIMESTAMP.
- You cannot directly cast a STRING value to BOOLEAN. You can use a CASE expression to evaluate string values such as 'T', 'true', and so on and return Boolean true and false values as appropriate.
- You can cast a BOOLEAN value to STRING, returning '1' for true values and '0' for false values.

Partitioning:

Although it might be convenient to use STRING columns for partition keys, even when those columns contain numbers, for performance and scalability it is much better to use numeric columns as partition keys whenever practical. Although the underlying HDFS directory name might be the same in either case, the in-memory storage for the partition key columns is more compact, and computations are faster, if partition key columns such as YEAR, MONTH, DAY and so on are declared as INT, SMALLINT, and so on.

Zero-length strings: For purposes of clauses such as DISTINCT and GROUP BY, Impala considers zero-length strings (""), NULL, and space to all be different values.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Avro considerations: The Avro specification allows string values up to 2**64 bytes in length. Impala queries for Avro tables use 32-bit integers to hold string lengths. In Impala 2.5 and higher, Impala truncates CHAR and VARCHAR values in Avro tables to (2**31)-1 bytes. If a query encounters a STRING value longer than (2**31)-1 bytes in an Avro table, the query fails. In earlier releases, encountering such long values in an Avro table could cause a crash.

Column statistics considerations: Because the values of this type have variable size, none of the column statistics fields are filled in until you run the COMPUTE STATS statement.

Examples:

The following examples demonstrate double-quoted and single-quoted string literals, and required escaping for quotation marks within string literals:

```
SELECT 'I am a single-quoted string';
SELECT "I am a double-quoted string";
SELECT 'I\'m a single-quoted string with an apostrophe';
SELECT "I\'m a double-quoted string with an apostrophe";
SELECT 'I am a "short" single-quoted string containing quotes';
SELECT "I am a \"short\" double-quoted string containing quotes";
```

The following examples demonstrate calls to string manipulation functions to concatenate strings, convert numbers to strings, or pull out substrings:

```
SELECT CONCAT("Once upon a time, there were ", CAST(3 AS STRING), ' little p
igs.');
SELECT SUBSTR("hello world",7,5);
```

The following examples show how to perform operations on STRING columns within a table:

```
CREATE TABLE t1 (s1 STRING, s2 STRING);
INSERT INTO t1 VALUES ("hello", 'world'), (CAST(7 AS STRING), "wonders");
SELECT s1, s2, length(s1) FROM t1 WHERE s2 LIKE 'w%';
```

Related Information

MAX_ROW_SIZE query option

TIMESTAMP data type

STRUCT complex type

A complex data type, representing multiple fields of a single item. Frequently used as the element type of an ARRAY or the VALUE part of a MAP.

Syntax:

```
column_name STRUCT < name : type [COMMENT 'comment_string'], ... >

type ::= primitive_type | complex_type
```

The names and number of fields within the STRUCT are fixed. Each field can be a different type. A field within a STRUCT can also be another STRUCT, or an ARRAY or a MAP, allowing you to create nested data structures with a maximum nesting depth of 100.

A STRUCT can be the top-level type for a column, or can itself be an item within an ARRAY or the value part of the key-value pair in a MAP.

When a STRUCT is used as an ARRAY element or a MAP value, you use a join clause to bring the ARRAY or MAP elements into the result set, and then refer to *array_name*.ITEM.*field* or *map_name*.VALUE.*field*. In the case of a STRUCT directly inside an ARRAY or MAP, you can omit the .ITEM and .VALUE pseudocolumns and refer directly to *array_name*.*field* or *map_name*.*field*.

Usage notes:

Because complex types are often used in combination, for example an ARRAY of STRUCT elements, if you are unfamiliar with the Impala complex types, start with Complex types on page 49 for background information and usage examples.

A STRUCT is similar conceptually to a table row: it contains a fixed number of named fields, each with a predefined type. To combine two related tables, while using complex types to minimize repetition, the typical way to represent that data is as an ARRAY of STRUCT elements.

Because a STRUCT has a fixed number of named fields, it typically does not make sense to have a STRUCT as the type of a table column. In such a case, you could just make each field of the STRUCT into a separate column of the table. The STRUCT type is most useful as an item of an ARRAY or the value part of the key-value pair in a MAP. A nested type column with a STRUCT at the lowest level lets you associate a variable number of row-like objects with each row of the table.

The STRUCT type is straightforward to reference within a query. You do not need to include the STRUCT column in a join clause or give it a table alias, as is required for the ARRAY and MAP types. You refer to the individual fields using dot notation, such as *struct_column_name.field_name*, without any pseudocolumn such as ITEM or VALUE.

You can pass a multi-part qualified name to DESCRIBE to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table T1 contains an ARRAY column A1, you could issue the statement DESCRIBE t1.a1. If table T1 contained a STRUCT column S1, and a field F1 within the STRUCT was a MAP, you could issue the statement DESCRIBE t1.s1.f1. An ARRAY is shown as a two-column table, with ITEM and POS columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

Internal details:

Within the Parquet data file, the values for each STRUCT field are stored adjacent to each other, so that they can be encoded and compressed using all the Parquet techniques for storing sets of similar or repeated values. The adjacency applies even when the STRUCT values are part of an ARRAY or MAP. During a query, Impala avoids unnecessary I/O by reading only the portions of the Parquet data file containing the requested STRUCT fields.

Added in: Impala 2.3.0

Restrictions:

- Columns with this data type can only be used in tables or partitions with the Parquet file format.
- Columns with this data type cannot be used as partition key columns in a partitioned table.
- The COMPUTE STATS statement does not produce any statistics for columns of this data type.
- The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.
- See the *Limitations and restrictions for complex types* topic for a full list of limitations and associated guidelines about complex type columns.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Examples:

Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types for the table definitions.

The following example shows a table with various kinds of STRUCT columns, both at the top level and nested within other complex types. Practice the CREATE TABLE and query notation for complex type columns using empty tables, until you can visualize a complex data structure and construct corresponding SQL statements reliably.

```
CREATE TABLE struct_demo
(
  id BIGINT,
  name STRING,
```

```
-- A STRUCT as a top-level column. Demonstrates how the table ID column
-- and the ID field within the STRUCT can coexist without a name conflict.
  employee_info STRUCT < employer: STRING, id: BIGINT, address: STRING >,

-- A STRUCT as the element type of an ARRAY.
  places_lived ARRAY < STRUCT <street: STRING, city: STRING, country: STR
ING >>,
-- A STRUCT as the value portion of the key-value pairs in a MAP.
  memorable_moments MAP < STRING, STRUCT < year: INT, place: STRING, deta
ils: STRING >>,
-- A STRUCT where one of the fields is another STRUCT.
  current_address STRUCT < street_address: STRUCT <street_number: INT, stree
t_name: STRING, street_type: STRING>, country: STRING, postal_code: STRING >
)
STORED AS PARQUET;
```

The following example shows how to examine the structure of a table containing one or more STRUCT columns by using the DESCRIBE statement. You can visualize each STRUCT as its own table, with columns named the same as each field of the STRUCT. If the STRUCT is nested inside another complex type, such as ARRAY, you can extend the qualified name passed to DESCRIBE until the output shows just the STRUCT fields.

```
DESCRIBE struct_demo;
+-------------------+--------------------------+
| name              | type                     |
+-------------------+--------------------------+
| id                | bigint                   |
| name              | string                   |
| employee_info     | struct<                  |
|                   |   employer:string,       |
|                   |   id:bigint,             |
|                   |   address:string         |
|                   | >                        |
| places_lived      | array<struct<            |
|                   |   street:string,         |
|                   |   city:string,           |
|                   |   country:string         |
|                   | >>                       |
| memorable_moments | map<string,struct<       |
|                   |   year:int,              |
|                   |   place:string,          |
|                   |   details:string         |
|                   | >>                       |
| current_address   | struct<                  |
|                   |   street_address:struct< |
|                   |     street_number:int,   |
|                   |     street_name:string,  |
|                   |     street_type:string   |
|                   |   >,                     |
|                   |   country:string,        |
|                   |   postal_code:string     |
|                   | >                        |
+-------------------+--------------------------+
```

The top-level column EMPLOYEE_INFO is a STRUCT. Describing *table_name.struct_name* displays the fields of the STRUCT as if they were columns of a table:

```
DESCRIBE struct_demo.employee_info;
+----------+--------+
| name     | type   |
+----------+--------+
```

```
| employer | string |
| id       | bigint |
| address  | string |
+----------+--------+
```

Because PLACES_LIVED is a STRUCT inside an ARRAY, the initial DESCRIBE shows the structure of the ARRAY:

```
DESCRIBE struct_demo.places_lived;
+------+------------------+
| name | type             |
+------+------------------+
| item | struct<          |
|      |   street:string, |
|      |   city:string,   |
|      |   country:string |
|      | >                |
| pos  | bigint           |
+------+------------------+
```

Ask for the details of the ITEM field of the ARRAY to see just the layout of the STRUCT:

```
DESCRIBE struct_demo.places_lived.item;
+---------+--------+
| name    | type   |
+---------+--------+
| street  | string |
| city    | string |
| country | string |
+---------+--------+
```

Likewise, MEMORABLE_MOMENTS has a STRUCT inside a MAP, which requires an extra level of qualified name to see just the STRUCT part:

```
DESCRIBE struct_demo.memorable_moments;
+-------+------------------+
| name  | type             |
+-------+------------------+
| key   | string           |
| value | struct<          |
|       |   year:int,      |
|       |   place:string,  |
|       |   details:string |
|       | >                |
+-------+------------------+
```

For a MAP, ask to see the VALUE field to see the corresponding STRUCT fields in a table-like structure:

```
DESCRIBE struct_demo.memorable_moments.value;
+---------+--------+
| name    | type   |
+---------+--------+
| year    | int    |
| place   | string |
| details | string |
+---------+--------+
```

For a STRUCT inside a STRUCT, we can see the fields of the outer STRUCT:

```
DESCRIBE struct_demo.current_address;
+---------------+----------------------+
```

```
| name           | type                    |
+----------------+-------------------------+
| street_address | struct<                 |
|                |   street_number:int,    |
|                |   street_name:string,   |
|                |   street_type:string    |
|                | >                       |
| country        | string                  |
| postal_code    | string                  |
+----------------+-------------------------+
```

Then we can use a further qualified name to see just the fields of the inner STRUCT:

```
DESCRIBE struct_demo.current_address.street_address;
+---------------+--------+
| name          | type   |
+---------------+--------+
| street_number | int    |
| street_name   | string |
| street_type   | string |
+---------------+--------+
```

The following example shows how to examine the structure of a table containing one or more STRUCT columns by using the DESCRIBE statement. You can visualize each STRUCT as its own table, with columns named the same as each field of the STRUCT. If the STRUCT is nested inside another complex type, such as ARRAY, you can extend the qualified name passed to DESCRIBE until the output shows just the STRUCT fields.

```
DESCRIBE struct_demo;
+-------------------+--------------------------+---------+
| name              | type                     | comment |
+-------------------+--------------------------+---------+
| id                | bigint                   |         |
| name              | string                   |         |
| employee_info     | struct<                  |         |
|                   |   employer:string,       |         |
|                   |   id:bigint,             |         |
|                   |   address:string         |         |
|                   | >                        |         |
| places_lived      | array<struct<            |         |
|                   |   street:string,         |         |
|                   |   city:string,           |         |
|                   |   country:string         |         |
|                   | >>                       |         |
| memorable_moments | map<string,struct<       |         |
|                   |   year:int,              |         |
|                   |   place:string,          |         |
|                   |   details:string         |         |
|                   | >>                       |         |
| current_address   | struct<                  |         |
|                   |   street_address:struct< |         |
|                   |     street_number:int,   |         |
|                   |     street_name:string,  |         |
|                   |     street_type:string   |         |
|                   |   >,                     |         |
|                   |   country:string,        |         |
|                   |   postal_code:string     |         |
|                   | >                        |         |
+-------------------+--------------------------+---------+
SELECT id, employee_info.id FROM struct_demo;

SELECT id, employee_info.id AS employee_id FROM struct_demo;
```

```
SELECT id, employee_info.id AS employee_id, employee_info.employer
  FROM struct_demo;

SELECT id, name, street, city, country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, places_lived.pos, places_lived.street, places_lived.city, p
laces_lived.country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, pl.pos, pl.street, pl.city, pl.country
  FROM struct_demo, struct_demo.places_lived AS pl;

SELECT id, name, places_lived.pos, places_lived.street, places_lived.city, p
laces_lived.country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, pos, street, city, country
  FROM struct_demo, struct_demo.places_lived;

SELECT id, name, memorable_moments.key,
  memorable_moments.value.year,
  memorable_moments.value.place,
  memorable_moments.value.details
FROM struct_demo, struct_demo.memorable_moments
WHERE memorable_moments.key IN ('Birthday','Anniversary','Graduation');

SELECT id, name, mm.key, mm.value.year, mm.value.place, mm.value.details
  FROM struct_demo, struct_demo.memorable_moments AS mm
WHERE mm.key IN ('Birthday','Anniversary','Graduation');
SELECT id, name, memorable_moments.key, memorable_moments.value.year,
  memorable_moments.value.place, memorable_moments.value.details
FROM struct_demo, struct_demo.memorable_moments
WHERE key IN ('Birthday','Anniversary','Graduation');

SELECT id, name, key, value.year, value.place, value.details
  FROM struct_demo, struct_demo.memorable_moments
WHERE key IN ('Birthday','Anniversary','Graduation');
SELECT id, name, key, year, place, details
  FROM struct_demo, struct_demo.memorable_moments
WHERE key IN ('Birthday','Anniversary','Graduation');
SELECT id, name,
  current_address.street_address.street_number,
  current_address.street_address.street_name,
  current_address.street_address.street_type,
  current_address.country,
  current_address.postal_code
FROM struct_demo;
```

For example, this table uses a struct that encodes several data values for each phone number associated with a person. Each person can have a variable-length array of associated phone numbers, and queries can refer to the category field to locate specific home, work, mobile, and so on kinds of phone numbers.

```
CREATE TABLE contact_info_many_structs
(
  id BIGINT, name STRING,
  phone_numbers ARRAY < STRUCT <category:STRING, country_code:STRING, are
a_code:SMALLINT, full_number:STRING, mobile:BOOLEAN, carrier:STRING > >
) STORED AS PARQUET;
```

Because structs are naturally suited to composite values where the fields have different data types, you might use them to decompose things such as addresses:

```
CREATE TABLE contact_info_detailed_address
(
  id BIGINT, name STRING,
  address STRUCT < house_number:INT, street:STRING, street_type:STRING, a
partment:STRING, city:STRING, region:STRING, country:STRING >
);
```

In a big data context, splitting out data fields such as the number part of the address and the street name could let you do analysis on each field independently. For example, which streets have the largest number range of addresses, what are the statistical properties of the street names, which areas have a higher proportion of “Roads”, “Courts” or “Boulevards”, and so on.

TIMESTAMP data type

In Impala, the TIMESTAMP data type holds a value of date and time. It can be decomposed into year, month, day, hour, minute and seconds fields, but with no time zone information available, it does not correspond to any specific point in time.

Internally, the resolution of the time portion of a TIMESTAMP value is in nanoseconds.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name TIMESTAMP
```

```
timestamp [+ | -] INTERVAL interval
DATE_ADD (timestamp, INTERVAL interval time_unit)
```

Range: 1400-01-01 to 9999-12-31

Out of range TIMESTAMP values are converted to NULL.

The range of Impala TIMESTAMP is different from the Hive TIMESTAMP type.

INTERVAL expressions:

You can perform date arithmetic by adding or subtracting a specified number of time units, using the INTERVAL keyword and the + operator, the - operator, date_add() or date_sub().

The following units are supported for time_unit in the INTERVAL clause:

- YEAR[S]
- MONTH[S]
- WEEK[S]
- DAY[S]
- HOUR[S]
- MINUTE[S]
- SECOND[S]
- MILLISECOND[S]
- MICROSECOND[S]
- NANOSECOND[S]

You can only specify one time unit in each interval expression, for example INTERVAL 3 DAYS or INTERVAL 25 HOURS, but you can produce any granularity by adding together successive INTERVAL values, such as *timestamp_value* + INTERVAL 3 WEEKS - INTERVAL 1 DAY + INTERVAL 10 MICROSECONDS.

Internal details: Represented in memory as a 16-byte value.

Time zones:

By default, Impala stores and interprets TIMESTAMP values in UTC time zone when writing to data files, reading from data files, or converting to and from system time values through functions.

When you set the --use_local_tz_for_unix_timestamp_conversions startup flag to TRUE, Impala treats the TIME STAMP values specified in the local time zone. The local time zone is determined in the following order with the TIMESTAMP query option takes the highest precedence:

1. The TIMESTAMP query option
2. $TZ environment variable
3. System time zone where the impalad coordinator runs

The --use_local_tz_for_unix_timestamp_conversions setting can be used to fix discrepancy in INTERVAL operations. For example, a TIMESTAMP + INTERVAL *n-hours* can be affected by Daylight Saving Time, which Impala does not consider by default as these operations are applied as if the timestamp was in UTC. You can use the --use_local_tz_for_unix_timestamp_conversions setting to fix the issue.

See Customizing time zones on page 45 for configuring to use custom time zone database and aliases.

See Impala date and time functions for the list of functions affected by the --use_local_tz_for_unix_timestamp_co nversions setting.

Time zone handling between Impala and Hive:

Interoperability between Hive and Impala is different depending on the file format.

- Text

  For text tables, TIMESTAMP values can be written and read interchangeably by Impala and Hive as Hive reads and writes TIMESTAMP values without converting with respect to time zones.
- Parquet

  Note: This section only applies to INT96 TIMESTAMP.

  When Hive writes to Parquet data files, the TIMESTAMP values are normalized to UTC from the local time zone of the host where the data was written. On the other hand, Impala does not make any time zone adjustment when it writes or reads INT96 TIMESTAMP values to Parquet files. This difference in time zone handling can cause potentially inconsistent results when Impala processes TIMESTAMP values in the Parquet files written by Hive.

  To avoid incompatibility problems or having to code workarounds, you can specify one or both of these impalad startup flags:

  - --use_local_tz_for_unix_timestamp_conversions=true
  - --convert_legacy_hive_parquet_utc_timestamps=true

  When the ##convert_legacy_hive_parquet_utc_timestamps setting is enabled, Impala recognizes the Parquet data files written by Hive, and applies the same UTC-to-local-timezone conversion logic during the query as Hive does.

  In Impala 3.0 and lower, the ##convert_legacy_hive_parquet_utc_timestamps setting had a severe impact on multi-threaded performance. The new time zone implementation in Impala 3.1 eliminated most of the performance overhead and made Impala scale well to multiple threads. The ##convert_legacy_hive_parquet_utc_timestamps setting is turned off by default for a performance reason. To avoid unexpected incompatibility problems, you should turn on the option when processing TIMESTAMP columns in Parquet files written by Hive.

  Hive currently cannot write INT64 TIMESTAMP values.

  In Impala 3.2 and higher, INT64 TIMESTAMP values annotated with the TIMESTAMP_MILLIS or TIME STAMP_MICROS OriginalType are assumed to be always UTC normalized, so the UTC to local conversion will be always done. INT64 TIMESTAMP annotated with the TIMESTAMP LogicalType specifies whether UTC to local conversion is necessary depending on the Parquet metadata.

Conversions:

Impala automatically converts STRING literals of the correct format into TIMESTAMP values. Timestamp values are accepted in the format 'yyyy#MM#dd HH:mm:ss.SSSSSS', and can consist of just the date, or just the time, with or without the fractional second portion. For example, you can specify TIMESTAMP values such as '1966#07#30', '08:30:00', or '1985#09#25 17:45:30.005'.

Leading zeroes are not required in the numbers representing the date component, such as month and date, or the time component, such as hour, minute, and second. For example, Impala accepts both '2018#1#1 01:02:03' and '2018-01-01 1:2:3' as valid.

In STRING to TIMESTAMP conversions, leading and trailing white spaces, such as a space, a tab, a newline, or a carriage return, are ignored. For example, Impala treats the following as equivalent: '1999#12#01 01:02:03 ', ' 1999#12#01 01:02:03', '1999#12#01 01:02:03\r\n\t'.

When you convert or cast a STRING literal to TIMESTAMP, you can use the following separators between the date part and the time part:

- One or more space characters

  Example: CAST('2001-01-09 01:05:01' AS TIMESTAMP)
- The character “T”

  Example: CAST('2001-01-09T01:05:01' AS TIMESTAMP)

Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting ##use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

In Impala 1.3 and higher, the FROM_UNIXTIME() and UNIX_TIMESTAMP() functions allow a wider range of format strings, with more flexibility in element order, repetition of letter placeholders, and separator characters. In Impala 2.3 and higher, the UNIX_TIMESTAMP() function also allows a numeric timezone offset to be specified as part of the input string. See Impala date and time functions on page 311 for details.

In Impala 2.2.0 and higher, built-in functions that accept or return integers representing TIMESTAMP values use the BIGINT type for parameters and return values, rather than INT. This change lets the date and time functions avoid an overflow error that would otherwise occur on January 19th, 2038 (known as the “Year 2038 problem” or “Y2K38 problem”). This change affects the FROM_UNIXTIME() and UNIX_TIMESTAMP() functions. You might need to change application code that interacts with these functions, change the types of columns that store the return values, or add CAST() calls to SQL statements that call these functions.

Partitioning:

Although you cannot use a TIMESTAMP column as a partition key, you can extract the individual years, months, days, hours, and so on and partition based on those columns. Because the partition key column values are represented in HDFS directory names, rather than as fields in the data files themselves, you can also keep the original TIMESTAMP values if desired, without duplicating data or wasting storage space. See *Partition Key Columns* for more details on partitioning with date and time values.

```
[localhost:21000] > create table timeline (event string) partitioned by (happened timestamp);
ERROR: AnalysisException: Type 'TIMESTAMP' is not supported as partition-column type in column: happened
```

NULL considerations: Casting any unrecognized STRING value to this type produces a NULL value.

HBase considerations: This data type is fully compatible with HBase tables.

Parquet consideration: INT96 and INT64 encoded Parquet timestamps are supported in Impala.

Parquet considerations: This type is fully compatible with Parquet tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

Kudu considerations:

In Impala 2.9 and higher, you can include TIMESTAMP columns in Kudu tables, instead of representing the date and time as a BIGINT value. The behavior of TIMESTAMP for Kudu tables has some special considerations:

- Any nanoseconds in the original 96-bit value produced by Impala are not stored, because Kudu represents date/time columns using 64-bit values. The nanosecond portion of the value is rounded, not truncated. Therefore, a TIMESTAMP value that you store in a Kudu table might not be bit-for-bit identical to the value returned by a query.
- The conversion between the Impala 96-bit representation and the Kudu 64-bit representation introduces some performance overhead when reading or writing TIMESTAMP columns. You can minimize the overhead during writes by performing inserts through the Kudu API. Because the overhead during reads applies to each query, you might continue to use a BIGINT column to represent date/time values in performance-critical applications.
- The Impala TIMESTAMP type has a narrower range for years than the underlying Kudu data type. Impala can represent years 1400-9999. If year values outside this range are written to a Kudu table by a non-Impala client, Impala returns NULL by default when reading those TIMESTAMP values during a query. Or, if the ABORT_ON_ERROR query option is enabled, the query fails when it encounters a value with an out-of-range year.

Restrictions:

If you cast a STRING with an unrecognized format to a TIMESTAMP, the result is NULL rather than an error. Make sure to test your data pipeline to be sure any textual date and time values are in a format that Impala TIMESTAMP can recognize.

Currently, Avro tables cannot contain TIMESTAMP columns. If you need to store date and time values in Avro tables, as a workaround you can use a STRING representation of the values, convert the values to BIGINT with the UNIX_TIMESTAMP() function, or create separate numeric columns for individual date and time fields using the EXTRACT() function.

Examples:

The following examples demonstrate using TIMESTAMP values with built-in functions:

```
select cast('1966-07-30' as timestamp);
select cast('1985-09-25 17:45:30.005' as timestamp);
select cast('08:30:00' as timestamp);
select hour('1970-01-01 15:30:00');         -- Succeeds, returns 15.
select hour('1970-01-01 15:30');            -- Returns NULL because seconds
field required.
select hour('1970-01-01 27:30:00');         -- Returns NULL because hour val
ue out of range.
select dayofweek('2004-06-13');             -- Returns 1, representing Sund
ay.
select dayname('2004-06-13');               -- Returns 'Sunday'.
select date_add('2004-06-13', 365);         -- Returns 2005-06-13 with zeros
 for hh:mm:ss fields.
select day('2004-06-13');                   -- Returns 13.
select datediff('1989-12-31','1984-09-01'); -- How many days between these 2
 dates?
select now();                               -- Returns current date and time
 in local timezone.
```

The following examples demonstrate using TIMESTAMP values with HDFS-backed tables:

```
create table dates_and_times (t timestamp);
insert into dates_and_times values
  ('1966-07-30'), ('1985-09-25 17:45:30.005'), ('08:30:00'), (now());
```

The following examples demonstrate using TIMESTAMP values with Kudu tables:

```
create table timestamp_t (x int primary key, s string, t timestamp, b bigint
)
  partition by hash (x) partitions 16
  stored as kudu;

-- The default value of now() has microsecond precision, so the final 3 d
igits
-- representing nanoseconds are all zero.
insert into timestamp_t values (1, cast(now() as string), now(), unix_timest
amp(now()));

-- Values with 1-499 nanoseconds are rounded down in the Kudu TIMESTAMP co
lumn.
insert into timestamp_t values (2, cast(now() + interval 100 nanoseconds as
 string), now() + interval 100 nanoseconds, unix_timestamp(now() + interval
 100 nanoseconds));
insert into timestamp_t values (3, cast(now() + interval 499 nanoseconds as
string), now() + interval 499 nanoseconds, unix_timestamp(now() + interval 4
99 nanoseconds));

-- Values with 500-999 nanoseconds are rounded up in the Kudu TIMESTAMP c
olumn.
insert into timestamp_t values (4, cast(now() + interval 500 nanoseconds as
 string), now() + interval 500 nanoseconds, unix_timestamp(now() + interval
 500 nanoseconds));
insert into timestamp_t values (5, cast(now() + interval 501 nanoseconds as
 string), now() + interval 501 nanoseconds, unix_timestamp(now() + interval
501 nanoseconds));

-- The string representation shows how underlying Impala TIMESTAMP can have
nanosecond precision.
-- The TIMESTAMP column shows how timestamps in a Kudu table are rounded to
 microsecond precision.
-- The BIGINT column represents seconds past the epoch and so if not affect
ed much by nanoseconds.
select s, t, b from timestamp_t order by t;
+-------------------------------+-------------------------------+-----------
-+
| s                             | t                             | b
  |
+-------------------------------+-------------------------------+---------
---+
| 2017-05-31 15:30:05.107157000 | 2017-05-31 15:30:05.107157000 | 1496244605
 |
| 2017-05-31 15:30:28.868151100 | 2017-05-31 15:30:28.868151000 | 1496244628
 |
| 2017-05-31 15:34:33.674692499 | 2017-05-31 15:34:33.674692000 | 1496244
873 |
| 2017-05-31 15:35:04.769166500 | 2017-05-31 15:35:04.769167000 | 1496244904
 |
| 2017-05-31 15:35:33.033082501 | 2017-05-31 15:35:33.033083000 | 1496244933
 |
+-------------------------------+-------------------------------+-----------
-+
```

Added in: Available in all versions of Impala.

Related information:

- To convert to or from different date formats, or perform date arithmetic, use the date and time functions described in Impala date and time functions on page 311. In particular, the from_unixtime() function requires a case-

sensitive format string such as "yyyy-MM-dd HH:mm:ss.SSSS", matching one of the allowed variations of a TIMESTAMP value (date plus time, only date, only time, optional fractional seconds).

- See SQL differences between Impala and Hive on page 437 for details about differences in TIMESTAMP handling between Impala and Hive.

Related Information

Hive data types

DATE data type

Partitioning

Customizing time zones

Impala 3.1 onward, you can customize the time zone definitions used in Impala.

- By default, Impala uses the OS's time zone database located in /usr/share/zoneinfo. This directory contains the IANA timezone database in a compiled binary format. The contents of the zoneinfo directory is controlled by the OS's package manager.
- Use the following start-up flags managed as impalad safety valves in Cloudera Manager.
  - ##hdfs_zone_info_zip: This flag allows Impala administrators to specify a custom timezone database. The flag should be set to a shared (not necessarily HDFS) path that points to a zip archive of a custom IANA timezone database. The timezone database is expected to be in a compiled binary format. If the startup flag is set, Impala will use the specified timezone database instead of the default /usr/share/zoneinfo database. The timezone db upgrade process is described in detail below.
  - ##hdfs_zone_alias_conf: This flag allows Impala administrators to specify definitions for custom timezone aliases. The flag should be set to a shared (not necessarily HDFS) path that specifies a config file containing custom timezone alias definitions. This config file can be used as a workaround for users who want to keep using their legacy timezone names. Configuring custom aliases is described in detail below.

Upgrading custom IANA time zone database:

1. Download latest IANA time zone database distribution:

   ```
   git clone https://github.com/eggert/tz
   ```

   Alternatively, download a specific tzdb version from:

   ```
    https://www.iana.org/time-zones/repository
   ```

2. Build timezone tools:

   ```
   cd tz
   make TOPDIR=tzdata install
   ```

3. Generate the compiled binary time zone database:

   ```
   ./zic -d ./tzdata/etc/zoneinfo africa antarctica asia australasia backward
    backzone etcetera europe factory northamerica pacificnew southamerica sys
   temv
   ```

4. Create zip archive:

   ```
   pushd ./tzdata/etc
   zip -r zoneinfo.zip zoneinfo
   popd
   ```

5. Copy the time zone database to HDFS:

   ```
   hdfs dfs -mkdir -p /tzdb/latest
   hdfs dfs -copyFromLocal ./tzdata/etc/zoneinfo.zip /tzdb/latest
   ```

6. Set the ##hdfs_zone_info_zip startup flag to /tzdb/latest/zoneinfo.zip as an impalad safety valve.

7. Perform a full restart of Impala service.

Configuring custom time zone aliases:

1. Create a tzalias.conf config file that contains time zone alias definitions formatted as ALIAS = DEFINITION. For example:

```
#
# Define aliases for existing timezone names:
#
Universal Coordinated Time = UTC
Mideast/Riyadh89 = Asia/Riyadh
PDT = America/Los_Angeles
#
# Define aliases as UTC offsets in seconds:
#
GMT-01:00 = 3600
GMT+01:00 = -3600
```

2. Copy the config file to HDFS:

```
hdfs dfs -mkdir -p /tzdb
hdfs dfs -copyFromLocal tzalias.conf /tzdb
```

3. Set the ##hdfs_zone_alias_conf startup flag to /tzdb/tzalias.conf as an impalad safety valve.
4. Perform a full restart of Impala service.

Added in: Impala 3.1.

TINYINT data type

A 1-byte integer data type used in CREATE TABLE and ALTER TABLE statements.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name TINYINT
```

Range: -128 .. 127. There is no UNSIGNED subtype.

Conversions: Impala automatically converts to a larger integer type (SMALLINT, INT, or BIGINT) or a floating-point type (FLOAT or DOUBLE) automatically. Use CAST() to convert to STRING or TIMESTAMP. Casting an integer or floating-point value N to TIMESTAMP produces a value that is N seconds past the start of the epoch date (January 1, 1970). By default, the result value represents a date and time in the UTC time zone. If the setting ##use_local_tz_for_unix_timestamp_conversions=true is in effect, the resulting TIMESTAMP represents a date and time in the local time zone.

Impala does not return column overflows as NULL, so that customers can distinguish between NULL data and overflow conditions similar to how they do so with traditional database systems. Impala returns the largest or smallest value in the range for the type. For example, valid values for a tinyint range from -128 to 127. In Impala, a tinyint with a value of -200 returns -128 rather than NULL. A tinyint with a value of 200 returns 127.

Usage notes:

For a convenient and automated way to check the bounds of the TINYINT type, call the functions MIN_TINYINT() and MAX_TINYINT().

If an integer value is too large to be represented as a TINYINT, use a SMALLINT instead.

NULL considerations: Casting any non-numeric value to this type produces a NULL value.

Examples:

```
CREATE TABLE t1 (x TINYINT);
```

```
SELECT CAST(100 AS TINYINT);
```

Parquet considerations:

Physically, Parquet files represent TINYINT and SMALLINT values as 32-bit integers. Although Impala rejects attempts to insert out-of-range values into such columns, if you create a new table with the CREATE TABLE ... LIK E PARQUET syntax, any TINYINT or SMALLINT columns in the original table turn into INT columns in the new table.

HBase considerations: This data type is fully compatible with HBase tables.

Text table considerations: Values of this type are potentially larger in text tables than in tables using Parquet or other binary formats.

Internal details: Represented in memory as a 1-byte value.

Added in: Available in all versions of Impala.

Column statistics considerations: Because this type has a fixed size, the maximum and average size fields are always filled in for column statistics, even before you run the COMPUTE STATS statement.

VARCHAR data type

A variable-length character type, truncated during processing if necessary to fit within the specified length.

Syntax:

In the column definition of a CREATE TABLE statement:

```
column_name VARCHAR(max_length)
```

The maximum length you can specify is 65,535.

Partitioning: This type can be used for partition key columns. Because of the efficiency advantage of numeric values over character-based values, if the partition key is a string representation of a number, prefer to use an integer type with sufficient range (INT, BIGINT, and so on) where practical.

HBase considerations: This data type cannot be used with HBase tables.

Parquet considerations:

- This type can be read from and written to Parquet files.
- There is no requirement for a particular level of Parquet.
- Parquet files generated by Impala and containing this type can be freely interchanged with other components such as Hive and MapReduce.
- Parquet data files can contain values that are longer than allowed by the VARCHAR(*n*) length limit. Impala ignores any extra trailing characters when it processes those values during a query.

Text table considerations:

Text data files can contain values that are longer than allowed by the VARCHAR(*n*) length limit. Any extra trailing characters are ignored when Impala processes those values during a query.

Avro considerations: The Avro specification allows string values up to 2**64 bytes in length. Impala queries for Avro tables use 32-bit integers to hold string lengths. In Impala 2.5 and higher, Impala truncates CHAR and VARCHAR values in Avro tables to (2**31)-1 bytes. If a query encounters a STRING value longer than (2**31)-1 bytes in an Avro table, the query fails. In earlier releases, encountering such long values in an Avro table could cause a crash.

Schema evolution considerations:

You can use ALTER TABLE ... CHANGE to switch column data types to and from VARCHAR. You can convert from STRING to VARCHAR(*n*), or from VARCHAR(*n*) to STRING, or from CHAR(*n*) to VARCHAR(*n*), or from VARCHAR(*n*) to CHAR(*n*). When switching back and forth between VARCHAR and CHAR, you can also change the length value. This schema evolution works the same for tables using any file format. If a table contains values

longer than the maximum length defined for a VARCHAR column, Impala does not return an error. Any extra trailing characters are ignored when Impala processes those values during a query.

Compatibility:

This type is available in Impala version 2.0 or higher.

Internal details: Represented in memory as a byte array with the minimum size needed to represent each value.

Column statistics considerations: Because the values of this type have variable size, none of the column statistics fields are filled in until you run the COMPUTE STATS statement.

Kudu considerations:

Currently, the data types CHAR, VARCHAR, ARRAY, MAP, and STRUCT cannot be used with Kudu tables.

Restrictions:

All data in CHAR and VARCHAR columns must be in a character encoding that is compatible with UTF-8. If you have binary data from another database system (that is, a BLOB type), use a STRING column to hold it.

Examples:

The following examples show how long and short VARCHAR values are treated. Values longer than the maximum specified length are truncated by CAST(), or when queried from existing data files. Values shorter than the maximum specified length are represented as the actual length of the value, with no extra padding as seen with CHAR values.

```
create table varchar_1 (s varchar(1));
create table varchar_4 (s varchar(4));
create table varchar_20 (s varchar(20));

insert into varchar_1 values (cast('a' as varchar(1))), (cast('b' as varchar
(1))), (cast('hello' as varchar(1))), (cast('world' as varchar(1)));
insert into varchar_4 values (cast('a' as varchar(4))), (cast('b' as varcha
r(4))), (cast('hello' as varchar(4))), (cast('world' as varchar(4)));
insert into varchar_20 values (cast('a' as varchar(20))), (cast('b' as var
char(20))), (cast('hello' as varchar(20))), (cast('world' as varchar(20)));
select * from varchar_1;
+---+
| s |
+---+
| a |
| b |
| h |
| w |
+---+
select * from varchar_4;
+------+
| s    |
+------+
| a    |
| b    |
| hell |
| worl |
+------+
[localhost:21000] > select * from varchar_20;
+-------+
| s     |
+-------+
| a     |
| b     |
| hello |
| world |
+-------+
select concat('[',s,']') as s from varchar_20;
+---------+
```

```
| s       |
+---------+
| [a]     |
| [b]     |
| [hello] |
| [world] |
+---------+
```

The following example shows how identical VARCHAR values compare as equal, even if the columns are defined with different maximum lengths. Both tables contain 'a' and 'b' values. The longer 'hello' and 'world' values from the VARCHAR_20 table were truncated when inserted into the VARCHAR_1 table.

```
select s from varchar_1 join varchar_20 using (s);
+-------+
| s     |
+-------+
| a     |
| b     |
+-------+
```

The following examples show how VARCHAR values are freely interchangeable with STRING values in contexts such as comparison operators and built-in functions:

```
select length(cast('foo' as varchar(100))) as length;
+--------+
| length |
+--------+
| 3      |
+--------+
select cast('xyz' as varchar(5)) > cast('abc' as varchar(10)) as greater;
+---------+
| greater |
+---------+
| true    |
+---------+
```

UDF considerations: This type cannot be used for the argument or return type of a user-defined function (UDF) or user-defined aggregate function (UDA).

Complex types

Complex types (also referred to as *nested types*) let you represent multiple data values within a single row/column position. They differ from the familiar column types such as BIGINT and STRING, known as *scalar types* or *primitive types*, which represent a single data value within a given row/column position. Impala supports the complex types ARRAY, MAP, and STRUCT in Impala 2.3 and higher. The Hive UNION type is not currently supported.

Once you understand the basics of complex types, refer to the individual type topics when you need to refresh your memory about syntax and examples:

- ARRAY complex type on page 11
- STRUCT complex type on page 34
- MAP complex type on page 27

Benefits of Impala complex types

The reasons for using Impala complex types include the following:

- You already have data produced by Hive or other non-Impala component that uses the complex type column names. You might need to convert the underlying data to Parquet to use it with Impala.

- Your data model originates with a non-SQL programming language or a NoSQL data management system. For example, if you are representing Python data expressed as nested lists, dictionaries, and tuples, those data structures correspond closely to Impala ARRAY, MAP, and STRUCT types.
- Your analytic queries involving multiple tables could benefit from greater locality during join processing. By packing more related data items within each HDFS data block, complex types let join queries avoid the network overhead of the traditional Hadoop shuffle or broadcast join techniques.

The Impala complex type support produces result sets with all scalar values, and the scalar components of complex types can be used with all SQL clauses, such as GROUP BY, ORDER BY, all kinds of joins, subqueries, and inline views. The ability to process complex type data entirely in SQL reduces the need to write application-specific code in Java or other programming languages to deconstruct the underlying data structures.

Overview of Impala complex types

The ARRAY and MAP types are closely related: they represent collections with arbitrary numbers of elements, where each element is the same type. In contrast, STRUCT groups together a fixed number of items into a single element. The parts of a STRUCT element (the *fields*) can be of different types, and each field has a name.

The elements of an ARRAY or MAP, or the fields of a STRUCT, can also be other complex types. You can construct elaborate data structures with up to 100 levels of nesting. For example, you can make an ARRAY whose elements are STRUCTs. Within each STRUCT, you can have some fields that are ARRAY, MAP, or another kind of STRUCT. The Impala documentation uses the terms complex and nested types interchangeably; for simplicity, it primarily uses the term complex types, to encompass all the properties of these types.

When visualizing your data model in familiar SQL terms, you can think of each ARRAY or MAP as a miniature table, and each STRUCT as a row within such a table. By default, the table represented by an ARRAY has two columns, POS to represent ordering of elements, and ITEM representing the value of each element. Likewise, by default, the table represented by a MAP encodes key-value pairs, and therefore has two columns, KEY and VALUE.

The ITEM and VALUE names are only required for the very simplest kinds of ARRAY and MAP columns, ones that hold only scalar values. When the elements within the ARRAY or MAP are of type STRUCT rather than a scalar type, then the result set contains columns with names corresponding to the STRUCT fields rather than ITEM or VALUE.

You write most queries that process complex type columns using familiar join syntax, even though the data for both sides of the join resides in a single table. The join notation brings together the scalar values from a row with the values from the complex type columns for that same row. The final result set contains all scalar values, allowing you to do all the familiar filtering, aggregation, ordering, and so on for the complex data entirely in SQL or using business intelligence tools that issue SQL queries.

Behind the scenes, Impala ensures that the processing for each row is done efficiently on a single host, without the network traffic involved in broadcast or shuffle joins. The most common type of join query for tables with complex type columns is INNER JOIN, which returns results only in those cases where the complex type contains some elements. Therefore, most query examples in this section use either the INNER JOIN clause or the equivalent comma notation.

Note:

Although Impala can query complex types that are present in Parquet files, Impala currently cannot create new Parquet files containing complex types. Therefore, the discussion and examples presume that you are working with existing Parquet data produced through Hive, Spark, or some other source. See Constructing Parquet / ORC files with complex columns using Hive for examples of constructing Parquet data files with complex type columns.

For learning purposes, you can create empty tables with complex type columns and practice query syntax, even if you do not have sample data with the required structure.

Design considerations for complex types

When planning to use Impala complex types, and designing the Impala schema, first learn how this kind of schema differs from traditional table layouts from the relational database and data warehousing fields. Because you might

have already encountered complex types in a Hadoop context while using Hive for ETL, also learn how to write high-performance analytic queries for complex type data using Impala SQL syntax.

How complex types differ from traditional data warehouse schemas

Complex types let you associate arbitrary data structures with a particular row. If you are familiar with schema design for relational database management systems or data warehouses, a schema with complex types has the following differences:

- Logically, related values can now be grouped tightly together in the same table.

  In traditional data warehousing, related values were typically arranged in one of two ways:

  - Split across multiple normalized tables. Foreign key columns specified which rows from each table were associated with each other. This arrangement avoided duplicate data and therefore the data was compact, but join queries could be expensive because the related data had to be retrieved from separate locations. (In the case of distributed Hadoop queries, the joined tables might even be transmitted between different hosts in a cluster.)
  - Flattened into a single denormalized table. Although this layout eliminated some potential performance issues by removing the need for join queries, the table typically became larger because values were repeated. The extra data volume could cause performance issues in other parts of the workflow, such as longer ETL cycles or more expensive full-table scans during queries.

  Complex types represent a middle ground that addresses these performance and volume concerns. By physically locating related data within the same data files, complex types increase locality and reduce the expense of join queries. By associating an arbitrary amount of data with a single row, complex types avoid the need to repeat lengthy values such as strings. Because Impala knows which complex type values are associated with each row, you can save storage by avoiding artificial foreign key values that are only used for joins. The flexibility of the STRU CT, ARRAY, and MAP types lets you model familiar constructs such as fact and dimension tables from a data warehouse, and wide tables representing sparse matrices.

Physical storage for complex types in Parquet

Physically, the scalar and complex columns in each row are located adjacent to each other in the same Parquet data file, ensuring that they are processed on the same host rather than being broadcast across the network when cross-referenced within a query. This co-location simplifies the process of copying, converting, and backing all the columns up at once. Because of the column-oriented layout of Parquet files, you can still query only the scalar columns of a table without imposing the I/O penalty of reading the (possibly large) values of the composite columns.

Within each Parquet data file, the constituent parts of complex type columns are stored in column-oriented format:

- Each field of a STRUCT type is stored like a column, with all the scalar values adjacent to each other and encoded, compressed, and so on using the Parquet space-saving techniques.
- For an ARRAY containing scalar values, all those values (represented by the ITEM pseudocolumn) are stored adjacent to each other.
- For a MAP, the values of the KEY pseudocolumn are stored adjacent to each other. If the VALUE pseudocolumn is a scalar type, its values are also stored adjacent to each other.
- If an ARRAY element, STRUCT field, or MAP VALUE part is another complex type, the column-oriented storage applies to the next level down (or the next level after that, and so on for deeply nested types) where the final elements, fields, or values are of scalar types.

The numbers represented by the POS pseudocolumn of an ARRAY are not physically stored in the data files. They are synthesized at query time based on the order of the ARRAY elements associated with each row.

File format support for Impala complex types

Currently, Impala queries support complex type data in the Parquet and ORC file formats. See *Using Parquet data files* for details about the performance benefits and physical layout of Parquet file format.

Because Impala does not parse the data structures containing nested types for unsupported formats such as text, Avro, SequenceFile, or RCFile, you cannot use data files in these formats with Impala, even if the query does not refer to the nested type columns. Also, if a table using an unsupported format originally contained nested type columns, and then those columns were dropped from the table using ALTER TABLE ... DROP COLUMN, any existing data files in the table still contain the nested type data and Impala queries on that table will generate errors.

The one exception to the preceding rule is COUNT(*) queries on RCFile tables that include complex types. Such queries are allowed in Impala 2.6 and higher.

You can perform DDL operations for tables involving complex types in most file formats other than Parquet or ORC. You cannot create tables in Impala with complex types using text files.

You can have a partitioned table with complex type columns that uses a format other than Parquet or ORC, and use ALTER TABLE to change the file format to Parquet/ORC for individual partitions. When you put Parquet/ORC files into those partitions, Impala can execute queries against that data as long as the query does not involve any of the non-Parquet and non-ORC partitions.

If you use the `parquet-tools` command to examine the structure of a Parquet data file that includes complex types, you see that both ARRAY and MAP are represented as a Bag in Parquet terminology, with all fields marked Optional because Impala allows any column to be nullable.

Impala supports either 2-level and 3-level encoding within each Parquet data file. When constructing Parquet data files outside Impala, use either encoding style but do not mix 2-level and 3-level encoding within the same data file.

Choosing between complex types and normalized tables

Choosing between multiple normalized fact and dimension tables, or a single table containing complex types, is an important design decision.

- If you are coming from a traditional database or data warehousing background, you might be familiar with how to split up data between tables. Your business intelligence tools might already be optimized for dealing with this kind of multi-table scenario through join queries.
- If you are pulling data from Impala into an application written in a programming language that has data structures analogous to the complex types, such as Python or Java, complex types in Impala could simplify data interchange and improve understandability and reliability of your program logic.
- You might already be faced with existing infrastructure or receive high volumes of data that assume one layout or the other. For example, complex types are popular with web-oriented applications, for example to keep information about an online user all in one place for convenient lookup and analysis, or to deal with sparse or constantly evolving data fields.
- If some parts of the data change over time while related data remains constant, using multiple normalized tables lets you replace certain parts of the data without reloading the entire data set. Conversely, if you receive related data all bundled together, such as in JSON files, using complex types can save the overhead of splitting the related items across multiple tables.
- From a performance perspective:
  - In Parquet or ORC tables, Impala can skip columns that are not referenced in a query, avoiding the I/O penalty of reading the embedded data. When complex types are nested within a column, the data is physically divided at a very granular level; for example, a query referring to data nested multiple levels deep in a complex type column does not have to read all the data from that column, only the data for the relevant parts of the column type hierarchy.
  - Complex types avoid the possibility of expensive join queries when data from fact and dimension tables is processed in parallel across multiple hosts. All the information for a row

containing complex types is typically to be in the same data block, and therefore does not need to be transmitted across the network when joining fields that are all part of the same row.

- The tradeoff with complex types is that fewer rows fit in each data block. Whether it is better to have more data blocks with fewer rows, or fewer data blocks with many rows, depends on the distribution of your data and the characteristics of your query workload. If the complex columns are rarely referenced, using them might lower efficiency. If you are seeing low parallelism due to a small volume of data (relatively few data blocks) in each table partition, increasing the row size by including complex columns might produce more data blocks and thus spread the work more evenly across the cluster. See *Scalability considerations* for more on this advanced topic.

Differences between Impala and Hive complex types

Impala can query Parquet and ORC tables containing ARRAY, STRUCT, and MAP columns produced by Hive. There are some differences to be aware of between the Impala SQL and Hive SQL syntax for complex types, primarily for queries.

Impala supports a subset of the syntax that Hive supports for specifying ARRAY, STRUCT, and MAP types in the CREATE TABLE statements.

Because Impala STRUCT columns include user-specified field names, you use the NAMED_ST RUCT() constructor in Hive rather than the STRUCT() constructor when you populate an Impala STRUCT column using a Hive INSERT statement.

The Hive UNION type is not currently supported in Impala.

While Impala usually aims for a high degree of compatibility with Hive SQL query syntax, Impala syntax differs from Hive for queries involving complex types. The differences are intended to provide extra flexibility for queries involving these kinds of tables.

- Impala uses dot notation for referring to element names or elements within complex types, and join notation for cross-referencing scalar columns with the elements of complex types within the same row, rather than the LATERAL VIEW clause and EXPLODE() function of Hive SQL.
- Using join notation lets you use all the kinds of join queries with complex type columns. For example, you can use a LEFT OUTER JOIN, LEFT ANTI JOIN, or LEFT SEMI JOIN query to evaluate different scenarios where the complex columns do or do not contain any elements.
- You can include references to collection types inside subqueries and inline views. For example, you can construct a FROM clause where one of the “tables” is a subquery against a complex type column, or use a subquery against a complex type column as the argument to an IN or EXISTS clause.
- The Impala pseudocolumn POS lets you retrieve the position of elements in an array along with the elements themselves, equivalent to the POSEXPLODE() function of Hive SQL. You do not use index notation to retrieve a single array element in a query; the join query loops through the array elements and you use WHERE clauses to specify which elements to return.
- Join clauses involving complex type columns do not require an ON or USING clause. Impala implicitly applies the join key so that the correct array entries or map elements are associated with the correct row from the table.
- Impala does not currently support the UNION complex type.

Limitations and restrictions for complex types

Complex type columns can only be used in tables or partitions with the Parquet or ORC file format.

Complex type columns cannot be used as partition key columns in a partitioned table.

When you use complex types with the ORDER BY, GROUP BY, HAVING, or WHERE clauses, you cannot refer to the column name by itself. Instead, you refer to the names of the scalar values within the complex type, such as the ITEM, POS, KEY, or VALUE pseudocolumns, or the field names from a STRUCT.

The maximum depth of nesting for complex types is 100 levels.

The maximum length of the column definition for any complex type, including declarations for any nested types, is 4000 characters.

For ideal performance and scalability, use small or medium-sized collections, where all the complex columns contain at most a few hundred megabytes per row. Remember, all the columns of a row are stored in the same HDFS data block, whose size in Parquet files typically ranges from 256 MB to 1 GB.

Including complex type columns in a table introduces some overhead that might make queries that do not reference those columns somewhat slower than Impala queries against tables without any complex type columns. Expect at most a 2x slowdown compared to tables that do not have any complex type columns.

Currently, the COMPUTE STATS statement does not collect any statistics for columns containing complex types. Impala uses heuristics to construct execution plans involving complex type columns.

Currently, Impala built-in functions and user-defined functions cannot accept complex types as parameters or produce them as function return values. (When the complex type values are materialized in an Impala result set, the result set contains the scalar components of the values, such as the POS or ITEM for an ARRAY, the KEY or VALUE for a MAP, or the fields of a STRUCT; these scalar data items can be used with built-in functions and UDFs as usual.)

Impala currently cannot write new data files containing complex type columns. Therefore, although the SELECT statement works for queries involving complex type columns, you cannot use a statement form that writes data to complex type columns, such as CREATE TABLE AS SELECT or INSERT ... SELECT. To create data files containing complex type data, use the Hive INSERT statement, or another ETL mechanism such as MapReduce jobs, Spark jobs, Pig, and so on.

Currently, Impala can query complex type columns only from Parquet/ORC tables or Parquet/ORC partitions within partitioned tables. Although you can use complex types in tables with Avro, text, and other file formats as part of your ETL pipeline, for example as intermediate tables populated through Hive, doing analytics through Impala requires that the data eventually ends up in a Parquet/ORC table. The requirement for Parquet/ORC data files means that you can use complex types with Impala tables hosted on other kinds of file storage systems such as Isilon and Amazon S3, but you cannot use Impala to query complex types from HBase tables.

Using complex types in SQL

When using complex types through SQL in Impala, you learn the notation for < > delimiters for the complex type columns in CREATE TABLE statements, and how to construct join queries to “unpack” the scalar values nested inside the complex data structures. You might need to condense a traditional RDBMS or data warehouse schema into a smaller number of Parquet tables, and use Hive, Spark, Pig, or other mechanism outside Impala to populate the tables with data.

Complex type syntax for DDL statements

The definition of *data_type*, as seen in the CREATE TABLE and ALTER TABLE statements, now includes complex types in addition to primitive types:

```
  primitive_type
| array_type
| map_type
| struct_type
```

Unions are not currently supported.

Array, struct, and map column type declarations are specified in the CREATE TABLE statement. You can also add or change the type of complex columns through the ALTER TABLE statement.

Currently, Impala queries allow complex types only in tables that use the Parquet format. If an Impala query encounters complex types in a table or partition using another file format, the query returns a runtime error.

You can use ALTER TABLE ... SET FILEFORMAT PARQUET to change the file format of an existing table containing complex types to Parquet, after which Impala can query it. Make sure to load Parquet files into the table after changing the file format, because the ALTER TABLE ... SET FILEFORMAT statement does not convert existing data to the new file format.

Partitioned tables can contain complex type columns. All the partition key columns must be scalar types.

Because use cases for Impala complex types require that you already have Parquet/ORC data files produced outside of Impala, you can use the Impala CREATE TABLE LIKE PARQUET syntax to produce a table with columns that match the structure of an existing Parquet file, including complex type columns for nested data structures. Remember to include the STORED AS PA RQUET clause in this case, because even with CREATE TABLE LIKE PARQUET, the default file format of the resulting table is still text.

You cannot use the CREATE TABLE AS SELECT syntax to create a table with nested type columns because the complex columns are omitted from the result set of an Impala SELECT * or SELECT *col_name* query, and because Impala currently does not support writing Parquet files with complex type columns,

Note:

Once you have a table set up with complex type columns, use the DESCRIBE and SHOW CREATE TABLE statements to see the correct notation with < and > delimiters and comma and colon separators within the complex type definitions. If you do not have existing data with the same layout as the table, you can query the empty table to practice with the notation for the SELECT statement. In the SELECT list, you use dot notation and pseudocolumns such as ITEM, KEY, and VALUE for referring to items within the complex type columns. In the FROM clause, you use join notation to construct table aliases for any referenced ARRAY and MAP columns.

For example, when defining a table that holds contact information, you might represent phone numbers differently depending on the expected layout and relationships of the data, and how well you can predict those properties in advance.

Here are different ways that you might represent phone numbers in a traditional relational schema, with equivalent representations using complex types.

Traditional relational representation of phone numbers: single table

The traditional, simplest way to represent phone numbers in a relational table is to store all contact info in a single table, with all columns having scalar types, and each potential phone number represented as a separate column. In this example, each person can only have these 3 types of phone numbers. If the person does not have a particular kind of phone number, the corresponding column is NULL for that row.

```
CREATE TABLE contacts_fixed_phones
(
    id BIGINT
  , name STRING
  , address STRING
  , home_phone STRING
  , work_phone STRING
  , mobile_phone STRING
) STORED AS PARQUET;
```

An array of phone numbers

Using a complex type column to represent the phone numbers adds some extra flexibility. Now there could be an unlimited number of phone numbers. Because the array elements have an order but not symbolic names, you could decide in advance that phone_number[0] is the home number, [1] is the work number, [2] is the mobile number, and so on. (In subsequent examples, you will see how to create a more flexible naming scheme using other complex type variations, such as a MAP or an ARRAY where each element is a STRUCT.)

```
CREATE TABLE contacts_array_of_phones
(
    id BIGINT
  , name STRING
  , address STRING
  , phone_number ARRAY < STRING >
) STORED AS PARQUET;
```

A map of phone numbers

Another way to represent an arbitrary set of phone numbers is with a MAP column. With a MAP, each element is associated with a key value that you specify, which could be a numeric, string, or other scalar type. This example uses a STRING key to give each phone number a name, such as 'home' or 'mobile'. A query could filter the data based on the key values, or display the key values in reports.

```
CREATE TABLE contacts_unlimited_phones
(
  id BIGINT, name STRING, address STRING, phone_number MAP < S
TRING,STRING >
) STORED AS PARQUET;
```

Traditional relational representation of phone numbers: normalized tables

If you are an experienced database designer, you already know how to work around the limitations of the single-table schema from *Traditional relational representation of phone numbers: single table*. By normalizing the schema, with the phone numbers in their own table, you can associate an arbitrary set of phone numbers with each person, and associate additional details with each phone number, such as whether it is a home, work, or mobile phone.

The flexibility of this approach comes with some drawbacks. Reconstructing all the data for a particular person requires a join query, which might require performance tuning on Hadoop because the data from each table might be transmitted from a different host. Data management tasks such as backups and refreshing the data require dealing with multiple tables instead of a single table.

This example illustrates a traditional database schema to store contact info normalized across 2 tables. The fact table establishes the identity and basic information about person. A dimension table stores information only about phone numbers, using an ID value to associate each phone number with a person ID from the fact table. Each person can have 0, 1, or many phones; the categories are not restricted to a few predefined ones; and the phone table can contain as many columns as desired, to represent all sorts of details about each phone number.

```
CREATE TABLE fact_contacts (id BIGINT, name STRING, address STR
ING) STORED AS PARQUET;
CREATE TABLE dim_phones
(
    contact_id BIGINT
  , category STRING
  , international_code STRING
```

```
  , area_code STRING
  , exchange STRING
  , extension STRING
  , mobile BOOLEAN
  , carrier STRING
  , current BOOLEAN
  , service_start_date TIMESTAMP
  , service_end_date TIMESTAMP
)
STORED AS PARQUET;
```

Phone Numbers Represented as an Array of Structs

To represent a schema equivalent to the one from *Traditional relational representation of phone numbers: normalized tables* using complex types, this example uses an ARRAY where each array element is a STRUCT. As with the earlier complex type examples, each person can have an arbitrary set of associated phone numbers. Making each array element into a STRUCT lets us associate multiple data items with each phone number, and give a separate name and type to each data item. The STRUCT fields of the ARRAY elements reproduce the columns of the dimension table from the previous example.

You can do all the same kinds of queries with the complex type schema as with the normalized schema from the previous example. The advantages of the complex type design are in the areas of convenience and performance. Now your backup and ETL processes only deal with a single table. When a query uses a join to cross-reference the information about a person with their associated phone numbers, all the relevant data for each row resides in the same HDFS data block, meaning each row can be processed on a single host without requiring network transmission.

```
CREATE TABLE contacts_detailed_phones
(
  id BIGINT, name STRING, address STRING
    , phone ARRAY < STRUCT <
        category: STRING
      , international_code: STRING
      , area_code: STRING
      , exchange: STRING
      , extension: STRING
      , mobile: BOOLEAN
      , carrier: STRING
      , current: BOOLEAN
      , service_start_date: TIMESTAMP
      , service_end_date: TIMESTAMP
    >>
) STORED AS PARQUET;
```

SQL statements that support complex types

The Impala SQL statements that support complex types are currently CREATE_TABLE, ALTE R_TABLE, DESCRIBE, LOAD_DATA, and SELECT. That is, currently Impala can create or alter tables containing complex type columns, examine the structure of a table containing complex type columns, import existing data files containing complex type columns into a table, and query Parquet/ORC tables containing complex types.

Impala currently cannot write new data files containing complex type columns. Therefore, although the SELECT statement works for queries involving complex type columns, you cannot use a statement form that writes data to complex type columns, such as CREATE TABLE AS SELECT or INSERT ... SELECT. To create data files containing complex type data, use the Hive INSERT statement, or another ETL mechanism such as MapReduce jobs, Spark jobs, Pig, and so on.

DDL statements and complex types

Column specifications for complex or nested types use < and > delimiters:

```
-- What goes inside the < > for an ARRAY is a single type, either
 a scalar or another
-- complex type (ARRAY, STRUCT, or MAP).
CREATE TABLE array_t
(
  id BIGINT,
  a1 ARRAY <STRING>,
  a2 ARRAY <BIGINT>,
  a3 ARRAY <TIMESTAMP>,
  a4 ARRAY <STRUCT <f1: STRING, f2: INT, f3: BOOLEAN>>
)
STORED AS PARQUET;

-- What goes inside the < > for a MAP is two comma-separated t
ypes specifying the types of the key-value pair:
-- a scalar type representing the key, and a scalar or complex
type representing the value.
CREATE TABLE map_t
(
  id BIGINT,
  m1 MAP <STRING, STRING>,
  m2 MAP <STRING, BIGINT>,
  m3 MAP <BIGINT, STRING>,
  m4 MAP <BIGINT, BIGINT>,
  m5 MAP <STRING, ARRAY <STRING>>
)
STORED AS PARQUET;

-- What goes inside the < > for a STRUCT is a comma-separated lis
t of fields, each field defined as
-- name:type. The type can be a scalar or a complex type. The fie
ld names for each STRUCT do not clash
-- with the names of table columns or fields in other STRUCTs. A
 STRUCT is most often used inside
-- an ARRAY or a MAP rather than as a top-level column.
CREATE TABLE struct_t
(
  id BIGINT,
  s1 STRUCT <f1: STRING, f2: BIGINT>,
  s2 ARRAY <STRUCT <f1: INT, f2: TIMESTAMP>>,
  s3 MAP <BIGINT, STRUCT <name: STRING, birthday: TIMESTAMP>>
)
STORED AS PARQUET;
```

Queries and complex types

The result set of an Impala query always contains all scalar types; the elements and fields within any complex type queries must be “unpacked” using join queries. A query cannot directly retrieve the entire value for a complex type column. Impala returns an error in this case. Queries using SELECT * are allowed for tables with complex types, but the columns with complex types are skipped.

The following example shows how referring directly to a complex type column returns an error, while SELECT * on the same table succeeds, but only retrieves the scalar columns.

Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types for the table definitions.

```
SELECT c_orders FROM customer LIMIT 1;
```

```
ERROR: AnalysisException: Expr 'c_orders' in select list returns
a complex type 'ARRAY<STRUCT<o_orderkey:BIGINT,o_orderstatus:STR
ING, ... l_receiptdate:STRING,l_shipinstruct:STRING,l_shipmode:S
TRING,l_comment:STRING>>>>'.
Only scalar types are allowed in the select list.

-- Original column has several scalar and one complex column.
DESCRIBE customer;
+--------------+-------------------------------------+
| name         | type                                |
+--------------+-------------------------------------+
| c_custkey    | bigint                              |
| c_name       | string                              |
...
| c_orders     | array<struct<                       |
|              |   o_orderkey:bigint,                |
|              |   o_orderstatus:string,             |
|              |   o_totalprice:decimal(12,2),       |
...
|              | >>                                  |
+--------------+-------------------------------------+

-- When we SELECT * from that table, only the scalar columns come
 back in the result set.
CREATE TABLE select_star_customer STORED AS PARQUET AS SELECT *
 FROM customer;
+------------------------+
| summary                |
+------------------------+
| Inserted 150000 row(s) |
+------------------------+
-- The c_orders column, being of complex type, was not included
 in the SELECT * result set.
DESC select_star_customer;
+--------------+---------------+
| name         | type          |
+--------------+---------------+
| c_custkey    | bigint        |
| c_name       | string        |
| c_address    | string        |
| c_nationkey  | smallint      |
| c_phone      | string        |
| c_acctbal    | decimal(12,2) |
| c_mktsegment | string        |
| c_comment    | string        |
+--------------+---------------+
```

References to fields within STRUCT columns use dot notation. If the field name is unambiguous, you can omit qualifiers such as table name, column name, or even the ITEM or VALUE pseudocolumn names for STRUCT elements inside an ARRAY or a MAP.

```
SELECT id, address.city FROM customers WHERE address.zip = 94305;
```

References to elements within ARRAY columns use the ITEM pseudocolumn:

```
select r_name, r_nations.item.n_name from region, region.r_natio
ns limit 7;
+--------+----------------+
| r_name | item.n_name    |
+--------+----------------+
| EUROPE | UNITED KINGDOM |
| EUROPE | RUSSIA         |
```

```
| EUROPE | ROMANIA        |
| EUROPE | GERMANY        |
| EUROPE | FRANCE         |
| ASIA   | VIETNAM        |
| ASIA   | CHINA          |
+--------+----------------+
```

References to fields within MAP columns use the KEY and VALUE pseudocolumns. In this example, once the query establishes the alias MAP_FIELD for a MAP column with a STRING key and an INT value, the query can refer to MAP_FIELD.KEY and MAP_FIELD.VALUE, which have zero, one, or many instances for each row from the containing table.

```
DESCRIBE table_0;
+---------+-----------------------+
| name    | type                  |
+---------+-----------------------+
| field_0 | string                |
| field_1 | map<string,int>       |
...

SELECT field_0, map_field.key, map_field.value
  FROM table_0, table_0.field_1 AS map_field
WHERE length(field_0) = 1
LIMIT 10;
+---------+-----------+-------+
| field_0 | key       | value |
+---------+-----------+-------+
| b       | gshsgkvd  | NULL  |
| b       | twrtcxj6  | 18    |
| b       | 2vp5      | 39    |
| b       | fh0s      | 13    |
| v       | 2         | 41    |
| v       | 8b58mz    | 20    |
| v       | hw        | 16    |
| v       | 65l388pyt | 29    |
| v       | 03k68g91z | 30    |
| v       | r2hlg5b   | NULL  |
+---------+-----------+-------+
```

When complex types are nested inside each other, you use a combination of joins, pseudocolumn names, and dot notation to refer to specific fields at the appropriate level. This is the most frequent form of query syntax for complex columns, because the typical use case involves two levels of complex types, such as an ARRAY of STRUCT elements.

```
SELECT id, phone_numbers.area_code FROM contact_info_many_structs
 INNER JOIN contact_info_many_structs.phone_numbers phone_numbers
 LIMIT 3;
```

You can express relationships between ARRAY and MAP columns at different levels as joins. You include comparison operators between fields at the top level and within the nested type columns so that Impala can do the appropriate join operation.

Note: Many of the complex type examples refer to tables such as CUSTOMER and REGION adapted from the tables used in the TPC-H benchmark. See Sample Schema and Data for Experimenting with Impala Complex Types for the table definitions.

For example, the following queries work equivalently. They each return customer and order data for customers that have at least one order.

```
SELECT c.c_name, o.o_orderkey FROM customer c, c.c_orders o LIMIT
 5;
+--------------------+------------+
| c_name             | o_orderkey |
+--------------------+------------+
| Customer#000072578 | 558821     |
| Customer#000072578 | 2079810    |
| Customer#000072578 | 5768068    |
| Customer#000072578 | 1805604    |
| Customer#000072578 | 3436389    |
+--------------------+------------+

SELECT c.c_name, o.o_orderkey FROM customer c INNER JOIN c.c_o
rders o LIMIT 5;
+--------------------+------------+
| c_name             | o_orderkey |
+--------------------+------------+
| Customer#000072578 | 558821     |
| Customer#000072578 | 2079810    |
| Customer#000072578 | 5768068    |
| Customer#000072578 | 1805604    |
| Customer#000072578 | 3436389    |
+--------------------+------------+
```

The following query using an outer join returns customers that have orders, plus customers with no orders (no entries in the C_ORDERS array):

```
SELECT c.c_custkey, o.o_orderkey
  FROM customer c LEFT OUTER JOIN c.c_orders o
LIMIT 5;
+-----------+------------+
| c_custkey | o_orderkey |
+-----------+------------+
| 60210     | NULL       |
| 147873    | NULL       |
| 72578     | 558821     |
| 72578     | 2079810    |
| 72578     | 5768068    |
+-----------+------------+
```

The following query returns only customers that have no orders. (With LEFT ANTI JOIN or LEFT SEMI JOIN, the query can only refer to columns from the left-hand table, because by definition there is no matching information in the right-hand table.)

```
SELECT c.c_custkey, c.c_name
  FROM customer c LEFT ANTI JOIN c.c_orders o
LIMIT 5;
+-----------+--------------------+
| c_custkey | c_name             |
+-----------+--------------------+
| 60210     | Customer#000060210 |
| 147873    | Customer#000147873 |
| 141576    | Customer#000141576 |
| 85365     | Customer#000085365 |
| 70998     | Customer#000070998 |
+-----------+--------------------+
```

You can also perform correlated subqueries to examine the properties of complex type columns for each row in the result set.

Count the number of orders per customer. Note the correlated reference to the table alias C. The COUNT(*) operation applies to all the elements of the C_ORDERS array for the corresponding row, avoiding the need for a GROUP BY clause.

```
select c_name, howmany FROM customer c, (SELECT COUNT(*) howmany
 FROM c.c_orders) v limit 5;
+--------------------+---------+
| c_name             | howmany |
+--------------------+---------+
| Customer#000030065 | 15      |
| Customer#000065455 | 18      |
| Customer#000113644 | 21      |
| Customer#000111078 | 0       |
| Customer#000024621 | 0       |
+--------------------+---------+
```

Count the number of orders per customer, ignoring any customers that have not placed any orders:

```
SELECT c_name, howmany_orders
FROM
  customer c,
  (SELECT COUNT(*) howmany_orders FROM c.c_orders) subq1
WHERE howmany_orders > 0
LIMIT 5;
+--------------------+----------------+
| c_name             | howmany_orders |
+--------------------+----------------+
| Customer#000072578 | 7              |
| Customer#000046378 | 26             |
| Customer#000069815 | 11             |
| Customer#000079058 | 12             |
| Customer#000092239 | 26             |
+--------------------+----------------+
```

Count the number of line items in each order. The reference to C.C_ORDERS in the FROM clause is needed because the O_ORDERKEY field is a member of the elements in the C_ORDERS array. The subquery labelled SUBQ1 is correlated: it is re-evaluated for the C_ORDERS.O_LINEITEMS array from each row of the CUSTOMERS table.

```
SELECT c_name, o_orderkey, howmany_line_items
FROM
  customer c,
  c.c_orders t2,
  (SELECT COUNT(*) howmany_line_items FROM c.c_orders.o_lineit
ems) subq1
WHERE howmany_line_items > 0
LIMIT 5;
+--------------------+------------+--------------------+
| c_name             | o_orderkey | howmany_line_items |
+--------------------+------------+--------------------+
| Customer#000020890 | 1884930    | 95                 |
| Customer#000020890 | 4570754    | 95                 |
| Customer#000020890 | 3771072    | 95                 |
| Customer#000020890 | 2555489    | 95                 |
| Customer#000020890 | 919171     | 95                 |
+--------------------+------------+--------------------+
```

Get the number of orders, the average order price, and the maximum items in any order per customer. For this example, the subqueries labelled SUBQ1 and SUBQ2 are correlated: they are

re-evaluated for each row from the original CUSTOMER table, and only apply to the complex columns associated with that row.

```
SELECT c_name, howmany, average_price, most_items
FROM
  customer c,
  (SELECT COUNT(*) howmany, AVG(o_totalprice) average_price FROM
 c.c_orders) subq1,
  (SELECT MAX(l_quantity) most_items FROM c.c_orders.o_lineitem
s ) subq2
LIMIT 5;
+--------------------+---------+---------------+------------+
| c_name             | howmany | average_price | most_items |
+--------------------+---------+---------------+------------+
| Customer#000030065 | 15      | 128908.34     | 50.00      |
| Customer#000088191 | 0       | NULL          | NULL       |
| Customer#000101555 | 10      | 164250.31     | 50.00      |
| Customer#000022092 | 0       | NULL          | NULL       |
| Customer#000036277 | 27      | 166040.06     | 50.00      |
+--------------------+---------+---------------+------------+
```

For example, these queries show how to access information about the ARRAY elements within the CUSTOMER table from the “nested TPC-H” schema, starting with the initial ARRAY elements and progressing to examine the STRUCT fields of the ARRAY, and then the elements nested within another ARRAY of STRUCT:

```
-- How many orders does each customer have?
-- The type of the ARRAY column doesn't matter, this is just cou
nting the elements.
SELECT c_custkey, count(*)
  FROM customer, customer.c_orders
GROUP BY c_custkey
LIMIT 5;
+-----------+----------+
| c_custkey | count(*) |
+-----------+----------+
| 61081     | 21       |
| 115987    | 15       |
| 69685     | 19       |
| 109124    | 15       |
| 50491     | 12       |
+-----------+----------+

-- How many line items are part of each customer order?
-- Now we examine a field from a STRUCT nested inside the ARRAY.
SELECT c_custkey, c_orders.o_orderkey, count(*)
  FROM customer, customer.c_orders c_orders, c_orders.o_lineitems
GROUP BY c_custkey, c_orders.o_orderkey
LIMIT 5;
+-----------+------------+----------+
| c_custkey | o_orderkey | count(*) |
+-----------+------------+----------+
| 63367     | 4985959    | 7        |
| 53989     | 1972230    | 2        |
| 143513    | 5750498    | 5        |
| 17849     | 4857989    | 1        |
| 89881     | 1046437    | 1        |
+-----------+------------+----------+

-- What are the line items in each customer order?
-- One of the STRUCT fields inside the ARRAY is another
-- ARRAY containing STRUCT elements. The join finds
```

```
-- all the related items from both levels of ARRAY.
SELECT c_custkey, o_orderkey, l_partkey
  FROM customer, customer.c_orders, c_orders.o_lineitems
LIMIT 5;
+-----------+------------+-----------+
| c_custkey | o_orderkey | l_partkey |
+-----------+------------+-----------+
| 113644    | 2738497    | 175846    |
| 113644    | 2738497    | 27309     |
| 113644    | 2738497    | 175873    |
| 113644    | 2738497    | 88559     |
| 113644    | 2738497    | 8032      |
+-----------+------------+-----------+
```

Pseudocolumns for ARRAY and MAP types

Each element in an ARRAY type has a position, indexed starting from zero, and a value. Each element in a MAP type represents a key-value pair. Impala provides pseudocolumns that let you retrieve this metadata as part of a query, or filter query results by including such things in a WHERE clause. You refer to the pseudocolumns as part of qualified column names in queries:

- ITEM: The value of an array element. If the ARRAY contains STRUCT elements, you can refer to either *array_name*.ITEM.*field_name* or use the shorthand *array_name.field_name*.
- POS: The position of an element within an array.
- KEY: The value forming the first part of a key-value pair in a map. It is not necessarily unique.
- VALUE: The data item forming the second part of a key-value pair in a map. If the VALUE part of the MAP element is a STRUCT, you can refer to either *map_name*.VALUE.*field_name* or use the shorthand *map_name*.*field_name*.

ITEM and POS pseudocolumns

When an ARRAY column contains STRUCT elements, you can refer to a field within the STRUCT using a qualified name of the form *array_column*.*field_name*. If the ARRAY contains scalar values, Impala recognizes the special name *array_column*.ITEM to represent the value of each scalar array element. For example, if a column contained an ARRAY where each element was a STRING, you would use *array_name*.ITEM to refer to each scalar value in the SELECT list, or the WHERE or other clauses.

This example shows a table with two ARRAY columns whose elements are of the scalar type STRI NG. When referring to the values of the array elements in the SELECT list, WHERE clause, or ORDER BY clause, you use the ITEM pseudocolumn because within the array, the individual elements have no defined names.

```
create TABLE persons_of_interest
(
person_id BIGINT,
aliases ARRAY <STRING>,
associates ARRAY <STRING>,
real_name STRING
)
STORED AS PARQUET;

-- Get all the aliases of each person.
SELECT real_name, aliases.ITEM
  FROM persons_of_interest, persons_of_interest.aliases
ORDER BY real_name, aliases.item;

-- Search for particular associates of each person.
SELECT real_name, associates.ITEM
  FROM persons_of_interest, persons_of_interest.associates
WHERE associates.item LIKE '% MacGuffin';
```

Because an array is inherently an ordered data structure, Impala recognizes the special name *array_column*.POS to represent the numeric position of each element within the array. The POS pseudocolumn lets you filter or reorder the result set based on the sequence of array elements.

The following example uses a table from a flattened version of the TPC-H schema. The REGION table only has a few rows, such as one row for Europe and one for Asia. The row for each region represents all the countries in that region as an ARRAY of STRUCT elements:

```
[localhost:21000] > desc region;
+-------------+-------------------------------------------------
--------------------+
| name        | type
                   |
+-------------+-------------------------------------------------
------------------+
| r_regionkey | smallint
                   |
| r_name      | string
                   |
| r_comment   | string
                   |
| r_nations   | array<struct<n_nationkey:smallint,n_name:string,n
_comment:string>> |
+-------------+-------------------------------------------------
--------------------+
```

To find the countries within a specific region, you use a join query. To find out the order of elements in the array, you also refer to the POS pseudocolumn in the select list:

```
[localhost:21000] > SELECT r1.r_name, r2.n_name, r2.POS
                  > FROM region r1 INNER JOIN r1.r_nations r2
                  > WHERE r1.r_name = 'ASIA';
+--------+-----------+-----+
| r_name | n_name    | pos |
+--------+-----------+-----+
| ASIA   | VIETNAM   | 0   |
| ASIA   | CHINA     | 1   |
| ASIA   | JAPAN     | 2   |
| ASIA   | INDONESIA | 3   |
| ASIA   | INDIA     | 4   |
+--------+-----------+-----+
```

Once you know the positions of the elements, you can use that information in subsequent queries, for example to change the ordering of results from the complex type column or to filter certain elements from the array:

```
[localhost:21000] > SELECT r1.r_name, r2.n_name, r2.POS
                  > FROM region r1 INNER JOIN r1.r_nations r2
                  > WHERE r1.r_name = 'ASIA'
                  > ORDER BY r2.POS DESC;
+--------+-----------+-----+
| r_name | n_name    | pos |
+--------+-----------+-----+
| ASIA   | INDIA     | 4   |
| ASIA   | INDONESIA | 3   |
| ASIA   | JAPAN     | 2   |
| ASIA   | CHINA     | 1   |
| ASIA   | VIETNAM   | 0   |
+--------+-----------+-----+
[localhost:21000] > SELECT r1.r_name, r2.n_name, r2.POS
                  > FROM region r1 INNER JOIN r1.r_nations r2
```

```
                  > WHERE r1.r_name = 'ASIA' AND r2.POS BETWEEN 1
 and 3;
+--------+-----------+-----+
| r_name | n_name    | pos |
+--------+-----------+-----+
| ASIA   | CHINA     | 1   |
| ASIA   | JAPAN     | 2   |
| ASIA   | INDONESIA | 3   |
+--------+-----------+-----+
```

KEY and VALUE pseudocolumns

The MAP data type is suitable for representing sparse or wide data structures, where each row might only have entries for a small subset of named fields. Because the element names (the map keys) vary depending on the row, a query must be able to refer to both the key and the value parts of each key-value pair. The KEY and VALUE pseudocolumns let you refer to the parts of the key-value pair independently within the query, as *map_column*.KEY and *map_column*.VALUE.

The KEY must always be a scalar type, such as STRING, BIGINT, or TIMESTAMP. It can be NULL. Values of the KEY field are not necessarily unique within the same MAP. You apply any required DISTINCT, GROUP BY, and other clauses in the query, and loop through the result set to process all the values matching any specified keys.

The VALUE can be either a scalar type or another complex type. If the VALUE is a STRUCT, you can construct a qualified name *map_column*.VALUE.*struct_field* to refer to the individual fields inside the value part. If the VALUE is an ARRAY or another MAP, you must include another join condition that establishes a table alias for *map_column*.VALUE, and then construct another qualified name using that alias, for example *table_alias*.ITEM or *table_alias*.KEY and *table_alias*.VALUE

The following example shows different ways to access a MAP column using the KEY and VALUE pseudocolumns. The DETAILS column has a STRING first part with short, standardized values such as 'Recurring', 'Lucid', or 'Anxiety'. This is the “key” that is used to look up particular kinds of elements from the MAP. The second part, also a STRING, is a longer free-form explanation. Impala gives you the standard pseudocolumn names KEY and VALUE for the two parts, and you apply your own conventions and interpretations to the underlying values.

Note: If you find that the single-item nature of the VALUE makes it difficult to model your data accurately, the solution is typically to add some nesting to the complex type. For example, to have several sets of key-value pairs, make the column an ARRAY whose elements are MAP. To make a set of key-value pairs that holds more elaborate information, make a MAP column whose VALUE part contains an ARRAY or a STRUCT.

```
CREATE TABLE dream_journal
(
  dream_id BIGINT,
  details MAP <STRING,STRING>
)
STORED AS PARQUET;

-- What are all the types of dreams that are recorded?
SELECT DISTINCT details.KEY FROM dream_journal, dream_journal.de
tails;

-- How many lucid dreams were recorded?
-- Because there is no GROUP BY, we count the 'Lucid' keys across
 all rows.
SELECT COUNT(details.KEY)
  FROM dream_journal, dream_journal.details
WHERE details.KEY = 'Lucid';
```

```
-- Print a report of a subset of dreams, filtering based on both
the lookup key
-- and the detailed value.
SELECT dream_id, details.KEY AS "Dream Type", details.VALUE AS
 "Dream Summary"
  FROM dream_journal, dream_journal.details
WHERE
  details.KEY IN ('Happy', 'Pleasant', 'Joyous')
  AND details.VALUE LIKE '%childhood%';
```

The following example shows a more elaborate version of the previous table, where the VALUE part of the MAP entry is a STRUCT rather than a scalar type. Now instead of referring to the VALUE pseudocolumn directly, you use dot notation to refer to the STRUCT fields inside it.

```
CREATE TABLE better_dream_journal
(
  dream_id BIGINT,
  details MAP <STRING,STRUCT <summary: STRING, when_happened: TIM
ESTAMP, duration: DECIMAL(5,2), woke_up: BOOLEAN> >
)
STORED AS PARQUET;


-- Do more elaborate reporting and filtering by examining multipl
e attributes within the same dream.
SELECT dream_id, details.KEY AS "Dream
 Type", details.VALUE.summary AS "Dream
 Summary", details.VALUE.duration AS "Duration"
  FROM better_dream_journal, better_dream_journal.details
WHERE
  details.KEY IN ('Anxiety', 'Nightmare')
  AND details.VALUE.duration > 60
  AND details.VALUE.woke_up = TRUE;

-- Remember that if the ITEM or VALUE contains a STRUCT, you can
reference
-- the STRUCT fields directly without the .ITEM or .VALUE qual
ifier.
SELECT dream_id, details.KEY AS "Dream Type", details.summary AS
 "Dream Summary", details.duration AS "Duration"
  FROM better_dream_journal, better_dream_journal.details
WHERE
  details.KEY IN ('Anxiety', 'Nightmare')
  AND details.duration > 60
  AND details.woke_up = TRUE;
```

Loading data containing complex types

Because the Impala INSERT statement does not currently support creating new data with complex type columns, or copying existing complex type values from one table to another, you primarily use Impala to query Parquet/ORC tables with complex types where the data was inserted through Hive, or create tables with complex types where you already have existing Parquet/ORC data files.

Because the Impala INSERT statement does not currently support creating new data with complex type columns, or copying existing complex type values from one table to another, you primarily use Impala to query Parquet/ORC tables with complex types where the data was inserted through Hive, or create tables with complex types where you already have existing Parquet/ORC data files.

If you have existing Parquet data files containing complex types, located outside of any Impala or Hive table, such as data files created by Spark jobs, you can use an Impala CREATE TABLE ... STORED AS PARQUET statement, followed by an Impala LOAD DATA statement to move the data files into the table. As an alternative, you can use an Impala CREATE EXTERNAL TABLE

statement to create a table pointing to the HDFS directory that already contains the Parquet or ORC data files.

The simplest way to get started with complex type data is to take a denormalized table containing duplicated values, and use an INSERT ... SELECT statement to copy the data into a Parquet table and condense the repeated values into complex types. With the Hive INSERT statement, you use the COLLECT_LIST(), NAMED_STRUCT(), and MAP() constructor functions within a GROUP BY query to produce the complex type values. COLLECT_LIST() turns a sequence of values into an ARRAY. NAMED_STRUCT() uses the first, third, and so on arguments as the field names for a STRUCT, to match the field names from the CREATE TABLE statement.

Note: Because Hive currently cannot construct individual rows using complex types through the INSERT ... VALUES syntax, you prepare the data in flat form in a separate table, then copy it to the table with complex columns using INSERT ... SELECT and the complex type constructors.

Using complex types as nested types

The ARRAY, STRUCT, and MAP types can be the top-level types for “nested type” columns. That is, each of these types can contain other complex or scalar types, with multiple levels of nesting to a maximum depth of 100. For example, you can have an array of structures, a map containing other maps, a structure containing an array of other structures, and so on. At the lowest level, there are always scalar types making up the fields of a STRUCT, elements of an ARRAY, and keys and values of a MAP.

Schemas involving complex types typically use some level of nesting for the complex type columns.

For example, to model a relationship like a dimension table and a fact table, you typically use an ARRAY where each array element is a STRUCT. The STRUCT fields represent what would traditionally be columns in a separate joined table. It makes little sense to use a STRUCT as the top-level type for a column, because you could just make the fields of the STRUCT into regular table columns.

Perhaps the only use case for a top-level STRUCT would be to to allow STRUCT fields with the same name as columns to coexist in the same table. The following example shows how a table could have a column named ID, and two separate STRUCT fields also named ID. Because the STRUCT fields are always referenced using qualified names, the identical ID names do not cause a conflict.

```
CREATE TABLE struct_namespaces
(
  id BIGINT
  , s1 STRUCT < id: BIGINT, field1: STRING >
  , s2 STRUCT < id: BIGINT, when_happened: TIMESTAMP >
)
STORED AS PARQUET;

select id, s1.id, s2.id from struct_namespaces;
```

It is common to make the value portion of each key-value pair in a MAP a STRUCT, ARRAY of STRUCT, or other complex type variation. That way, each key in the MAP can be associated with a flexible and extensible data structure. The key values are not predefined ahead of time (other than by specifying their data type). Therefore, the MAP can accommodate a rapidly evolving schema, or sparse data structures where each row contains only a few data values drawn from a large set of possible choices.

Although you can use an ARRAY of scalar values as the top-level column in a table, such a simple array is typically of limited use for analytic queries. The only property of the array elements, aside from the element value, is the ordering sequence available through the POS pseudocolumn. To record any additional item about each array element, such as a TIMESTAMP or a symbolic name, you use an ARRAY of STRUCT rather than of scalar values.

If you are considering having multiple ARRAY or MAP columns, with related items under the same position in each ARRAY or the same key in each MAP, prefer to use a STRUCT to group all the related items into a single ARRAY or MAP. Doing so avoids the additional storage overhead and potential duplication of key values from having an extra complex type column. Also, because each ARRAY or MAP that you reference in the query SELECT list requires an additional join clause, minimizing the number of complex type columns also makes the query easier to read and maintain, relying more on dot notation to refer to the relevant fields rather than a sequence of join clauses.

For example, here is a table with several complex type columns all at the top level and containing only scalar types. To retrieve every data item for the row requires a separate join for each ARRAY or MAP column. The fields of the STRUCT can be referenced using dot notation, but there is no real advantage to using the STRUCT at the top level rather than just making separate columns FIEL D1 and FIELD2.

```
CREATE TABLE complex_types_top_level
(
  id BIGINT,
  a1 ARRAY<INT>,
  a2 ARRAY<STRING>,
  s STRUCT<field1: INT, field2: STRING>,
-- Numeric lookup key for a string value.
  m1 MAP<INT,STRING>,
-- String lookup key for a numeric value.
  m2 MAP<STRING,INT>
)
STORED AS PARQUET;

describe complex_types_top_level;
+------+-----------------+
| name | type            |
+------+-----------------+
| id   | bigint          |
| a1   | array<int>      |
| a2   | array<string>   |
| s    | struct<         |
|      |   field1:int,   |
|      |   field2:string |
|      | >               |
| m1   | map<int,string> |
| m2   | map<string,int> |
+------+-----------------+
select
  id,
  a1.item,
  a2.item,
  s.field1,
  s.field2,
  m1.key,
  m1.value,
  m2.key,
  m2.value
from
  complex_types_top_level,
  complex_types_top_level.a1,
  complex_types_top_level.a2,
  complex_types_top_level.m1,
  complex_types_top_level.m2;
```

For example, here is a table with columns containing an ARRAY of STRUCT, a MAP where each key value is a STRUCT, and a MAP where each key value is an ARRAY of STRUCT.

```
CREATE TABLE nesting_demo
```

```
(
  user_id BIGINT,
  family_members ARRAY < STRUCT < name: STRING, email: STRING,
 date_joined: TIMESTAMP >>,
  foo map < STRING, STRUCT < f1: INT, f2: INT, f3: TIMESTAMP, f4:
 BOOLEAN >>,
  gameplay MAP < STRING , ARRAY < STRUCT <
    name: STRING, highest: BIGINT, lives_used: INT, total_spent:
 DECIMAL(16,2)
  >>>
)
STORED AS PARQUET;
```

The DESCRIBE statement rearranges the < and > separators and the field names within each STRU CT for easy readability:

```
DESCRIBE nesting_demo;
+----------------+-----------------------------+
| name           | type                        |
+----------------+-----------------------------+
| user_id        | bigint                      |
| family_members | array<struct<               |
|                |   name:string,              |
|                |   email:string,             |
|                |   date_joined:timestamp     |
|                | >>                          |
| foo            | map<string,struct<          |
|                |   f1:int,                   |
|                |   f2:int,                   |
|                |   f3:timestamp,             |
|                |   f4:boolean                |
|                | >>                          |
| gameplay       | map<string,array<struct<    |
|                |   name:string,              |
|                |   highest:bigint,           |
|                |   lives_used:int,           |
|                |   total_spent:decimal(16,2) |
|                | >>>                         |
+----------------+-----------------------------+
```

To query the complex type columns, you use join notation to refer to the lowest-level scalar values. If the value is an ARRAY element, the fully qualified name includes the ITEM pseudocolumn. If the value is inside a MAP, the fully qualified name includes the KEY or VALUE pseudocolumn. Each reference to a different ARRAY or MAP (even if nested inside another complex type) requires an additional join clause.

```
SELECT
-- The lone scalar field doesn't require any dot notation or join
 clauses.
    user_id
-- Retrieve the fields of a STRUCT inside an ARRAY.
-- The FAMILY_MEMBERS name refers to the FAMILY_MEMBERS table
alias defined later in the FROM clause.
  , family_members.item.name
  , family_members.item.email
  , family_members.item.date_joined
-- Retrieve the KEY and VALUE fields of a MAP, with the value bei
ng a STRUCT consisting of more fields.
-- The FOO name refers to the FOO table alias defined later in th
e FROM clause.
  , foo.key
  , foo.value.f1
```

```
  , foo.value.f2
  , foo.value.f3
  , foo.value.f4
-- Retrieve the KEY fields of a MAP, and expand the VALUE part in
to ARRAY items consisting of STRUCT fields.
-- The GAMEPLAY name refers to the GAMEPLAY table alias defined
later in the FROM clause (referring to the MAP item).
-- The GAME_N name refers to the GAME_N table alias defined la
ter in the FROM clause (referring to the ARRAY
-- inside the MAP item's VALUE part.)
  , gameplay.key
  , game_n.name
  , game_n.highest
  , game_n.lives_used
  , game_n.total_spent
FROM
    nesting_demo
  , nesting_demo.family_members AS family_members
  , nesting_demo.foo AS foo
  , nesting_demo.gameplay AS gameplay
  , nesting_demo.gameplay.value AS game_n;
```

Once you understand the notation to refer to a particular data item in the SELECT list, you can use the same qualified name to refer to that data item in other parts of the query, such as the WHERE clause, ORDER BY or GROUP BY clauses, or calls to built-in functions. For example, you might frequently retrieve the VALUE part of each MAP item in the SELECT list, while choosing the specific MAP items by running comparisons against the KEY part in the WHERE clause.

Accessing complex type data in flattened form using views

The layout of complex and nested types is largely a physical consideration. The complex type columns reside in the same data files rather than in separate normalized tables, for your convenience in managing related data sets and performance in querying related data sets. You can use views to treat tables with complex types as if they were flattened. By putting the join logic and references to the complex type columns in the view definition, you can query the same tables using existing queries intended for tables containing only scalar columns. This technique also lets you use tables with complex types with BI tools that are not aware of the data types and query notation for accessing complex type columns.

For example, the variation of the TPC-H schema containing complex types has a table REGION. This table has 5 rows, corresponding to 5 regions such as NORTH AMERICA and AFRICA. Each row has an ARRAY column, where each array item is a STRUCT containing details about a country in that region.

```
DESCRIBE region;
+-------------+-------------------------+
| name        | type                    |
+-------------+-------------------------+
| r_regionkey | smallint                |
| r_name      | string                  |
| r_comment   | string                  |
| r_nations   | array<struct<           |
|             |   n_nationkey:smallint, |
|             |   n_name:string,        |
|             |   n_comment:string      |
|             | >>                      |
+-------------+-------------------------+
```

The same data could be represented in traditional denormalized form, as a single table where the information about each region is repeated over and over, alongside the information about each country. The nested complex types let us avoid the repetition, while still keeping the data in a single table rather than normalizing across multiple tables.

To use this table with a JDBC or ODBC application that expected scalar columns, we could create a view that represented the result set as a set of scalar columns (three columns from the original table, plus three more from the STRUCT fields of the array elements). In the following examples, any column with an R_* prefix is taken unchanged from the original table, while any column with an N_* prefix is extracted from the STRUCT inside the ARRAY.

```
CREATE VIEW region_view AS
  SELECT
    r_regionkey,
    r_name,
    r_comment,
    array_field.item.n_nationkey AS n_nationkey,
    array_field.item.n_name AS n_name,
    array_field.n_comment AS n_comment
FROM
  region, region.r_nations AS array_field;
```

Then we point the application queries at the view rather than the original table. From the perspective of the view, there are 25 rows in the result set, one for each nation in each region, and queries can refer freely to fields related to the region or the nation.

```
-- Retrieve info such as the nation name from the original R_NAT
IONS array elements.
select n_name from region_view where r_name in ('EUROPE', 'ASIA
');
+----------------+
| n_name         |
+----------------+
| UNITED KINGDOM |
| RUSSIA         |
| ROMANIA        |
| GERMANY        |
| FRANCE         |
| VIETNAM        |
| CHINA          |
| JAPAN          |
| INDONESIA      |
| INDIA          |
+----------------+

-- UNITED STATES in AMERICA and UNITED KINGDOM in EUROPE.
SELECT DISTINCT r_name FROM region_view WHERE n_name LIKE 'UNITED
%';
+---------+
| r_name  |
+---------+
| AMERICA |
| EUROPE  |
+---------+

-- For conciseness, we only list some view columns in the SELECT
list.
-- SELECT * would bring back all the data, unlike SELECT *
-- queries on the original table with complex type columns.
SELECT r_regionkey, r_name, n_nationkey, n_name FROM region_view
 LIMIT 7;
+-------------+--------+-------------+----------------+
| r_regionkey | r_name | n_nationkey | n_name         |
+-------------+--------+-------------+----------------+
| 3           | EUROPE | 23          | UNITED KINGDOM |
| 3           | EUROPE | 22          | RUSSIA         |
| 3           | EUROPE | 19          | ROMANIA        |
```

```
| 3           | EUROPE | 7           | GERMANY        |
| 3           | EUROPE | 6           | FRANCE         |
| 2           | ASIA   | 21          | VIETNAM        |
| 2           | ASIA   | 18          | CHINA          |
+-------------+--------+-------------+----------------+
```

Tutorials and examples for complex types

The following examples illustrate the query syntax for some common use cases involving complex type columns:

Sample schema and data for experimenting with Impala complex types

The tables used for earlier examples of complex type syntax are trivial ones with no actual data. The more substantial examples of the complex type feature use these tables, adapted from the schema used for TPC-H testing:

```
SHOW TABLES;
+----------+
| name     |
+----------+
| customer |
| part     |
| region   |
| supplier |
+----------+

DESCRIBE customer;
+--------------+-------------------------------------+
| name         | type                                |
+--------------+-------------------------------------+
| c_custkey    | bigint                              |
| c_name       | string                              |
| c_address    | string                              |
| c_nationkey  | smallint                            |
| c_phone      | string                              |
| c_acctbal    | decimal(12,2)                       |
| c_mktsegment | string                              |
| c_comment    | string                              |
| c_orders     | array<struct<                       |
|              |   o_orderkey:bigint,                |
|              |   o_orderstatus:string,             |
|              |   o_totalprice:decimal(12,2),       |
|              |   o_orderdate:string,               |
|              |   o_orderpriority:string,           |
|              |   o_clerk:string,                   |
|              |   o_shippriority:int,               |
|              |   o_comment:string,                 |
|              |   o_lineitems:array<struct<         |
|              |     l_partkey:bigint,               |
|              |     l_suppkey:bigint,               |
|              |     l_linenumber:int,               |
|              |     l_quantity:decimal(12,2),       |
|              |     l_extendedprice:decimal(12,2),  |
|              |     l_discount:decimal(12,2),       |
|              |     l_tax:decimal(12,2),            |
|              |     l_returnflag:string,            |
|              |     l_linestatus:string,            |
|              |     l_shipdate:string,              |
|              |     l_commitdate:string,            |
|              |     l_receiptdate:string,           |
|              |     l_shipinstruct:string,          |
|              |     l_shipmode:string,              |
|              |     l_comment:string                |
```

```
|              |   >>                                  |
|              | >>                                    |
+--------------+---------------------------------------+

DESCRIBE part;
+---------------+---------------+
| name          | type          |
+---------------+---------------+
| p_partkey     | bigint        |
| p_name        | string        |
| p_mfgr        | string        |
| p_brand       | string        |
| p_type        | string        |
| p_size        | int           |
| p_container   | string        |
| p_retailprice | decimal(12,2) |
| p_comment     | string        |
+---------------+---------------+

DESCRIBE region;
+-------------+-------------------------------------------------
------------------+
| name        | type
                  |
+-------------+-------------------------------------------------
--------------------+
| r_regionkey | smallint
                    |
| r_name      | string
                    |
| r_comment   | string
                    |
| r_nations   | array<struct<n_nationkey:smallint,n_name:string
,n_comment:string>> |
+-------------+-------------------------------------------------
----------------------+

DESCRIBE supplier;
+-------------+-----------------------------------------------+
| name        | type                                          |
+-------------+-----------------------------------------------+
| s_suppkey   | bigint                                        |
| s_name      | string                                        |
| s_address   | string                                        |
| s_nationkey | smallint                                      |
| s_phone     | string                                        |
| s_acctbal   | decimal(12,2)                                 |
| s_comment   | string                                        |
| s_partsupps | array<struct<ps_partkey:bigint,               |
|             | ps_availqty:int,ps_supplycost:decimal(12,2),  |
|             | ps_comment:string>>                           |
+-------------+-----------------------------------------------+
```

The volume of data used in the following examples is:

```
SELECT count(*) FROM customer;
+----------+
| count(*) |
+----------+
| 150000   |
+----------+
SELECT count(*) FROM part;
```

```
+----------+
| count(*) |
+----------+
| 200000   |
+----------+

SELECT count(*) FROM region;
+----------+
| count(*) |
+----------+
| 5        |
+----------+
SELECT count(*) FROM supplier;
+----------+
| count(*) |
+----------+
| 10000    |
+----------+
```

Constructing Parquet / ORC files with complex columns using Hive

The following examples demonstrate the Hive syntax to transform flat data (tables with all scalar columns) into Parquet/ORC tables where Impala can query the complex type columns. Each example shows the full sequence of steps, including switching back and forth between Impala and Hive. Although the source table can use any file format, the destination table must use the Parquet/ORC file format. We take Parquet in the following examples. You can replace Parquet with ORC to do the same things in ORC file format.

Create table with ARRAY in Impala, load data in Hive, query in Impala:

This example shows the cycle of creating the tables and querying the complex data in Impala, and using Hive (either the hive shell or beeline) for the data loading step. The data starts in flattened, denormalized form in a text table. Hive writes the corresponding Parquet data, including an ARRAY column. Then Impala can run analytic queries on the Parquet table, using join notation to unpack the ARRAY column.

```
/* Initial DDL and loading of flat, denormalized data happens in
 impala-shell */CREATE TABLE flat_array (country STRING, city ST
RING);INSERT INTO flat_array VALUES
    ('Canada', 'Toronto') , ('Canada', 'Vancouver') , ('Canada',
 "St. John\'s")
  , ('Canada', 'Saint John') , ('Canada', 'Montreal') , ('Canada
', 'Halifax')
  , ('Canada', 'Winnipeg') , ('Canada', 'Calgary') , ('Canada',
 'Saskatoon')
  , ('Canada', 'Ottawa') , ('Canada', 'Yellowknife') , ('France',
 'Paris')
  , ('France', 'Nice') , ('France', 'Marseilles') , ('France',
 'Cannes')
  , ('Greece', 'Athens') , ('Greece', 'Piraeus') , ('Greece', 'Ha
nia')
  , ('Greece', 'Heraklion') , ('Greece', 'Rethymnon') , ('Gree
ce', 'Fira');

CREATE TABLE complex_array (country STRING, city ARRAY <STRING>)
STORED AS PARQUET;
```

```
/* Conversion to Parquet and complex and/or nested columns happe
ns in Hive */
```

```
INSERT INTO complex_array SELECT country, collect_list(city) FROM
 flat_array GROUP BY country;
Query ID = dev_20151108160808_84477ff2-82bd-4ba4-9a77-554fa7b8
c0cb
Total jobs = 1
Launching Job 1 out of 1
...
```

```
/* Back to impala-shell again for analytic queries */

REFRESH complex_array;
SELECT country, city.item FROM complex_array, complex_array.city
+---------+-------------+
| country | item        |
+---------+-------------+
| Canada  | Toronto     |
| Canada  | Vancouver   |
| Canada  | St. John's  |
| Canada  | Saint John  |
| Canada  | Montreal    |
| Canada  | Halifax     |
| Canada  | Winnipeg    |
| Canada  | Calgary     |
| Canada  | Saskatoon   |
| Canada  | Ottawa      |
| Canada  | Yellowknife |
| France  | Paris       |
| France  | Nice        |
| France  | Marseilles  |
| France  | Cannes      |
| Greece  | Athens      |
| Greece  | Piraeus     |
| Greece  | Hania       |
| Greece  | Heraklion   |
| Greece  | Rethymnon   |
| Greece  | Fira        |
+---------+-------------+
```

Create table with STRUCT and ARRAY in Impala, load data in Hive, query in Impala:

This example shows the cycle of creating the tables and querying the complex data in Impala, and using Hive (either the hive shell or beeline) for the data loading step. The data starts in flattened, denormalized form in a text table. Hive writes the corresponding Parquet data, including a STRUCT column with an ARRAY field. Then Impala can run analytic queries on the Parquet table, using join notation to unpack the ARRAY field from the STRUCT column.

```
/* Initial DDL and loading of flat, denormalized data happens in
 impala-shell */

CREATE TABLE flat_struct_array (continent STRING, country STRI
NG, city STRING);
INSERT INTO flat_struct_array VALUES
    ('North America', 'Canada', 'Toronto') , ('North America', 'C
anada', 'Vancouver')
  , ('North America', 'Canada', "St. John\'s") , ('North Americ
a', 'Canada', 'Saint John')
  , ('North America', 'Canada', 'Montreal') , ('North America',
'Canada', 'Halifax')
  , ('North America', 'Canada', 'Winnipeg') , ('North America',
 'Canada', 'Calgary')
```

```
  , ('North America', 'Canada', 'Saskatoon') , ('North America',
 'Canada', 'Ottawa')
  , ('North America', 'Canada', 'Yellowknife') , ('Europe', 'Fran
ce', 'Paris')
  , ('Europe', 'France', 'Nice') , ('Europe', 'France', 'Marse
illes')
  , ('Europe', 'France', 'Cannes') , ('Europe', 'Greece', 'Athen
s')
  , ('Europe', 'Greece', 'Piraeus') , ('Europe', 'Greece', 'Hani
a')
  , ('Europe', 'Greece', 'Heraklion') , ('Europe', 'Greece', 'Re
thymnon')
  , ('Europe', 'Greece', 'Fira');
CREATE TABLE complex_struct_array (continent STRING, country ST
RUCT <name: STRING, city: ARRAY <STRING> >) STORED AS PARQUET;
```

```
/* Conversion to Parquet and complex and/or nested columns happe
ns in Hive */

INSERT INTO complex_struct_array SELECT continent, named_struct('
name', country, 'city', collect_list(city)) FROM flat_array_array
 GROUP BY continent, country;
Query ID = dev_20151108163535_11a4fa53-0003-4638-97e6-ef13cdb8e09
e
Total jobs = 1
Launching Job 1 out of 1
...
```

```
/* Back to impala-shell again for analytic queries */

REFRESH complex_struct_array;
SELECT t1.continent, t1.country.name, t2.item
  FROM complex_struct_array t1, t1.country.city t2
+---------------+--------------+-------------+
| continent     | country.name | item        |
+---------------+--------------+-------------+
| Europe        | France       | Paris       |
| Europe        | France       | Nice        |
| Europe        | France       | Marseilles  |
| Europe        | France       | Cannes      |
| Europe        | Greece       | Athens      |
| Europe        | Greece       | Piraeus     |
| Europe        | Greece       | Hania       |
| Europe        | Greece       | Heraklion   |
| Europe        | Greece       | Rethymnon   |
| Europe        | Greece       | Fira        |
| North America | Canada       | Toronto     |
| North America | Canada       | Vancouver   |
| North America | Canada       | St. John's  |
| North America | Canada       | Saint John  |
| North America | Canada       | Montreal    |
| North America | Canada       | Halifax     |
| North America | Canada       | Winnipeg    |
| North America | Canada       | Calgary     |
| North America | Canada       | Saskatoon   |
| North America | Canada       | Ottawa      |
| North America | Canada       | Yellowknife |
+---------------+--------------+-------------+
```

Flattening normalized tables into a single table with complex types

One common use for complex types is to embed the contents of one table into another. The traditional technique of denormalizing results in a huge number of rows with some column values repeated over and over. With complex types, you can keep the same number of rows as in the original normalized table, and put all the associated data from the other table in a single new column.

In this flattening scenario, you might frequently use a column that is an ARRAY consisting of STRUCT elements, where each field within the STRUCT corresponds to a column name from the table that you are combining.

The following example shows a traditional normalized layout using two tables, and then an equivalent layout using complex types in a single table.

```
/* Traditional relational design */
-- This table just stores numbers, allowing us to look up details
 about the employee
-- and details about their vacation time using a three-table j
oin query.
CREATE table employee_vacations
(
  employee_id BIGINT,
  vacation_id BIGINT
)
STORED AS PARQUET;

-- Each kind of information to track gets its own "fact table".
CREATE table vacation_details
(
  vacation_id BIGINT,
  vacation_start TIMESTAMP,
  duration INT
)
STORED AS PARQUET;
-- Any time we print a human-readable report, we join with this
 table to
-- display info about employee #1234.
CREATE TABLE employee_contact
(
  employee_id BIGINT,
  name STRING,
  address STRING,
  phone STRING,
  email STRING,
  address_type STRING /* 'home', 'work', 'remote', etc. */
)
STORED AS PARQUET;

/* Equivalent flattened schema using complex types */

-- For analytic queries using complex types, we can bundle the di
mension table
-- and multiple fact tables into a single table.
CREATE TABLE employee_vacations_nested_types
(
-- We might still use the employee_id for other join queries.
-- The table needs at least one scalar column to serve as an ide
ntifier
-- for the complex type columns.
  employee_id BIGINT,
```

```
-- Columns of the VACATION_DETAILS table are folded into a STRUC
T.
-- We drop the VACATION_ID column because Impala doesn't need
-- synthetic IDs to join a complex type column.
-- Each row from the VACATION_DETAILS table becomes an array el
ement.
  vacation ARRAY < STRUCT <
    vacation_start: TIMESTAMP,
    duration: INT
  >>,

-- The ADDRESS_TYPE column, with a small number of predefined val
ues that are distinct
-- for each employee, makes the EMPLOYEE_CONTACT table a good
candidate to turn into a MAP,
-- with each row represented as a STRUCT. The string value from
 ADDRESS_TYPE becomes the
-- "key" (the anonymous first field) of the MAP.
  contact MAP < STRING, STRUCT <
    address: STRING,
    phone: STRING,
    email: STRING
  >>
)
STORED AS PARQUET;
```

Interchanging complex type tables and data files with Hive and other components

You can produce Parquet data files through several Hadoop components and APIs.

If you have a Hive-created Parquet table that includes ARRAY, STRUCT, or MAP columns, Impala can query that same table in Impala 2.3 and higher, subject to the usual restriction that all other columns are of data types supported by Impala, and also that the file type of the table must be Parquet.

If you have a Parquet data file produced outside of Impala, Impala can automatically deduce the appropriate table structure using the syntax CREATE TABLE ... LIKE PARQUET '*hdfs_path_of_parquet_file*'. In Impala 2.3 and higher, this feature works for Parquet files that include ARRAY, STRUCT, or MAP types.

```
/* In impala-shell, find the HDFS data directory of the original
 table.
DESCRIBE FORMATTED tpch_nested_parquet.customer;
...
| Location: | hdfs://localhost:20500/test-warehouse/tpch_nested
_parquet.db/customer   | NULL |
...
# In the Unix shell, find the path of any Parquet data file in
that HDFS directory.
$ hdfs dfs -ls hdfs://localhost:20500/test-warehouse/tpch_neste
d_parquet.db/customer
Found 4 items
-rwxr-xr-x   3 dev supergroup  171298918 2015-09-22 23:30 hdfs://
localhost:20500/blah/tpch_nested_parquet.db/customer/000000_0
...

/* Back in impala-shell, use the HDFS path in a CREATE TABLE LIKE
 PARQUET statement. */
CREATE TABLE customer_ctlp
  LIKE PARQUET 'hdfs://localhost:20500/blah/tpch_nested_parquet.d
b/customer/000000_0'
  STORED AS PARQUET;
```

```
/* Confirm that old and new tables have the same column layout, i
ncluding complex types. */
DESCRIBE tpch_nested_parquet.customer
+--------------+-------------------------------------+---------+
| name         | type                                | comment |
+--------------+-------------------------------------+---------+
| c_custkey    | bigint                              |         |
| c_name       | string                              |         |
| c_address    | string                              |         |
| c_nationkey  | smallint                            |         |
| c_phone      | string                              |         |
| c_acctbal    | decimal(12,2)                       |         |
| c_mktsegment | string                              |         |
| c_comment    | string                              |         |
| c_orders     | array<struct<                       |         |
|              |   o_orderkey:bigint,                |         |
|              |   o_orderstatus:string,             |         |
|              |   o_totalprice:decimal(12,2),       |         |
|              |   o_orderdate:string,               |         |
|              |   o_orderpriority:string,           |         |
|              |   o_clerk:string,                   |         |
|              |   o_shippriority:int,               |         |
|              |   o_comment:string,                 |         |
|              |   o_lineitems:array<struct<         |         |
|              |     l_partkey:bigint,               |         |
|              |     l_suppkey:bigint,               |         |
|              |     l_linenumber:int,               |         |
|              |     l_quantity:decimal(12,2),       |         |
|              |     l_extendedprice:decimal(12,2),  |         |
|              |     l_discount:decimal(12,2),       |         |
|              |     l_tax:decimal(12,2),            |         |
|              |     l_returnflag:string,            |         |
|              |     l_linestatus:string,            |         |
|              |     l_shipdate:string,              |         |
|              |     l_commitdate:string,            |         |
|              |     l_receiptdate:string,           |         |
|              |     l_shipinstruct:string,          |         |
|              |     l_shipmode:string,              |         |
|              |     l_comment:string                |         |
|              |   >>                                |         |
|              | >>                                  |         |
+--------------+-------------------------------------+---------+

describe customer_ctlp;
+--------------+-------------------------------------+---------
--------------------+
| name         | type                                | comment
                     |
+--------------+-------------------------------------+---------
--------------------+
| c_custkey    | bigint                              | Inferred
 from Parquet file. |
| c_name       | string                              | Inferred
 from Parquet file. |
| c_address    | string                              | Inferred
 from Parquet file. |
| c_nationkey  | int                                 | Inferred
 from Parquet file. |
| c_phone      | string                              | Inferred
 from Parquet file. |
| c_acctbal    | decimal(12,2)                       | Inferred
 from Parquet file. |
```

```
| c_mktsegment | string                                  | Inferred
 from Parquet file. |
| c_comment    | string                                  | Inferred
 from Parquet file. |
| c_orders     | array<struct<                           | Inferred
 from Parquet file. |
|              |   o_orderkey:bigint,                    |
                |
|              |   o_orderstatus:string,                 |
                |
|              |   o_totalprice:decimal(12,2),           |
                |
|              |   o_orderdate:string,                   |
                |
|              |   o_orderpriority:string,               |
                |
|              |   o_clerk:string,                       |
                |
|              |   o_shippriority:int,                   |
                |
|              |   o_comment:string,                     |
                |
|              |   o_lineitems:array<struct<             |
                |
|              |     l_partkey:bigint,                   |
                |
|              |     l_suppkey:bigint,                   |
                |
|              |     l_linenumber:int,                   |
                |
|              |     l_quantity:decimal(12,2),           |
                |
|              |     l_extendedprice:decimal(12,2),      |
                |
|              |     l_discount:decimal(12,2),           |
                |
|              |     l_tax:decimal(12,2),                |
                |
|              |     l_returnflag:string,                |
                |
|              |     l_linestatus:string,                |
                |
|              |     l_shipdate:string,                  |
                |
|              |     l_commitdate:string,                |
                |
|              |     l_receiptdate:string,               |
                |
|              |     l_shipinstruct:string,              |
                |
|              |     l_shipmode:string,                  |
                |
|              |     l_comment:string                    |
                |
|              |   >>                                    |
                |
|              | >>                                      |
                |
+--------------+-----------------------------------------+---------
--------------------+
```

Related Information

Using Parquet Data Files
Scalability considerations

Impala SQL literals

Each of the Impala data types has corresponding notation for literal values of that type. You specify literal values in SQL statements, such as in the SELECT list or WHERE clause of a query, or as an argument to a function call.

Numeric literals

To write literals for the integer types (TINYINT, SMALLINT, INT, and BIGINT), use a sequence of digits with optional leading zeros.

To write literals for the floating-point types (DECIMAL, FLOAT, and DOUBLE), use a sequence of digits with an optional decimal point (. character). To preserve accuracy during arithmetic expressions, Impala interprets floating-point literals as the DECIMAL type with the smallest appropriate precision and scale, until required by the context to convert the result to FLOAT or DOUBLE.

Integer values are promoted to floating-point when necessary, based on the context.

You can also use exponential notation by including an e character. For example, 1e6 is 1 times 10 to the power of 6 (1 million). A number in exponential notation is always interpreted as floating-point.

When Impala encounters a numeric literal, it considers the type to be the “smallest” that can accurately represent the value. The type is promoted to larger or more accurate types if necessary, based on subsequent parts of an expression.

STRING literals

String literals are quoted using either single or double quotation marks. You can use either kind of quotes for string literals, even both kinds for different literals within the same statement.

Quoted literals are considered to be of type STRING. To use quoted literals in contexts requiring a CHAR or VARC HAR value, CAST() the literal to a CHAR or VARCHAR of the appropriate length.

Escaping special characters:

To encode special characters within a string literal, precede them with the backslash (\) escape character:

- \t represents a tab.
- \n represents a newline or linefeed. This might cause extra line breaks in `impala-shell` output.
- \r represents a carriage return. This might cause unusual formatting (making it appear that some content is overwritten) in `impala-shell` output.
- \b represents a backspace. This might cause unusual formatting (making it appear that some content is overwritten) in `impala-shell` output.
- \0 represents an ASCII nul character (not the same as a SQL NULL). This might not be visible in `impala-shell` output.
- \Z represents a DOS end-of-file character. This might not be visible in `impala-shell` output.
- \% and _ can be used to escape wildcard characters within the string passed to the LIKE operator.
- \ followed by 3 octal digits represents the ASCII code of a single character; for example, \101 is ASCII 65, the character A.
- Use two consecutive backslashes (\\) to prevent the backslash from being interpreted as an escape character.
- Use the backslash to escape single or double quotation mark characters within a string literal, if the literal is enclosed by the same type of quotation mark.
- If the character following the \ does not represent the start of a recognized escape sequence, the character is passed through unchanged.

Quotes within quotes:

To include a single quotation character within a string value, enclose the literal with either single or double quotation marks, and optionally escape the single quote as a \' sequence. Earlier releases required escaping a single quote inside double quotes. Continue using escape sequences in this case if you also need to run your SQL code on older versions of Impala.

To include a double quotation character within a string value, enclose the literal with single quotation marks, no escaping is necessary in this case. Or, enclose the literal with double quotation marks and escape the double quote as a \" sequence.

```
[localhost:21000] > select "What\'s happening?" as single_within_double,
                  >        'I\'m not sure.' as single_within_single,
                  >        "Homer wrote \"The Iliad\"." as double_within_d
ouble,
                  >        'Homer also wrote "The Odyssey".' as double_wi
thin_single;
+----------------------+----------------------+--------------------------+-
-------------------------------+
| single_within_double | single_within_single | double_within_double     |
 double_within_single            |
+----------------------+----------------------+--------------------------
+--------------------------------+
| What's happening?    | I'm not sure.        | Homer wrote "The Iliad". | H
omer also wrote "The Odyssey". |
+----------------------+----------------------+--------------------------+-
-------------------------------+
```

Field terminator character in CREATE TABLE:

Note: The CREATE TABLE clauses FIELDS TERMINATED BY, ESCAPED BY, and LINES TERMINATED BY have special rules for the string literal used for their argument, because they all require a single character. You can use a regular character surrounded by single or double quotation marks, an octal sequence such as '\054' (representing a comma), or an integer in the range '-127'..'128' (with quotation marks but no backslash), which is interpreted as a single-byte ASCII character. Negative values are subtracted from 256; for example, FIELDS TERMINATED BY '-2' sets the field delimiter to ASCII code 254, the “Icelandic Thorn” character used as a delimiter by some data formats.

impala-shell considerations:

When dealing with output that includes non-ASCII or non-printable characters such as linefeeds and backspaces, use the `impala-shell` options to save to a file, turn off pretty printing, or both rather than relying on how the output appears visually.

BOOLEAN literals

For BOOLEAN values, the literals are TRUE and FALSE, with no quotation marks and case-insensitive.

TIMESTAMP literals

Impala automatically converts STRING literals of the correct format into TIMESTAMP values. Timestamp values are accepted in the format 'yyyy#MM#dd HH:mm:ss.SSSSSS', and can consist of just the date, or just the time, with or without the fractional second portion. For example, you can specify TIMESTAMP values such as '1966#07#30', '08:30:00', or '1985#09#25 17:45:30.005'.

Leading zeroes are not required in the numbers representing the date component, such as month and date, or the time component, such as hour, minute, and second. For example, Impala accepts both '2018#1#1 01:02:03' and '2018-01-01 1:2:3' as valid.

In STRING to TIMESTAMP conversions, leading and trailing white spaces, such as a space, a tab, a newline, or a carriage return, are ignored. For example, Impala treats the following as equivalent: '1999#12#01 01:02:03 ', ' 1999#12#01 01:02:03', '1999#12#01 01:02:03\r\n\t'.

When you convert or cast a STRING literal to TIMESTAMP, you can use the following separators between the date part and the time part:

- One or more space characters

  Example: CAST('2001-01-09 01:05:01' AS TIMESTAMP)
- The character “T”

  Example: CAST('2001-01-09T01:05:01' AS TIMESTAMP)

You can also use INTERVAL expressions to add or subtract from timestamp literal values, such as CAST('1966#0 7#30' AS TIMESTAMP) + INTERVAL 5 YEARS + INTERVAL 3 DAYS.

Depending on your data pipeline, you might receive date and time data as text, in notation that does not exactly match the format for Impala TIMESTAMP literals. See *Impala Date and Time Functions* for functions that can convert between a variety of string literals (including different field order, separators, and timezone notation) and equivalent TIMESTAMP or numeric values.

DATE literals

The DATE literals are in the form of DATE'YYYY-MM-DD'. For example, DATE '2013-01-01'

NULL

The notion of NULL values is familiar from all kinds of database systems, but each SQL dialect can have its own behavior and restrictions on NULL values. For Big Data processing, the precise semantics of NULL values are significant: any misunderstanding could lead to inaccurate results or misformatted data, that could be time-consuming to correct for large data sets.

- NULL is a different value than an empty string. The empty string is represented by a string literal with nothing inside, "" or ''.
- In a delimited text file, the NULL value is represented by the special token \N.
- When Impala inserts data into a partitioned table, and the value of one of the partitioning columns is NULL or the empty string, the data is placed in a special partition that holds only these two kinds of values. When these values are returned in a query, the result is NULL whether the value was originally NULL or an empty string. This behavior is compatible with the way Hive treats NULL values in partitioned tables. Hive does not allow empty strings as partition keys, and it returns a string value such as __HIVE_DEFAULT_PARTITION__ instead of NULL when such values are returned from a query. For example:

  ```
  create table t1 (i int) partitioned by (x int, y string);
  -- Select an INT column from another table, with all rows going into a spe
  cial HDFS subdirectory
  -- named __HIVE_DEFAULT_PARTITION__. Depending on whether one or both of
  the partitioning keys
  -- are null, this special directory name occurs at different levels of the
   physical data directory
  -- for the table.
  insert into t1 partition(x=NULL, y=NULL) select c1 from some_other_table;
  insert into t1 partition(x, y=NULL) select c1, c2 from some_other_table;
  insert into t1 partition(x=NULL, y) select c1, c3  from some_other_table;
  ```
- There is no NOT NULL clause when defining a column to prevent NULL values in that column.
- There is no DEFAULT clause to specify a non-NULL default value.
- If an INSERT operation mentions some columns but not others, the unmentioned columns contain NULL for all inserted rows.
- In Impala 1.2.1 and higher, all NULL values come at the end of the result set for ORDER BY ... ASC queries, and at the beginning of the result set for ORDER BY ... DESC queries. In effect, NULL is considered greater than all other values for sorting purposes. The original Impala behavior always put NULL values at the end, even for ORDER BY ... DESC queries. The new behavior in Impala 1.2.1 makes Impala more compatible with other

popular database systems. In Impala 1.2.1 and higher, you can override or specify the sorting behavior for NULL by adding the clause NULLS FIRST or NULLS LAST at the end of the ORDER BY clause.

Note: Because the NULLS FIRST and NULLS LAST keywords are not currently available in Hive queries, any views you create using those keywords will not be available through Hive.

- In all other contexts besides sorting with ORDER BY, comparing a NULL to anything else returns NULL, making the comparison meaningless. For example, 10 > NULL produces NULL, 10 < NULL also produces NULL, 5 BE TWEEN 1 AND NULL produces NULL, and so on.

Several built-in functions serve as shorthand for evaluating expressions and returning NULL, 0, or some other substitution value depending on the expression result: ifnull(), isnull(), nvl(), nullif(), nullifzero(), and zeroifnull().

Kudu considerations:

Columns in Kudu tables have an attribute that specifies whether or not they can contain NULL values. A column with a NULL attribute can contain nulls. A column with a NOT NULL attribute cannot contain any nulls, and an INSE RT, UPDATE, or UPSERT statement will skip any row that attempts to store a null in a column designated as NOT NULL. Kudu tables default to the NULL setting for each column, except columns that are part of the primary key.

In addition to columns with the NOT NULL attribute, Kudu tables also have restrictions on NULL values in columns that are part of the primary key for a table. No column that is part of the primary key in a Kudu table can contain any NULL values.

Related Information

Impala SQL data types

Impala Shell configuration options

Impala SQL operators

SQL operators are used primarily in the WHERE clause to perform operations, such as comparison operations and arithmetic operations.

Arithmetic operators

The arithmetic operators use expressions with a left-hand argument, the operator, and then (in most cases) a right-hand argument.

Syntax:

```
left_hand_arg binary_operator right_hand_arg
unary_operator single_arg
```

- + and -: Can be used either as unary or binary operators.
  - With unary notation, such as +5, -2.5, or -*col_name*, they multiply their single numeric argument by +1 or -1. Therefore, unary + returns its argument unchanged, while unary - flips the sign of its argument. Although you can double up these operators in expressions such as ++5 (always positive) or -+2 or +-2 (both always negative), you cannot double the unary minus operator because -- is interpreted as the start of a comment. (You can use a double unary minus operator if you separate the - characters, for example with a space or parentheses.)
  - With binary notation, such as 2+2, 5-2.5, or *col1* + *col2*, they add or subtract respectively the right-hand argument to (or from) the left-hand argument. Both arguments must be of numeric types.
- * and /: Multiplication and division respectively. Both arguments must be of numeric types.

  When multiplying, the shorter argument is promoted if necessary (such as SMALLINT to INT or BIGINT, or FLOAT to DOUBLE), and then the result is promoted again to the next larger type. Thus, multiplying a TINY INT and an INT produces a BIGINT result. Multiplying a FLOAT and a FLOAT produces a DOUBLE result.

Multiplying a FLOAT and a DOUBLE or a DOUBLE and a DOUBLE produces a DECIMAL(38,17), because DECIMAL values can represent much larger and more precise values than DOUBLE.

When dividing, Impala always treats the arguments and result as DOUBLE values to avoid losing precision. If you need to insert the results of a division operation into a FLOAT column, use the CAST() function to convert the result to the correct type.

- DIV: Integer division. Arguments are not promoted to a floating-point type, and any fractional result is discarded. For example, 13 DIV 7 returns 1, 14 DIV 7 returns 2, and 15 DIV 7 returns 2. This operator is the same as the QUOTIENT() function.
- %: Modulo operator. Returns the remainder of the left-hand argument divided by the right-hand argument. Both arguments must be of one of the integer types.
- &, |, ~, and ^: Bitwise operators that return the logical AND, logical OR, NOT, or logical XOR (exclusive OR) of their argument values. Both arguments must be of one of the integer types. If the arguments are of different type, the argument with the smaller type is implicitly extended to match the argument with the longer type.

You can chain a sequence of arithmetic expressions, optionally grouping them with parentheses.

The arithmetic operators generally do not have equivalent calling conventions using functional notation. For example, prior to Impala 2.2, there is no MOD() function equivalent to the % modulo operator. Conversely, there are some arithmetic functions that do not have a corresponding operator. For example, for exponentiation you use the POW() function, but there is no ** exponentiation operator. See *Impala Mathematical Functions* for the arithmetic functions you can use.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex types on page 49 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM() and AVG() are computed using the numeric R_NATIONKEY field, and the general-purpose MAX() and MIN() values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
```

```
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct
_vals |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5
    |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5
    |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5
    |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5
    |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5
    |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
```

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

The following example shows how to do an arithmetic operation using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used in an arithmetic expression, such as multiplying by 10:

```
-- The SMALLINT is a field within an array of structs.
```

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey * 10
  from region, region.r_nations as nation
where nation.item.n_nationkey < 5;
+-------------+-------------+-------------------------------+
| r_name      | item.n_name | nation.item.n_nationkey * 10 |
+-------------+-------------+-------------------------------+
| AMERICA     | CANADA      | 30                            |
| AMERICA     | BRAZIL      | 20                            |
| AMERICA     | ARGENTINA   | 10                            |
| MIDDLE EAST | EGYPT       | 40                            |
| AFRICA      | ALGERIA     | 0                             |
+-------------+-------------+-------------------------------+
```

BETWEEN operator

In a WHERE clause, compares an expression to both a lower and upper bound. The comparison is successful is the expression is greater than or equal to the lower bound, and less than or equal to the upper bound. If the bound values are switched, so the lower bound is greater than the upper bound, does not match any values.

Syntax:

```
expression BETWEEN lower_bound AND upper_bound
```

Data types: Typically used with numeric data types. Works with any data type, although not very practical for BOOL EAN values. (BETWEEN false AND true will match all BOOLEAN values.) Use CAST() if necessary to ensure the lower and upper bound values are compatible types. Call string or date/time functions if necessary to extract or transform the relevant portion to compare, especially if the value can be transformed into a number.

Usage notes:

Be careful when using short string operands. A longer string that starts with the upper bound value will not be included, because it is considered greater than the upper bound. For example, BETWEEN 'A' and 'M' would not match the string value 'Midway'. Use functions such as upper(), lower(), substr(), trim(), and so on if necessary to ensure the comparison works as expected.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

The following example shows how to do a BETWEEN comparison using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used in a comparison operator:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey
from region, region.r_nations as nation
where nation.item.n_nationkey between 3 and 5
+-------------+-------------+------------------+
| r_name      | item.n_name | item.n_nationkey |
+-------------+-------------+------------------+
| AMERICA     | CANADA      | 3                |
| MIDDLE EAST | EGYPT       | 4                |
| AFRICA      | ETHIOPIA    | 5                |
+-------------+-------------+------------------+
```

Comparison operators

Impala supports the familiar comparison operators for checking equality and sort order for the column data types:

Syntax:

```
left_hand_expression comparison_operator right_hand_expression
```

- =, !=, <>: apply to all scalar types.
- <, <=, >, >=: apply to all scalar types; for BOOLEAN, TRUE is considered greater than FALSE.

Alternatives:

The IN and BETWEEN operators provide shorthand notation for expressing combinations of equality, less than, and greater than comparisons with a single operator.

Because comparing any value to NULL produces NULL rather than TRUE or FALSE, use the IS NULL and IS NOT NULL operators to check if a value is NULL or not.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

The following example shows how to do an arithmetic operation using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used with a comparison operator such as <:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey
from region, region.r_nations as nation
where nation.item.n_nationkey < 5
+-------------+-------------+------------------+
| r_name      | item.n_name | item.n_nationkey |
+-------------+-------------+------------------+
| AMERICA     | CANADA      | 3                |
| AMERICA     | BRAZIL      | 2                |
| AMERICA     | ARGENTINA   | 1                |
| MIDDLE EAST | EGYPT       | 4                |
| AFRICA      | ALGERIA     | 0                |
+-------------+-------------+------------------+
```

EXISTS operator

The EXISTS operator tests whether a subquery returns any results. You typically use it to find values from one table that have corresponding values in another table.

The converse, NOT EXISTS, helps to find all the values from one table that do not have any corresponding values in another table.

Syntax:

```
EXISTS (subquery)
NOT EXISTS (subquery)
```

Usage notes:

The subquery can refer to a different table than the outer query block, or the same table. For example, you might use EXISTS or NOT EXISTS to check the existence of parent/child relationships between two columns of the same table.

You can also use operators and function calls within the subquery to test for other kinds of relationships other than strict equality. For example, you might use a call to COUNT() in the subquery to check whether the number of matching values is higher or lower than some limit. You might call a UDF in the subquery to check whether values in one table matches a hashed representation of those same values in a different table.

NULL considerations:

If the subquery returns any value at all (even NULL), EXISTS returns TRUE and NOT EXISTS returns false.

Restrictions:

Correlated subqueries used in EXISTS and IN operators cannot include a LIMIT clause.

Prior to Impala 2.6, the NOT EXISTS operator required a correlated subquery. In Impala 2.6 and higher, NOT EXIS TS works with uncorrelated queries also.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

ILIKE operator

A case-insensitive comparison operator for STRING data, with basic wildcard capability using _ to match a single character and % to match multiple characters. The argument expression must match the entire string value. Typically, it is more efficient to put any % wildcard match at the end of the string.

This operator, available in Impala 2.5 and higher, is the equivalent of the LIKE operator, but with case-insensitive comparisons.

Syntax:

```
string_expression ILIKE wildcard_expression
string_expression NOT ILIKE wildcard_expression
```

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

IN operator

The IN operator compares an argument value to a set of values, and returns TRUE if the argument matches any value in the set. The NOT IN operator reverses the comparison, and checks if the argument value is not part of a set of values.

Syntax:

```
expression IN (expression [, expression])
expression IN (subquery)

expression NOT IN (expression [, expression])
expression NOT IN (subquery)
```

The left-hand expression and the set of comparison values must be of compatible types.

The left-hand expression must consist only of a single value, not a tuple. Although the left-hand expression is typically a column name, it could also be some other value. For example, the WHERE clauses WHERE id IN (5) and WHERE 5 IN (id) produce the same results.

The set of values to check against can be specified as constants, function calls, column names, or other expressions in the query text. The maximum number of expressions in the IN list is 9999. (The maximum number of elements of a single expression is 10,000 items, and the IN operator itself counts as one.)

In Impala 2.0 and higher, the set of values can also be generated by a subquery. IN can evaluate an unlimited number of results using a subquery.

Usage notes:

Any expression using the IN operator could be rewritten as a series of equality tests connected with OR, but the IN syntax is often clearer, more concise, and easier for Impala to optimize. For example, with partitioned tables, queries frequently use IN clauses to filter data by comparing the partition key columns to specific values.

NULL considerations:

If there really is a matching non-null value, IN returns TRUE:

If the searched value is not found in the comparison values, and the comparison values include NULL, the result is NULL:

If the left-hand argument is NULL, IN always returns NULL. This rule applies even if the comparison values include NULL.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Restrictions:

Correlated subqueries used in EXISTS and IN operators cannot include a LIMIT clause.

IREGEXP operator

Tests whether a value matches a regular expression, using case-insensitive string comparisons. Uses the POSIX regular expression syntax where ^ and $ match the beginning and end of the string, . represents any single character, * represents a sequence of zero or more items, + represents a sequence of one or more items, ? produces a non-greedy match, and so on.

This operator, available in Impala 2.5 and higher, is the equivalent of the REGEXP operator, but with case-insensitive comparisons.

Syntax:

```
string_expression IREGEXP regular_expression
```

Usage notes:

The | symbol is the alternation operator, typically used within () to match different sequences. The () groups do not allow backreferences. To retrieve the part of a value matched within a () section, use the regexp_extract() built-in function. (Currently, there is not any case-insensitive equivalent for the regexp_extract() function.)

In Impala 1.3.1 and higher, the REGEXP and RLIKE operators now match a regular expression string that occurs anywhere inside the target string, the same as if the regular expression was enclosed on each side by .*. Previously, these operators only succeeded when the regular expression matched the entire target string. This change improves compatibility with the regular expression support for popular database systems. There is no change to the behavior of the regexp_extract() and regexp_replace() built-in functions.

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the *RE2 documentation*. It has most idioms familiar from regular expressions in Perl, Python, and so on, including .*? for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of

an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

IS DISTINCT FROM operator

The IS DISTINCT FROM operator, and its converse the IS NOT DISTINCT FROM operator, test whether or not values are identical. IS NOT DISTINCT FROM is similar to the = operator, and IS DISTINCT FROM is similar to the != operator, except that NULL values are treated as identical. Therefore, IS NOT DISTINCT FROM returns true rather than NULL, and IS DISTINCT FROM returns false rather than NULL, when comparing two NULL values. If one of the values being compared is NULL and the other is not, IS DISTINCT FROM returns true and IS NOT DISTINCT FROM returns false, again instead of returning NULL in both cases.

Syntax:

```
expression1 IS DISTINCT FROM expression2

expression1 IS NOT DISTINCT FROM expression2
expression1 <=> expression2
```

The operator <=> is an alias for IS NOT DISTINCT FROM. It is typically used as a NULL-safe equality operator in join queries. That is, A <=> B is true if A equals B or if both A and B are NULL.

Usage notes:

This operator provides concise notation for comparing two values and always producing a true or false result, without treating NULL as a special case. Otherwise, to unambiguously distinguish between two values requires a compound expression involving IS [NOT] NULL tests of both operands in addition to the = or != operator.

The <=> operator, used like an equality operator in a join query, is more efficient than the equivalent clause: IF (A IS NULL OR B IS NULL, A IS NULL AND B IS NULL, A = B). The <=> operator can use a hash join, while the IF expression cannot.

IS NULL operator

The IS NULL operator, and its converse the IS NOT NULL operator, test whether a specified value is NULL. Because using NULL with any of the other comparison operators such as = or != also returns NULL rather than TRUE or FALSE, you use a special-purpose comparison operator to check for this special condition.

In Impala 2.1.1 and higher, you can use the operators IS UNKNOWN and IS NOT UNKNOWN as synonyms for IS NULL and IS NOT NULL, respectively.

Syntax:

```
expression IS NULL
expression IS NOT NULL

expression IS UNKNOWN
expression IS NOT UNKNOWN
```

Usage notes:

In many cases, NULL values indicate some incorrect or incomplete processing during data ingestion or conversion. You might check whether any values in a column are NULL, and if so take some followup action to fill them in.

With sparse data, often represented in “wide” tables, it is common for most values to be NULL with only an occasional non-NULL value. In those cases, you can use the IS NOT NULL operator to identify the rows containing any data at all for a particular column, regardless of the actual value.

With a well-designed database schema, effective use of NULL values and IS NULL and IS NOT NULL operators can save having to design custom logic around special values such as 0, -1, 'N/A', empty string, and so on. NULL lets you distinguish between a value that is known to be 0, false, or empty, and a truly unknown value.

Complex type considerations:

The IS [NOT] UNKNOWN operator, as with the IS [NOT] NULL operator, is not applicable to complex type columns (STRUCT, ARRAY, or MAP). Using a complex type column with this operator causes a query error.

IS TRUE operator

This variation of the IS operator tests for truth or falsity, with right-hand arguments [NOT] TRUE, [NOT] FALSE, and [NOT] UNKNOWN.

Syntax:

```
expression IS TRUE
expression IS NOT TRUE

expression IS FALSE
expression IS NOT FALSE
```

Usage notes:

This IS TRUE and IS FALSE forms are similar to doing equality comparisons with the Boolean values TRUE and FALSE, except that IS TRUE and IS FALSE always return either TRUE or FALSE, even if the left-hand side expression returns NULL

These operators let you simplify Boolean comparisons that must also check for NULL, for example X != 10 AND X IS NOT NULL is equivalent to (X != 10) IS TRUE.

In Impala 2.1.1 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE(), ISNOTTRUE(), ISFALSE(), and ISNOTFALSE().

Complex type considerations:

The IS [NOT] TRUE and IS [NOT] FALSE operators are not applicable to complex type columns (STRUCT, ARRAY, or MAP). Using a complex type column with these operators causes a query error.

LIKE operator

A comparison operator for STRING data, with basic wildcard capability using the underscore (_) to match a single character and the percent sign (%) to match multiple characters. The argument expression must match the entire string value. Typically, it is more efficient to put any % wildcard match at the end of the string.

Syntax:

```
string_expression LIKE wildcard_expression
string_expression NOT LIKE wildcard_expression
```

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Examples:

```
select distinct c_last_name from customer where c_last_name like 'Mc%' or c_
last_name like 'Mac%';
select count(c_last_name) from customer where c_last_name like 'M%';
select c_email_address from customer where c_email_address like '%.edu';

-- We can find 4-letter names beginning with 'M' by calling functions...
select distinct c_last_name from customer where length(c_last_name) = 4 and
 substr(c_last_name,1,1) = 'M';
```

```
-- ...or in a more readable way by matching M followed by exactly 3 charac
ters.
select distinct c_last_name from customer where c_last_name like 'M___';
```

For case-insensitive comparisons, see the ILIKE operator. For a more general kind of search operator using regular expressions, see the REGEXP operator or its case-insensitive counterpart the IREGEXP operator.

Logical operators

Logical operators return a BOOLEAN value, based on a binary or unary logical operation between arguments that are also BOOLEAN values. Typically, the argument expressions use comparison operators.

Syntax:

```
boolean_expression binary_logical_operator boolean_expression
unary_logical_operator boolean_expression
```

The Impala logical operators are:

- AND: A binary operator that returns true if its left-hand and right-hand arguments both evaluate to true, NULL if either argument is NULL, and false otherwise.
- OR: A binary operator that returns true if either of its left-hand and right-hand arguments evaluate to true, NULL if one argument is NULL and the other is either NULL or false, and false otherwise.
- NOT: A unary operator that flips the state of a Boolean expression from true to false, or false to true. If the argument expression is NULL, the result remains NULL. (When NOT is used this way as a unary logical operator, it works differently than the IS NOT NULL comparison operator, which returns true when applied to a NULL.)

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

The following example shows how to do an arithmetic operation using a numeric field of a STRUCT type that is an item within an ARRAY column. Once the scalar numeric value R_NATIONKEY is extracted, it can be used in an arithmetic expression, such as multiplying by 10:

```
-- The SMALLINT is a field within an array of structs.
describe region;
+-------------+--------------------------+---------+
| name        | type                     | comment |
+-------------+--------------------------+---------+
| r_regionkey | smallint                 |         |
| r_name      | string                   |         |
| r_comment   | string                   |         |
| r_nations   | array<struct<            |         |
|             |   n_nationkey:smallint,  |         |
|             |   n_name:string,         |         |
|             |   n_comment:string       |         |
|             | >>                       |         |
+-------------+--------------------------+---------+

-- When we refer to the scalar value using dot notation,
-- we can use arithmetic and comparison operators on it
-- like any other number.
select r_name, nation.item.n_name, nation.item.n_nationkey
  from region, region.r_nations as nation
where
  nation.item.n_nationkey between 3 and 5
  or nation.item.n_nationkey < 15;
```

```
+-------------+----------------+------------------+
| r_name      | item.n_name    | item.n_nationkey |
+-------------+----------------+------------------+
| EUROPE      | UNITED KINGDOM | 23               |
| EUROPE      | RUSSIA         | 22               |
| EUROPE      | ROMANIA        | 19               |
| ASIA        | VIETNAM        | 21               |
| ASIA        | CHINA          | 18               |
| AMERICA     | UNITED STATES  | 24               |
| AMERICA     | PERU           | 17               |
| AMERICA     | CANADA         | 3                |
| MIDDLE EAST | SAUDI ARABIA   | 20               |
| MIDDLE EAST | EGYPT          | 4                |
| AFRICA      | MOZAMBIQUE     | 16               |
| AFRICA      | ETHIOPIA       | 5                |
+-------------+----------------+------------------+
```

REGEXP operator

Tests whether a value matches a regular expression. Uses the POSIX regular expression syntax where ^ and $ match the beginning and end of the string, . represents any single character, * represents a sequence of zero or more items, + represents a sequence of one or more items, ? produces a non-greedy match, and so on.

Syntax:

```
string_expression REGEXP regular_expression
```

Usage notes:

The RLIKE operator is a synonym for REGEXP.

The | symbol is the alternation operator, typically used within () to match different sequences. The () groups do not allow backreferences. To retrieve the part of a value matched within a () section, use the regexp_extract() built-in function.

In Impala 1.3.1 and higher, the REGEXP and RLIKE operators now match a regular expression string that occurs anywhere inside the target string, the same as if the regular expression was enclosed on each side by .*. Previously, these operators only succeeded when the regular expression matched the entire target string. This change improves compatibility with the regular expression support for popular database systems. There is no change to the behavior of the regexp_extract() and regexp_replace() built-in functions.

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the *RE2 documentation*. It has most idioms familiar from regular expressions in Perl, Python, and so on, including .*? for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Complex type considerations:

You cannot refer to a column with a complex data type (ARRAY, STRUCT, or MAP) directly in an operator. You can apply operators only to scalar values that make up a complex type (the fields of a STRUCT, the items of an ARRAY, or the key or value portion of a MAP) as part of a join query that refers to the scalar value using the appropriate dot notation or ITEM, KEY, or VALUE pseudocolumn names.

Example:

```
- Find all customers whose first name starts with 'J', followed by 0 or more
 of any character.
select c_first_name, c_last_name from customer where c_first_name regexp
'^J.*';
-- Match multiple character sequences, either 'Mac' or 'Mc'.
```

```
select c_first_name, c_last_name from customer where c_last_name regexp '^
(Mac|Mc)donald$';
```

RLIKE operator

It is a synonym for the REGEXP operator.

Related Information

Impala mathematical functions
RE2 documentation on GitHub

Impala SQL comments

Impala supports the syntax of standard SQL comments.

- All text from a -- sequence to the end of the line is considered a comment and ignored. This type of comment can occur on a single line by itself, or after all or part of a statement.
- All text from a /* sequence to the next */ sequence is considered a comment and ignored. This type of comment can stretch over multiple lines. This type of comment can occur on one or more lines by itself, in the middle of a statement, or before or after a statement.

Impala SQL statements

The Impala SQL dialect supports a range of standard elements, plus some extensions for Big Data use cases related to data loading and data warehousing.

Note:

In the `impala-shell` interpreter, a semicolon at the end of each statement is required. Since the semicolon is not actually part of the SQL syntax, we do not include it in the syntax definition of each statement, but we do show it in examples intended to be run in `impala-shell`.

The following sections show the major SQL statements that you work with in Impala:

DDL statements

DDL refers to “Data Definition Language”, 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. Most Impala DDL statements start with the keywords CREATE, DROP, or ALTER.

The Impala DDL statements are:

- ALTER TABLE statement on page 99
- ALTER VIEW statement on page 114
- COMPUTE STATS statement on page 115
- CREATE DATABASE statement on page 123
- CREATE FUNCTION statement on page 125
- CREATE TABLE statement on page 131
- CREATE VIEW statement on page 145
- DROP DATABASE statement on page 161
- DROP FUNCTION statement on page 163
- DROP TABLE statement on page 169
- DROP VIEW statement on page 171
- GRANT statement on page 174
- REVOKE statement on page 193

After Impala executes a DDL command, information about available tables, columns, views, partitions, and so on is automatically synchronized between all the Impala nodes in a cluster. (Prior to Impala 1.2, you had to issue a REFR ESH or an INVALIDATE METADATA statement manually on the other nodes to make them aware of the changes.)

If the timing of metadata updates is significant, for example if you use round-robin scheduling where each query could be issued through a different Impala node, you can enable the SYNC_DDL query option to make the DDL statement wait until all nodes have been notified about the metadata changes.

See *Using Impala with the Amazon S3 Filesystem* for details about how Impala DDL statements interact with tables and partitions stored in the Amazon S3 filesystem.

Although the INSERT statement is officially classified as a DML (data manipulation language) statement, it also involves metadata changes that must be broadcast to all Impala nodes, and so is also affected by the SYNC_DDL query option.

Because the SYNC_DDL query option makes each DDL operation take longer than normal, you might only enable it before the last DDL operation in a sequence. For example, if you are running a script that issues multiple of DDL operations to set up an entire new schema, add several new partitions, and so on, you might minimize the performance overhead by enabling the query option only before the last CREATE, DROP, ALTER, or INSERT statement. The script only finishes when all the relevant metadata changes are recognized by all the Impala nodes, so you could connect to any node and issue queries through it.

The classification of DDL, DML, and other statements is not necessarily the same between Impala and Hive. Impala organizes these statements in a way intended to be familiar to people familiar with relational databases or data warehouse products. Statements that modify the metastore database, such as COMPUTE STATS, are classified as DDL. Statements that only query the metastore database, such as SHOW or DESCRIBE, are put into a separate category of utility statements.

Note: The query types shown in the Impala debug web user interface might not match exactly the categories listed here. For example, currently the USE statement is shown as DDL in the debug web UI. The query types shown in the debug web UI are subject to change, for improved consistency.

DML statements

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.

Impala supports the following DML statements:

- DELETE: Works for Kudu tables only
- INSERT
- LOAD DATA: Does not apply for HBase or Kudu tables
- UPDATE: Works for Kudu tables only
- UPSERT: Works for Kudu tables only

INSERT in Impala is primarily optimized for inserting large volumes of data in a single statement, to make effective use of the multi-megabyte HDFS blocks. This is the way in Impala to create new data files. If you intend to insert one or a few rows at a time, such as using the INSERT ... VALUES syntax, that technique is much more efficient for Impala tables stored in HBase.

LOAD DATA moves existing data files into the directory for an Impala table, making them immediately available for Impala queries. This is one way in Impala to work with data files produced by other Hadoop components. (CREATE EXTERNAL TABLE is the other alternative; with external tables, you can query existing data files, while the files remain in their original location.)

In Impala 2.8 and higher, Impala does support the UPDATE, DELETE, and UPSERT statements for Kudu tables. For HDFS or S3 tables, to simulate the effects of an UPDATE or DELETE statement in other database systems, typically you use INSERT or CREATE TABLE AS SELECT to copy data from one table to another, filtering out or changing the appropriate rows during the copy operation.

You can also achieve a result similar to UPDATE by using Impala tables stored in HBase. When you insert a row into an HBase table, and the table already contains a row with the same value for the key column, the older row is hidden, effectively the same as a single-row UPDATE.

Impala can perform DML operations for tables or partitions stored in the Amazon S3 filesystem.

ALTER DATABASE statement

The ALTER DATABASE statement changes the characteristics of a database.

Use the SET OWNER clause to transfer the ownership of the database from the current owner to another user or a role.

The database owner is originally set to the user who creates the database. When object ownership is enabled in Ranger, an owner of a database can have the ALL with GRANT or ALL without GRANT privilege. The term OWNER is used to differentiate between the ALL privilege that is explicitly granted via the GRANT statement and a privilege that is implicitly granted by the CREATE DATABASE statement.

Syntax:

```
ALTER DATABASE database_name SET OWNER USER user_name;
ALTER DATABASE database_name SET OWNER ROLE role_name;
```

Statement type: DDL

Cancellation: Cannot be cancelled.

Added in: Impala 3.1 release.

ALTER TABLE statement

The ALTER TABLE statement changes the structure or properties of an existing Impala table.

In Impala, this is primarily a logical operation that updates the table metadata in the metastore database that Impala shares with Hive. Most ALTER TABLE operations do not actually rewrite, move, and so on the actual data files. (The RENAME TO clause is the one exception; it can cause HDFS files to be moved to different paths.) When you do an ALTER TABLE operation, you typically need to perform corresponding physical filesystem operations, such as rewriting the data files to include extra fields, or converting them to a different file format.

Syntax:

```
ALTER TABLE [old_db_name.]old_table_name RENAME TO
 [new_db_name.]new_table_name

ALTER TABLE name ADD [IF NOT EXISTS] COLUMNS (col_spec[, col_spec ...])
ALTER TABLE name REPLACE COLUMNS (col_spec[, col_spec ...])

ALTER TABLE name ADD COLUMN [IF NOT EXISTS] col_spec
ALTER TABLE name DROP [COLUMN] column_name
ALTER TABLE name CHANGE column_name col_spec

ALTER TABLE name SET OWNER USER user_name
ALTER TABLE name SET OWNER ROLE role_name


-- Kudu tables only.
ALTER TABLE name ALTER [COLUMN] column_name
  { SET kudu_storage_attr attr_value
    | DROP DEFAULT }

kudu_storage_attr ::= { DEFAULT | BLOCK_SIZE | ENCODING | COMPRESSION }

-- Non-Kudu tables only.
```

```
ALTER TABLE name ALTER [COLUMN] column_name
  SET COMMENT 'comment_text'

ALTER TABLE name ADD [IF NOT EXISTS] PARTITION (partition_spec)
  [location_spec]
  [cache_spec]
ALTER TABLE name ADD [IF NOT EXISTS] RANGE PARTITION kudu_partition_spec

ALTER TABLE name DROP [IF EXISTS] PARTITION (partition_spec)
  [PURGE]
ALTER TABLE name DROP [IF EXISTS] RANGE PARTITION kudu_partition_spec

ALTER TABLE name RECOVER PARTITIONS

ALTER TABLE name [PARTITION (partition_spec)]
  SET { FILEFORMAT file_format
  | ROW FORMAT row_format
  | LOCATION 'hdfs_path_of_directory'
  | TBLPROPERTIES (table_properties)
  | SERDEPROPERTIES (serde_properties) }

ALTER TABLE name colname
  ('statsKey'='val, ...)

statsKey ::= numDVs | numNulls | avgSize | maxSize

ALTER TABLE name [PARTITION (partition_spec)] SET { CACHED IN
 'pool_name' [WITH REPLICATION = integer] | UNCACHED }

new_name ::= [new_database.]new_table_name

col_spec ::= col_name type_name  COMMENT 'column-comment' [kudu_attributes]

kudu_attributes ::= { [NOT] NULL | ENCODING codec | COMPRESSION algorithm |
  DEFAULT constant | BLOCK_SIZE number }

partition_spec ::= simple_partition_spec | complex_partition_spec

simple_partition_spec ::= partition_col=constant_value

complex_partition_spec ::= comparison_expression_on_partition_col

kudu_partition_spec ::= constant range_operator
 VALUES range_operator constant | VALUE = constant

cache_spec ::= CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED

location_spec ::= LOCATION 'hdfs_path_of_directory'

table_properties ::= 'name'='value'[, 'name'='value' ...]

serde_properties ::= 'name'='value'[, 'name'='value' ...]

file_format ::= { PARQUET | TEXTFILE | RCFILE | SEQUENCEFILE | AVRO }

row_format ::= DELIMITED
  [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']]
  [LINES TERMINATED BY 'char']
```

Statement type: DDL

Complex type considerations:

In Impala 2.3 and higher, the ALTER TABLE statement can change the metadata for tables containing complex types (ARRAY, STRUCT, and MAP). For example, you can use an ADD COLUMNS, DROP COLUMN, or CHANGE clause to modify the table layout for complex type columns. Although Impala queries only work for complex type columns in Parquet tables, the complex type support in the ALTER TABLE statement applies to all file formats. For example, you can use Impala to update metadata for a staging table in a non-Parquet file format where the data is populated by Hive. Or you can use ALTER TABLE SET FILEFORMAT to change the format of an existing table to Parquet so that Impala can query it. Remember that changing the file format for a table does not convert the data files within the table; you must prepare any Parquet data files containing complex types outside Impala, and bring them into the table using LOAD DATA or updating the table's LOCATION property.

Usage notes:

Whenever you specify partitions in an ALTER TABLE statement, through the PARTITION (*partition_spec*) clause, you must include all the partitioning columns in the specification.

Most of the ALTER TABLE operations work the same for internal tables (managed by Impala) as for external tables (with data files located in arbitrary locations). The exception is renaming a table; for an external table, the underlying data directory is not renamed or moved.

To drop or alter multiple partitions:

In Impala 2.8 and higher, the expression for the partition clause with a DROP or SET operation can include comparison operators such as <, IN, or BETWEEN, and Boolean operators such as AND and OR.

For example, you might drop a group of partitions corresponding to a particular date range after the data “ages out”:

```
alter table historical_data drop partition (year < 1995);
alter table historical_data drop partition (year = 1996 and month between 1
 and 6);
```

For tables with multiple partition keys columns, you can specify multiple conditions separated by commas, and the operation only applies to the partitions that match all the conditions (similar to using an AND clause):

```
alter table historical_data drop partition (year < 1995, last_name like 'A%
');
```

This technique can also be used to change the file format of groups of partitions, as part of an ETL pipeline that periodically consolidates and rewrites the underlying data files in a different file format:

```
alter table fast_growing_data partition (year = 2016, month in (10,11,12))
set fileformat parquet;
```

Note:

The extended syntax involving comparison operators and multiple partitions applies to the SET FILEFORMAT, SET TBLPROPERTIES, SET SERDEPROPERTIES, and SET [UN]CACHED clauses. You can also use this syntax with the PARTITION clause in the COMPUTE INCREMENTAL STATS statement, and with the PARTITION clause of the SHOW FILES statement. Some forms of ALTER TABLE still only apply to one partition at a time: the SET LOCATION and ADD PARTITION clauses. The PARTITION clauses in the LOAD DATA and INSERT statements also only apply to one partition at a time.

A DDL statement that applies to multiple partitions is considered successful (resulting in no changes) even if no partitions match the conditions. The results are the same as if the IF EXISTS clause was specified.

The performance and scalability of this technique is similar to issuing a sequence of single-partition ALTER TABLE statements in quick succession. To minimize bottlenecks due to communication with the metastore database, or causing other DDL operations on the same table to wait, test the effects of performing ALTER TABLE statements that affect large numbers of partitions.

Amazon S3 considerations:

You can specify an s3a:// prefix on the LOCATION attribute of a table or partition to make Impala query data from the Amazon S3 filesystem. In Impala 2.6 and higher, Impala automatically handles creating or removing the associated folders when you issue ALTER TABLE statements with the ADD PARTITION or DROP PARTITION clauses.

In Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed.

HDFS caching (CACHED IN clause):

If you specify the CACHED IN clause, any existing or future data files in the table directory or the partition subdirectories are designated to be loaded into memory with the HDFS caching mechanism.

In Impala 2.2 and higher, the optional WITH REPLICATION clause for CREATE TABLE and ALTER TABLE lets you specify a *replication factor*, the number of hosts on which to cache the same data blocks. When Impala processes a cached data block, where the cache replication factor is greater than 1, Impala randomly selects a host that has a cached copy of that data block. This optimization avoids excessive CPU usage on a single host when the same cached data block is processed multiple times. Cloudera recommends specifying a value greater than or equal to the HDFS block replication factor.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

The following sections show examples of the use cases for various ALTER TABLE clauses.

To rename a table (RENAME TO clause):

The RENAME TO clause lets you change the name of an existing table, and optionally which database it is located in.

For internal tables, this operation physically renames the directory within HDFS that contains the data files; the original directory name no longer exists. By qualifying the table names with database names, you can use this technique to move an internal table (and its associated data directory) from one database to another. For example:

```
create database d1;
create database d2;
create database d3;
use d1;
create table mobile (x int);
use d2;
-- Move table from another database to the current one.
alter table d1.mobile rename to mobile;
```

```
use d1;
-- Move table from one database to another.
alter table d2.mobile rename to d3.mobile;
```

To change the owner of a table:

```
ALTER TABLE name SET OWNER USER user_name;
ALTER TABLE name SET OWNER ROLE role_name;
```

The table owner is originally set to the user who creates the table. When object ownership is enabled in Ranger, an owner of a table can have the ALL with GRANT or ALL without GRANT privilege. The term OWNER is used to differentiate between the ALL privilege that is explicitly granted via the GRANT statement and a privilege that is implicitly granted by the CREATE TABLE statement.

Use the ALTER TABLE SET OWNER to transfer the ownership from the current owner to another user or a role.

To change the physical location where Impala looks for data files associated with a table or partition:

```
ALTER TABLE table_name [PARTITION (partition_spec)] SET LOCATION
 'hdfs_path_of_directory';
```

The path you specify is the full HDFS path where the data files reside, or will be created. Impala does not create any additional subdirectory named after the table. Impala does not move any data files to this new location or change any data files that might already exist in that directory.

To set the location for a single partition, include the PARTITION clause. Specify all the same partitioning columns for the table, with a constant value for each, to precisely identify the single partition affected by the statement:

```
create table p1 (s string) partitioned by (month int, day int);
-- Each ADD PARTITION clause creates a subdirectory in HDFS.
alter table p1 add partition (month=1, day=1);
alter table p1 add partition (month=1, day=2);
alter table p1 add partition (month=2, day=1);
alter table p1 add partition (month=2, day=2);
-- Redirect queries, INSERT, and LOAD DATA for one partition
-- to a specific different directory.
alter table p1 partition (month=1, day=1) set location '/usr/external_data/n
ew_years_day';
```

Note: If you are creating a partition for the first time and specifying its location, for maximum efficiency, use a single ALTER TABLE statement including both the ADD PARTITION and LOCATION clauses, rather than separate statements with ADD PARTITION and SET LOCATION clauses.

To automatically detect new partition directories added through Hive or HDFS operations:

In Impala 2.3 and higher, the RECOVER PARTITIONS clause scans a partitioned table to detect if any new partition directories were added outside of Impala, such as by Hive ALTER TABLE statements or by `hdfs dfs` or `hadoop fs` commands. The RECOVER PARTITIONS clause automatically recognizes any data files present in these new directories, the same as the REFRESH statement does.

For example, here is a sequence of examples showing how you might create a partitioned table in Impala, create new partitions through Hive, copy data files into the new partitions with the `hdfs` command, and have Impala recognize the new partitions and new data:

In Impala, create the table, and a single partition for demonstration purposes:

```
create database recover_partitions;
use recover_partitions;
create table t1 (s string) partitioned by (yy int, mm int);
insert into t1 partition (yy = 2016, mm = 1) values ('Partition exists');
```

```
show files in t1;
+-------------------------------------------------------------------------+---
---+--------------+
| Path                                                                    | Size
 | Partition    |
+-------------------------------------------------------------------------+---
---+--------------+
| /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=1/data.txt | 17B
 | yy=2016/mm=1 |
+-------------------------------------------------------------------------+---
---+--------------+
quit;
```

In Hive, create some new partitions. In a real use case, you might create the partitions and populate them with data as the final stages of an ETL pipeline.

```
hive> use recover_partitions;
OK
hive> alter table t1 add partition (yy = 2016, mm = 2);
OK
hive> alter table t1 add partition (yy = 2016, mm = 3);
OK
hive> quit;
```

For demonstration purposes, manually copy data (a single row) into these new partitions, using manual HDFS operations:

```
$ hdfs dfs -ls /user/hive/warehouse/recover_partitions.db/t1/yy=2016/
Found 3 items
drwxr-xr-x - impala   hive 0 2016-05-09 16:06 /user/hive/warehouse/recover
_partitions.db/t1/yy=2016/mm=1
drwxr-xr-x - jrussell hive 0 2016-05-09 16:14 /user/hive/warehouse/recove
r_partitions.db/t1/yy=2016/mm=2
drwxr-xr-x - jrussell hive 0 2016-05-09 16:13 /user/hive/warehouse/recover_p
artitions.db/t1/yy=2016/mm=3

$ hdfs dfs -cp /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=1/
data.txt \
  /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=2/data.txt
$ hdfs dfs -cp /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=1/d
ata.txt \
  /user/hive/warehouse/recover_partitions.db/t1/yy=2016/mm=3/data.txt
```

```
hive> select * from t1;
OK
Partition exists  2016  1
Partition exists  2016  2
Partition exists  2016  3
hive> quit;
```

In Impala, initially the partitions and data are not visible. Running ALTER TABLE with the RECOVER PARTITIONS clause scans the table data directory to find any new partition directories, and the data files inside them:

```
select * from t1;
+------------------+------+----+
| s                | yy   | mm |
+------------------+------+----+
| Partition exists | 2016 | 1  |
+------------------+------+----+

alter table t1 recover partitions;
select * from t1;
+------------------+------+----+
| s                | yy   | mm |
+------------------+------+----+
| Partition exists | 2016 | 1  |
| Partition exists | 2016 | 3  |
| Partition exists | 2016 | 2  |
+------------------+------+----+
```

To change the key-value pairs of the TBLPROPERTIES and SERDEPROPERTIES fields:

```
ALTER TABLE table_name SET TBLPROPERTIES ('key1'='value1',
 'key2'='value2'[, ...]);
ALTER TABLE table_name SET SERDEPROPERTIES ('key1'='value1',
 'key2'='value2'[, ...]);
```

The TBLPROPERTIES clause is primarily a way to associate arbitrary user-specified data items with a particular table.

The SERDEPROPERTIES clause sets up metadata defining how tables are read or written, needed in some cases by Hive but not used extensively by Impala. You would use this clause primarily to change the delimiter in an existing text table or partition, by setting the 'serialization.format' and 'field.delim' property values to the new delimiter character. The SERDEPROPERTIES clause does not change the existing data in the table. The change only affects the future inserts into the table.

Use the DESCRIBE FORMATTED statement to see the current values of these properties for an existing table.

To manually set or update table or column statistics:

Although for most tables the COMPUTE STATS or COMPUTE INCREMENTAL STATS statement is all you need to keep table and column statistics up to date for a table, sometimes for a very large table or one that is updated frequently, the length of time to recompute all the statistics might make it impractical to run those statements as often as needed. As a workaround, you can use the ALTER TABLE statement to set table statistics at the level of the entire table or a single partition, or column statistics at the level of the entire table.

You can set the numrows value for table statistics by changing the TBLPROPERTIES setting for a table or partition. For example:

```
create table analysis_data stored as parquet as select * from raw_data;
Inserted 1000000000 rows in 181.98s
compute stats analysis_data;
insert into analysis_data select * from smaller_table_we_forgot_before;
Inserted 1000000 rows in 15.32s
-- Now there are 1001000000 rows. We can update this single data point in t
he stats.
```

```
alter table analysis_data set tblproperties('numRows'='1001000000', 'STATS
_GENERATED_VIA_STATS_TASK'='true');
```

```
-- If the table originally contained 1 million rows, and we add another part
ition with 30 thousand rows,
-- change the numRows property for the partition and the overall table.
alter table partitioned_data partition(year=2009, month=4) set tblproperties
 ('numRows'='30000', 'STATS_GENERATED_VIA_STATS_TASK'='true');
alter table partitioned_data set tblproperties ('numRows'='1030000', 'STATS_
GENERATED_VIA_STATS_TASK'='true');
```

In Impala 2.6 and higher, you can use the SET COLUMN STATS clause to set a specific stats value for a particular column.

You specify a case-insensitive symbolic name for the kind of statistics: numDVs, numNulls, avgSize, maxSize. The key names and values are both quoted. This operation applies to an entire table, not a specific partition. For example:

```
create table t1 (x int, s string);
insert into t1 values (1, 'one'), (2, 'two'), (2, 'deux');
show column stats t1;
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| x      | INT    | -1               | -1     | 4        | 4        |
| s      | STRING | -1               | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
alter table t1 set column stats x ('numDVs'='2','numNulls'='0');
alter table t1 set column stats s ('numdvs'='3','maxsize'='4');
show column stats t1;
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| x      | INT    | 2                | 0      | 4        | 4        |
| s      | STRING | 3                | -1     | 4        | -1       |
+--------+--------+------------------+--------+----------+----------+
```

To reorganize columns for a table:

You can add multiple columns at a time using the ALTER TABLE statement. If you specify the IF NOT EXISTS clause, Impala silently ignores the ADD request and does not return an error if a column with the same name exists in the table.

When you replace columns, all the original column definitions are discarded.

You might use these statements if you receive a new set of data files with different data types or columns in a different order. The data files are retained, so if the new columns are incompatible with the old ones, use INSERT OVERWRITE or LOAD DATA OVERWRITE to replace all the data before issuing any further queries.

For example, here is how you might add columns to an existing table. The first ALTER TABLE adds two new columns, and the second ALTER TABLE adds one new column. A single Impala query reads both the old and new data files, containing different numbers of columns. For any columns not present in a particular data file, all the column values are considered to be NULL.

```
create table t1 (x int);
insert into t1 values (1), (2);

alter table t1 add columns (s string, t timestamp);
insert into t1 values (3, 'three', now());

alter table t1 add columns (b boolean);
```

```
insert into t1 values (4, 'four', now(), true);

select * from t1 order by x;
+---+-------+-------------------------------+------+
| x | s     | t                             | b    |
+---+-------+-------------------------------+------+
| 1 | NULL  | NULL                          | NULL |
| 2 | NULL  | NULL                          | NULL |
| 3 | three | 2016-05-11 11:19:45.054457000 | NULL |
| 4 | four  | 2016-05-11 11:20:20.260733000 | true |
+---+-------+-------------------------------+------+
```

You might use the CHANGE clause to rename a single column, or to treat an existing column as a different type than before, such as to switch between treating a column as STRING and TIMESTAMP, or between INT and BIGINT. You can only drop a single column at a time; to drop multiple columns, issue multiple ALTER TABLE statements, or define the new set of columns with a single ALTER TABLE ... REPLACE COLUMNS statement.

The following examples show some safe operations to drop or change columns. Dropping the final column in a table lets Impala ignore the data causing any disruption to existing data files. Changing the type of a column works if existing data values can be safely converted to the new type. The type conversion rules depend on the file format of the underlying table. For example, in a text table, the same value can be interpreted as a STRING or a numeric value, while in a binary format such as Parquet, the rules are stricter and type conversions only work between certain sizes of integers.

```
create table optional_columns (x int, y int, z int, a1 int, a2 int);
insert into optional_columns values (1,2,3,0,0), (2,3,4,100,100);
-- When the last column in the table is dropped, Impala ignores the
-- values that are no longer needed. (Dropping A1 but leaving A2
-- would cause problems, as we will see in a subsequent example.)
alter table optional_columns drop column a2;
alter table optional_columns drop column a1;

select * from optional_columns;
+---+---+---+
| x | y | z |
+---+---+---+
| 1 | 2 | 3 |
| 2 | 3 | 4 |
+---+---+---+
```

```
create table int_to_string (s string, x int);
insert into int_to_string values ('one', 1), ('two', 2);
-- What was an INT column will now be interpreted as STRING.
-- This technique works for text tables but not other file formats.
-- The second X represents the new name of the column, which we keep the s
ame.
alter table int_to_string change x x string;

-- Once the type is changed, we can insert non-integer values into the X
column
-- and treat that column as a string, for example by uppercasing or conca
tenating.
insert into int_to_string values ('three', 'trois');
select s, upper(x) from int_to_string;
+-------+----------+
| s     | upper(x) |
+-------+----------+
| one   | 1        |
| two   | 2        |
| three | TROIS    |
```

```
+-------+----------+
```

Remember that Impala does not actually do any conversion for the underlying data files as a result of ALTER TABLE statements. If you use ALTER TABLE to create a table layout that does not agree with the contents of the underlying files, you must replace the files yourself, such as using LOAD DATA to load a new set of data files, or INSERT OVERWRITE to copy from another table and replace the original data.

The following example shows what happens if you delete the middle column from a Parquet table containing three columns. The underlying data files still contain three columns of data. Because the columns are interpreted based on their positions in the data file instead of the specific column names, a SELECT * query now reads the first and second columns from the data file, potentially leading to unexpected results or conversion errors. For this reason, if you expect to someday drop a column, declare it as the last column in the table, where its data can be ignored by queries after the column is dropped. Or, re-run your ETL process and create new data files if you drop or change the type of a column in a way that causes problems with existing data files.

```
-- Parquet table showing how dropping a column can produce unexpected resul
ts.
create table p1 (s1 string, s2 string, s3 string) stored as parquet;

insert into p1 values ('one', 'un', 'uno'), ('two', 'deux', 'dos'),
  ('three', 'trois', 'tres');
select * from p1;
+-------+-------+------+
| s1    | s2    | s3   |
+-------+-------+------+
| one   | un    | uno  |
| two   | deux  | dos  |
| three | trois | tres |
+-------+-------+------+

alter table p1 drop column s2;
-- The S3 column contains unexpected results.
-- Because S2 and S3 have compatible types, the query reads
-- values from the dropped S2, because the existing data files
-- still contain those values as the second column.
select * from p1;
+-------+-------+
| s1    | s3    |
+-------+-------+
| one   | un    |
| two   | deux  |
| three | trois |
+-------+-------+
```

```
-- Parquet table showing how dropping a column can produce conversion error
s.
create table p2 (s1 string, x int, s3 string) stored as parquet;
insert into p2 values ('one', 1, 'uno'), ('two', 2, 'dos'), ('three', 3, 't
res');
select * from p2;
+-------+---+------+
| s1    | x | s3   |
+-------+---+------+
| one   | 1 | uno  |
| two   | 2 | dos  |
| three | 3 | tres |
+-------+---+------+
alter table p2 drop column x;
select * from p2;
WARNINGS:
```

```
File 'hdfs_filename' has an incompatible Parquet schema for column 'add_col
umns.p2.s3'.
Column type: STRING, Parquet schema:
optional int32 x [i:1 d:1 r:0]

File 'hdfs_filename' has an incompatible Parquet schema for column 'add_col
umns.p2.s3'.
Column type: STRING, Parquet schema:
optional int32 x [i:1 d:1 r:0]
```

In Impala 2.6 and higher, if an Avro table is created without column definitions in the CREATE TABLE statement, and columns are later added through ALTER TABLE, the resulting table is now queryable. Missing values from the newly added columns now default to NULL.

To change the file format that Impala expects data to be in, for a table or partition:

Use an ALTER TABLE ... SET FILEFORMAT clause. You can include an optional PARTITION (*col1=val1*, *col2=val2*, ... clause so that the file format is changed for a specific partition rather than the entire table.

Because this operation only changes the table metadata, you must do any conversion of existing data using regular Hadoop techniques outside of Impala. Any new data created by the Impala INSERT statement will be in the new format. You cannot specify the delimiter for Text files; the data files must be comma-delimited.

To set the file format for a single partition, include the PARTITION clause. Specify all the same partitioning columns for the table, with a constant value for each, to precisely identify the single partition affected by the statement:

```
create table p1 (s string) partitioned by (month int, day int);
-- Each ADD PARTITION clause creates a subdirectory in HDFS.
alter table p1 add partition (month=1, day=1);
alter table p1 add partition (month=1, day=2);
alter table p1 add partition (month=2, day=1);
alter table p1 add partition (month=2, day=2);
-- Queries and INSERT statements will read and write files
-- in this format for this specific partition.
alter table p1 partition (month=2, day=2) set fileformat parquet;
```

To change the row format with different delimiter characters:

Use the SET ROW FORMAT DELIMITED clause to ingest data files that use a different delimiter character or a different line end character. When specifying delimiter and line end characters with the FIELDS TERMINATED BY, ESCAPED BY, and LINES TERMINATED BY clauses, you can use the following:

- A regular ASCII character surrounded by single or double quotation marks.
- An octal sequence, such as '\054' representing a comma or '\0' for ASCII null (hex 00).
- Special characters, such as:
  - '\t' for tab
  - '\n' for newline or linefeed
  - '\r' for carriage return
- An integer in the range '-127'..'128' (with quotation marks but no backslash)

  Negative values are subtracted from 256. For example, FIELDS TERMINATED BY '-2' sets the field delimiter to ASCII code 254.

For the ESCAPED BY clause, choose an escape character that is not used anywhere else in the file. The character following the escape character is taken literally as part of a field value.

Surrounding field values with quotation marks does not help Impala to parse fields with embedded delimiter characters as the quotation marks are considered to be part of the column value.

If you want to use \ as the escape character, specify the clause in `impala-shell` as ESCAPED BY '\\'.

To add or drop partitions for a table, the table must already be partitioned (that is, created with a PARTITIONED BY clause). The partition is a physical directory in HDFS, with a name that encodes a particular column value (the

partition key). The Impala INSERT statement already creates the partition if necessary, so the ALTER TABLE ... ADD PARTITION is primarily useful for importing data by moving or copying existing data files into the HDFS directory corresponding to a partition. (You can use the LOAD DATA statement to move files into the partition directory, or ALTER TABLE ... PARTITION (...) SET LOCATION to point a partition at a directory that already contains data files.

The DROP PARTITION clause is used to remove the HDFS directory and associated data files for a particular set of partition key values; for example, if you always analyze the last 3 months worth of data, at the beginning of each month you might drop the oldest partition that is no longer needed. Removing partitions reduces the amount of metadata associated with the table and the complexity of calculating the optimal query plan, which can simplify and speed up queries on partitioned tables, particularly join queries. Here is an example showing the ADD PARTITION and DROP PARTITION clauses.

To avoid errors while adding or dropping partitions whose existence is not certain, add the optional IF [NOT] EXI STS clause between the ADD or DROP keyword and the PARTITION keyword. That is, the entire clause becomes ADD IF NOT EXISTS PARTITION or DROP IF EXISTS PARTITION. The following example shows how partitions can be created automatically through INSERT statements, or manually through ALTER TABLE statements. The IF [NOT] EXISTS clauses let the ALTER TABLE statements succeed even if a new requested partition already exists, or a partition to be dropped does not exist.

Inserting 2 year values creates 2 partitions:

```
create table partition_t (s string) partitioned by (y int);
insert into partition_t (s,y) values ('two thousand',2000), ('nineteen nine
ty',1990);
show partitions partition_t;
+-------+-------+--------+------+--------------+-------------------+--------
+-------------------+
| y     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format
 | Incremental stats |
+-------+-------+--------+------+--------------+-------------------+--------
+-------+
| 1990  | -1    | 1      | 16B  | NOT CACHED   | NOT CACHED        | TEXT
 | false |
| 2000  | -1    | 1      | 13B  | NOT CACHED   | NOT CACHED        | TEXT
| false |
| Total | -1    | 2      | 29B  | 0B           |                   |
 |       |
+-------+-------+--------+------+--------------+-------------------+--------
+-------+
```

Without the IF NOT EXISTS clause, an attempt to add a new partition might fail:

```
alter table partition_t add partition (y=2000);
ERROR: AnalysisException: Partition spec already exists: (y=2000).
```

The IF NOT EXISTS clause makes the statement succeed whether or not there was already a partition with the specified key value:

```
alter table partition_t add if not exists partition (y=2000);
alter table partition_t add if not exists partition (y=2010);
show partitions partition_t;
+-------+-------+--------+------+--------------+-------------------+--------
+-------------------+
| y     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format
| Incremental stats |
+-------+-------+--------+------+--------------+-------------------+--------
+-------+
```

```
| 1990  | -1    | 1      | 16B  | NOT CACHED   | NOT CACHED        | TEXT
| false |
| 2000  | -1    | 1      | 13B  | NOT CACHED   | NOT CACHED        | TEXT
 | false |
| 2010  | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT
| false |
| Total | -1    | 2      | 29B  | 0B           |                   |
 |       |
+-------+-------+--------+------+--------------+-------------------+--------
+-------+
```

Likewise, the IF EXISTS clause lets DROP PARTITION succeed whether or not the partition is already in the table:

```
alter table partition_t drop if exists partition (y=2000);
alter table partition_t drop if exists partition (y=1950);
show partitions partition_t;
+-------+-------+--------+------+--------------+-------------------+--------
+-------------------+
| y     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format
 | Incremental stats |
+-------+-------+--------+------+--------------+-------------------+--------
+-------+
| 1990  | -1    | 1      | 16B  | NOT CACHED   | NOT CACHED        | TEXT
 | false |
| 2010  | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT
| false |
| Total | -1    | 1      | 16B  | 0B           |                   |
 |       |
+-------+-------+--------+------+--------------+-------------------+--------
+-------+
```

The optional PURGE keyword, available in Impala 2.3 and higher, is used with the DROP PARTITION clause to remove associated HDFS data files immediately rather than going through the HDFS trashcan mechanism. Use this keyword when dropping a partition if it is crucial to remove the data as quickly as possible to free up space, or if there is a problem with the trashcan, such as the trash cannot being configured or being in a different HDFS encryption zone than the data files.

```
-- Create an empty table and define the partitioning scheme.
create table part_t (x int) partitioned by (month int);
-- Create an empty partition into which you could copy data files from some
 other source.
alter table part_t add partition (month=1);
-- After changing the underlying data, issue a REFRESH statement to make
the data visible in Impala.
refresh part_t;
-- Later, do the same for the next month.
alter table part_t add partition (month=2);

-- Now you no longer need the older data.
alter table part_t drop partition (month=1);
-- If the table was partitioned by month and year, you would issue a state
ment like:
-- alter table part_t drop partition (year=2003,month=1);
-- which would require 12 ALTER TABLE statements to remove a year's worth of
 data.
-- If the data files for subsequent months were in a different file format,
-- you could set a different file format for the new partition as you create
 it.
alter table part_t add partition (month=3) set fileformat=parquet;
```

The value specified for a partition key can be an arbitrary constant expression, without any references to columns. For example:

```
alter table time_data add partition (month=concat('Decem','ber'));
alter table sales_data add partition (zipcode = cast(9021 * 10 as string));
```

Note:

An alternative way to reorganize a table and its associated data files is to use CREATE TABLE to create a variation of the original table, then use INSERT to copy the transformed or reordered data to the new table. The advantage of ALTER TABLE is that it avoids making a duplicate copy of the data files, allowing you to reorganize huge volumes of data in a space-efficient way using familiar Hadoop techniques.

To switch a table between internal and external:

You can switch a table from internal to external, or from external to internal, by using the ALTER TABLE statement:

```
-- Switch a table from internal to external.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='TRUE');

-- Switch a table from external to internal.
ALTER TABLE table_name SET TBLPROPERTIES('EXTERNAL'='FALSE');
```

If the Kudu service is integrated with the Hive Metastore, the above operations are not supported.

Cancellation: Cannot be cancelled.

HDFS permissions:

Most ALTER TABLE clauses do not actually read or write any HDFS files, and so do not depend on specific HDFS permissions. For example, the SET FILEFORMAT clause does not actually check the file format existing data files or convert them to the new format, and the SET LOCATION clause does not require any special permissions on the new location. (Any permission-related failures would come later, when you actually query or insert into the table.)

In general, ALTER TABLE clauses that do touch HDFS files and directories require the same HDFS permissions as corresponding CREATE, INSERT, or SELECT statements. The permissions allow the user ID that the `impalad` daemon runs under, typically the impala user, to read or write files or directories, or (in the case of the execute bit) descend into a directory. The RENAME TO clause requires read, write, and execute permission in the source and destination database directories and in the table data directory, and read and write permission for the data files within the table. The ADD PARTITION and DROP PARTITION clauses require write and execute permissions for the associated partition directory.

Kudu considerations:

Because of the extra constraints and features of Kudu tables, such as the NOT NULL and DEFAULT attributes for columns, ALTER TABLE has specific requirements related to Kudu tables:

- In an ADD COLUMNS operation, you can specify the NULL, NOT NULL, and DEFAULT *default_value* column attributes.
- In Impala 2.9 and higher, you can also specify the ENCODING, COMPRESSION, and BLOCK_SIZE attributes when adding a column.
- If you add a column with a NOT NULL attribute, it must also have a DEFAULT attribute, so the default value can be assigned to that column for all existing rows.
- The DROP COLUMN clause works the same for a Kudu table as for other kinds of tables.
- Although you can change the name of a column with the CHANGE clause, you cannot change the type of a column in a Kudu table.
- You cannot change the nullability of existing columns in a Kudu table.
- In Impala 2.10, you can change the default value, encoding, compression, or block size of existing columns in a Kudu table by using the SET clause.
- You cannot use the REPLACE COLUMNS clause with a Kudu table.

- The RENAME TO clause for a Kudu table only affects the name stored in the metastore database that Impala uses to refer to the table. To change which underlying Kudu table is associated with an Impala table name, you must change the TBLPROPERTIES property of the table: SET TBLPROPERTIES('kudu.table_name '='*kudu_tbl_name*). You can only change underlying Kudu tables for the external tables.

The following are some examples of using the ADD COLUMNS clause for a Kudu table:

```
CREATE TABLE t1 ( x INT, PRIMARY KEY (x) )
  PARTITION BY HASH (x) PARTITIONS 16
  STORED AS KUDU

ALTER TABLE t1 ADD COLUMNS (y STRING ENCODING prefix_encoding);
ALTER TABLE t1 ADD COLUMNS (z INT DEFAULT 10);
ALTER TABLE t1 ADD COLUMNS (a STRING NOT NULL DEFAULT '', t TIMESTAMP COM
PRESSION default_compression);
```

The following are some examples of modifying column defaults and storage attributes for a Kudu table:

```
create table kt (x bigint primary key, s string default 'yes', t timestamp)
  stored as kudu;

-- You can change the default value for a column, which affects any rows
-- inserted after this change is made.
alter table kt alter column s set default 'no';
-- You can remove the default value for a column, which affects any rows
-- inserted after this change is made. If the column is nullable, any
-- future inserts default to NULL for this column. If the column is marked
-- NOT NULL, any future inserts must specify a value for the column.
alter table kt alter column s drop default;
insert into kt values (1, 'foo', now());
-- Because of the DROP DEFAULT above, omitting S from the insert
-- gives it a value of NULL.
insert into kt (x, t) values (2, now());

select * from kt;
+---+------+-------------------------------+
| x | s    | t                             |
+---+------+-------------------------------+
| 2 | NULL | 2017-10-02 00:03:40.652156000 |
| 1 | foo  | 2017-10-02 00:03:04.346185000 |
+---+------+-------------------------------+

-- Other storage-related attributes can also be changed for columns.
-- These changes take effect for any newly inserted rows, or rows
-- rearranged due to compaction after deletes or updates.
alter table kt alter column s set encoding prefix_encoding;
-- The COLUMN keyword is optional in the syntax.
alter table kt alter x set block_size 2048;
alter table kt alter column t set compression zlib;

desc kt;
+------+-----------+---------+-------------+----------+---------------+--
---------------+---------------------+------------+
| name | type      | comment | primary_key | nullable | default_value | enco
ding        | compression         | block_size |
+------+-----------+---------+-------------+----------+---------------+----
-------------+---------------------+------------+
| x    | bigint    |         | true        | false    |               | AU
TO_ENCODING   | DEFAULT_COMPRESSION | 2048       |
| s    | string    |         | false       | true     |               | P
REFIX_ENCODING | DEFAULT_COMPRESSION | 0          |
```

```
| t    | timestamp |          | false       | true     |               | AUTO
_ENCODING   | ZLIB              | 0          |
+------+-----------+---------+-------------+----------+---------------+----
-------------+---------------------+------------+
```

Kudu tables all use an underlying partitioning mechanism. The partition syntax is different than for non-Kudu tables. You can use the ALTER TABLE statement to add and drop *range partitions* from a Kudu table. Any new range must not overlap with any existing ranges. Dropping a range removes all the associated rows from the table.

Related Information

Impala with Amazon S3

ALTER VIEW statement

The ALTER VIEW statement changes the characteristics of a view.

Because a view is a logical construct, an alias for a query, with no physical data behind it, ALTER VIEW only involves changes to metadata in the metastore database, not any data files in HDFS.

To see the definition of the updated view, issue a DESCRIBE FORMATTED statement.

Syntax:

```
ALTER VIEW [database_name.]view_name
   [(column_name [COMMENT 'column_comment'][, ...])]
   AS select_statement;

ALTER VIEW [database_name.]view_name
   RENAME TO [database_name.]view_name;

ALTER VIEW [database_name.]view_name SET OWNER USER user_name;
ALTER VIEW [database_name.]view_name SET OWNER ROLE role_name;
```

- The AS clause associates the view with a different query.

  An optional list of column names can be specified with or without the column-level comments.

  For example:

  ```
  ALTER VIEW v1 AS SELECT x, UPPER(s) s FROM t2;
  ALTER VIEW v1 (c1, c2) AS SELECT x, UPPER(s) s FROM t2;
  ALTER VIEW v7 (c1 COMMENT 'Comment for c1', c2) AS SELECT t1.c1, t1.c2 F
  ROM t1;
  ```

- The RENAME TO clause changes the name of the view, moves the view to a different database, or both.

  For example:

  ```
  ALTER VIEW db1.v1 RENAME TO db2.v2; -- Move the view to a different data
  base with a new name.
  ALTER VIEW db1.v1 RENAME TO db1.v2;  -- Rename the view in the same databa
  se.
  ALTER VIEW db1.v1 RENAME TO db2.v1; -- Move the view to a difference dat
  abase with the same view name.
  ```

- The SET OWNER clause transfers the ownership of the view from the current owner to another user or a role.

  The view owner is originally set to the user who creates the view. When object ownership is enabled in Ranger, an owner of a view can have the ALL with GRANT or ALL without GRANT privilege. The term OWNER is used to differentiate between the ALL privilege that is explicitly granted via the GRANT statement and a privilege that is implicitly granted by the CREATE VIEW statement.

Statement type: DDL

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

COMMENT statement

The COMMENT statement adds, changes, or removes a comment about a database, a table, or a column.

You can alternatively use the CREATE and ALTER statements to add comments to the objects.

You can view the comment on a database, a table, or a column using the SHOW or DESCRIBE statement.

Syntax:

```
COMMENT ON DATABASE db_name IS {'comment' | NULL}
```

```
COMMENT ON TABLE [db_name.]table_name IS {'comment' | NULL}
```

```
COMMENT ON COLUMN [db_name.]table_name.column_name IS {'comment' | NULL}
```

Parameters:

- *db_name*: Specify the database name if not for the current database.
- NULL: If given for the comment, removes the existing comment.
- The *comment* string can be up to 256 characters long.

Privileges required:

To add a comment, the ALTER privilege on the object is required.

To view a comment, the SELECT, INSERT, or REFRESH on the object is required.

Usage notes:

Added in: Impala 3.1 release.

COMPUTE STATS statement

The COMPUTE STATS statement gathers information about volume and distribution of data in a table and all associated columns and partitions. The information is stored in the metastore database, and used by Impala to help optimize queries.

For example, if Impala can determine that a table is large or small, or has many or few distinct values it can organize and parallelize the work appropriately for a join query or insert operation. For details about the kinds of information gathered by this statement, see the *Table and column statistics* topic.

Syntax:

```
COMPUTE STATS [db_name.]table_name  [ ( column_list ) ] [TABLESAMPLE
 SYSTEM(percentage) [REPEATABLE(seed)]]

column_list ::= column_name [ , column_name, ... ]
COMPUTE INCREMENTAL STATS [db_name.]table_name [PARTITION (partition_spec)]
```

```
partition_spec ::= partition_col=constant_value

partition_spec ::= simple_partition_spec | complex_partition_spec

simple_partition_spec ::= partition_col=constant_value

complex_partition_spec ::= comparison_expression_on_partition_col
```

The PARTITION clause is only allowed in combination with the INCREMENTAL clause. It is optional for COMP UTE INCREMENTAL STATS, and required for DROP INCREMENTAL STATS. Whenever you specify partitions through the PARTITION (*partition_spec*) clause in a COMPUTE INCREMENTAL STATS or DROP INCREME NTAL STATS statement, you must include all the partitioning columns in the specification, and specify constant values for all the partition key columns.

Usage notes:

Originally, Impala relied on users to run the Hive ANALYZE TABLE statement, but that method of gathering statistics proved unreliable and difficult to use. The Impala COMPUTE STATS statement was built to improve the reliability and user-friendliness of this operation. COMPUTE STATS does not require any setup steps or special configuration. You only run a single Impala COMPUTE STATS statement to gather both table and column statistics, rather than separate Hive ANALYZE TABLE statements for each kind of statistics.

For non-incremental COMPUTE STATS statement, the columns for which statistics are computed can be specified with an optional comma-separate list of columns.

If no column list is given, the COMPUTE STATS statement computes column-level statistics for all columns of the table. This adds potentially unneeded work for columns whose stats are not needed by queries. It can be especially costly for very wide tables and unneeded large string fields.

COMPUTE STATS returns an error when a specified column cannot be analyzed, such as when the column does not exist, the column is of an unsupported type for COMPUTE STATS, e.g. colums of complex types, or the column is a partitioning column.

If an empty column list is given, no column is analyzed by COMPUTE STATS.

In Impala 2.12 and higher, an optional TABLESAMPLE clause immediately after a table reference specifies that the COMPUTE STATS operation only processes a specified percentage of the table data. For tables that are so large that a full COMPUTE STATS operation is impractical, you can use COMPUTE STATS with a TABLESAMPLE clause to extrapolate statistics from a sample of the table data. See *Table and column statistics* about the experimental stats extrapolation and sampling features.

The COMPUTE INCREMENTAL STATS variation is a shortcut for partitioned tables that works on a subset of partitions rather than the entire table. The incremental nature makes it suitable for large tables with many partitions, where a full COMPUTE STATS operation takes too long to be practical each time a partition is added or dropped.

Important:

For a particular table, use either COMPUTE STATS or COMPUTE INCREMENTAL STATS. The two kinds of stats do not interoperate with each other at the table level. Without dropping the stats, if you run COMPUTE INCREMENTAL STATS it will overwrite the full compute stats or if you run COMPUTE STATS it will drop all incremental stats for consistency.

When you run COMPUTE INCREMENTAL STATS on a table for the first time, the statistics are computed again from scratch regardless of whether the table already has statistics. Therefore, expect a one-time resource-intensive operation for scanning the entire table when running COMPUTE INCREMENTAL STATS for the first time on a given table.

In Impala 3.0 and lower, approximately 400 bytes of metadata per column per partition are needed for caching. Tables with a big number of partitions and many columns can add up to a significant memory overhead as the metadata must be cached on the `catalogd` host and on every `impalad` host that is eligible to be a coordinator. If this metadata for all tables exceeds 2 GB, you might experience service downtime. In Impala 3.1 and higher, the issue was alleviated with an improved handling of incremental stats.

COMPUTE INCREMENTAL STATS only applies to partitioned tables. If you use the INCREMENTAL clause for an unpartitioned table, Impala automatically uses the original COMPUTE STATS statement. Such tables display false under the Incremental stats column of the SHOW TABLE STATS output.

Note:

Because many of the most performance-critical and resource-intensive operations rely on table and column statistics to construct accurate and efficient plans, COMPUTE STATS is an important step at the end of your ETL process. Run COMPUTE STATS on all tables as your first step during performance tuning for slow queries, or troubleshooting for out-of-memory conditions:

- Accurate statistics help Impala construct an efficient query plan for join queries, improving performance and reducing memory usage.
- Accurate statistics help Impala distribute the work effectively for insert operations into Parquet tables, improving performance and reducing memory usage.
- Accurate statistics help Impala estimate the memory required for each query, which is important when you use resource management features, such as admission control and the YARN resource management framework. The statistics help Impala to achieve high concurrency, full utilization of available memory, and avoid contention with workloads from other Hadoop components.
- In Impala 2.8 and higher, when you run the COMPUTE STATS or COMPUTE INCREMENTAL STATS statement against a Parquet table, Impala automatically applies the query option setting MT_DOP=4 to increase the amount of intra-node parallelism during this CPU-intensive operation. See MT_DOP query option for details about what this query option does and how to use it with CPU-intensive SELECT statements.

Computing stats for groups of partitions:

In Impala 2.8 and higher, you can run COMPUTE INCREMENTAL STATS on multiple partitions, instead of the entire table or one partition at a time. You include comparison operators other than = in the PARTITION clause, and the COMPUTE INCREMENTAL STATS statement applies to all partitions that match the comparison expression.

For example, the INT_PARTITIONS table contains 4 partitions. The following COMPUTE INCREMENTAL STATS statements affect some but not all partitions, as indicated by the Updated *n* partition(s) messages. The partitions that are affected depend on values in the partition key column X that match the comparison expression in the PARTITION clause.

```
show partitions int_partitions;
+-------+-------+--------+------+-------------+------------------+-------
--+...
| x     | #Rows | #Files | Size | Bytes Cached | Cache Replication | Format
  |...
+-------+-------+--------+------+-------------+------------------+-------
--+...
| 99    | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | PARQ
UET |...
| 120   | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT
 |...
| 150   | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT
  |...
| 200   | -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT
 |...
| Total | -1    | 0      | 0B   | 0B           |                   |
  |...
+-------+-------+--------+------+-------------+------------------+-------
--+...

compute incremental stats int_partitions partition (x < 100);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 1 column(s). |
```

```
+-----------------------------------------+

compute incremental stats int_partitions partition (x in (100, 150, 200));
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 2 partition(s) and 1 column(s). |
+-----------------------------------------+

compute incremental stats int_partitions partition (x between 100 and 175);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 2 partition(s) and 1 column(s). |
+-----------------------------------------+

compute incremental stats int_partitions partition (x in (100, 150, 200) or
x < 100);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 3 partition(s) and 1 column(s). |
+-----------------------------------------+

compute incremental stats int_partitions partition (x != 150);
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 3 partition(s) and 1 column(s). |
+-----------------------------------------+
```

Complex type considerations:

Currently, the statistics created by the COMPUTE STATS statement do not include information about complex type columns. The column stats metrics for complex columns are always shown as -1. For queries involving complex type columns, Impala uses heuristics to estimate the data distribution within such columns.

HBase considerations:

COMPUTE STATS works for HBase tables also. The statistics gathered for HBase tables are somewhat different than for HDFS-backed tables, but that metadata is still used for optimization when HBase tables are involved in join queries.

Amazon S3 considerations:

COMPUTE STATS also works for tables where data resides in the Amazon Simple Storage Service (S3).

Performance considerations:

The statistics collected by COMPUTE STATS are used to optimize join queries INSERT operations into Parquet tables, and other resource-intensive kinds of SQL statements.

For large tables, the COMPUTE STATS statement itself might take a long time and you might need to tune its performance. The COMPUTE STATS statement does not work with the EXPLAIN statement, or the SUMMARY command in `impala-shell`. You can use the PROFILE statement in `impala-shell` to examine timing information for the statement as a whole. If a basic COMPUTE STATS statement takes a long time for a partitioned table, consider switching to the COMPUTE INCREMENTAL STATS syntax so that only newly added partitions are analyzed each time.

Examples:

This example shows two tables, T1 and T2, with a small number distinct values linked by a parent-child relationship between T1.ID and T2.PARENT. T1 is tiny, while T2 has approximately 100K rows. Initially, the statistics includes physical measurements such as the number of files, the total size, and size measurements for fixed-length columns

such as with the INT type. Unknown values are represented by -1. After running COMPUTE STATS for each table, much more information is available through the SHOW STATS statements. If you were running a join query involving both of these tables, you would need statistics for both tables to get the most effective optimization for the query.

```
[localhost:21000] > show table stats t1;
Query: show table stats t1
+-------+--------+------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+------+--------+
| -1    | 1      | 33B  | TEXT   |
+-------+--------+------+--------+
Returned 1 row(s) in 0.02s
[localhost:21000] > show table stats t2;
Query: show table stats t2
+-------+--------+----------+--------+
| #Rows | #Files | Size     | Format |
+-------+--------+----------+--------+
| -1    | 28     | 960.00KB | TEXT   |
+-------+--------+----------+--------+
Returned 1 row(s) in 0.01s
[localhost:21000] > show column stats t1;
Query: show column stats t1
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| id     | INT    | -1               | -1     | 4        | 4        |
| s      | STRING | -1               | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 1.71s
[localhost:21000] > show column stats t2;
Query: show column stats t2
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| parent | INT    | -1               | -1     | 4        | 4        |
| s      | STRING | -1               | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.01s
[localhost:21000] > compute stats t1;
Query: compute stats t1
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 5.30s
[localhost:21000] > show table stats t1;
Query: show table stats t1
+-------+--------+------+--------+
| #Rows | #Files | Size | Format |
+-------+--------+------+--------+
| 3     | 1      | 33B  | TEXT   |
+-------+--------+------+--------+
Returned 1 row(s) in 0.01s
[localhost:21000] > show column stats t1;
Query: show column stats t1
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| id     | INT    | 3                | -1     | 4        | 4        |
| s      | STRING | 3                | -1     | -1       | -1       |
+--------+--------+------------------+--------+----------+----------+
```

```
Returned 2 row(s) in 0.02s
[localhost:21000] > compute stats t2;
Query: compute stats t2
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 2 column(s). |
+-----------------------------------------+
Returned 1 row(s) in 5.70s
[localhost:21000] > show table stats t2;
Query: show table stats t2
+-------+--------+----------+--------+
| #Rows | #Files | Size     | Format |
+-------+--------+----------+--------+
| 98304 | 1      | 960.00KB | TEXT   |
+-------+--------+----------+--------+
Returned 1 row(s) in 0.03s
[localhost:21000] > show column stats t2;
Query: show column stats t2
+--------+--------+------------------+--------+----------+----------+
| Column | Type   | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+--------+------------------+--------+----------+----------+
| parent | INT    | 3                | -1     | 4        | 4        |
| s      | STRING | 6                | -1     | 14       | 9.3      |
+--------+--------+------------------+--------+----------+----------+
Returned 2 row(s) in 0.01s
```

The following example shows how to use the INCREMENTAL clause, available in Impala 2.1.0 and higher. The COMPUTE INCREMENTAL STATS syntax lets you collect statistics for newly added or changed partitions, without rescanning the entire table.

```
-- Initially the table has no incremental stats, as indicated
-- 'false' under Incremental stats.
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+-------
-----------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Increme
ntal stats
+-------------+-------+--------+----------+--------------+---------+-----
-------------
| Books       | -1    | 1      | 223.74KB | NOT CACHED   | PARQUET | false
| Children    | -1    | 1      | 230.05KB | NOT CACHED   | PARQUET | false
| Electronics | -1    | 1      | 232.67KB | NOT CACHED   | PARQUET | false
| Home        | -1    | 1      | 232.56KB | NOT CACHED   | PARQUET | false
| Jewelry     | -1    | 1      | 223.72KB | NOT CACHED   | PARQUET | false
| Men         | -1    | 1      | 231.25KB | NOT CACHED   | PARQUET | false
| Music       | -1    | 1      | 237.90KB | NOT CACHED   | PARQUET | false
| Shoes       | -1    | 1      | 234.90KB | NOT CACHED   | PARQUET | false
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | -1    | 1      | 226.27KB | NOT CACHED   | PARQUET | false
| Total       | -1    | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+-------
-----------
-- After the first COMPUTE INCREMENTAL STATS,
-- all partitions have stats. The first
-- COMPUTE INCREMENTAL STATS scans the whole
-- table, discarding any previous stats from
-- a traditional COMPUTE STATS statement.
compute incremental stats item_partitioned;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 10 partition(s) and 21 column(s). |
```

```
+-----------------------------------------+
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+--------
----------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incr
emental stats
+-------------+-------+--------+----------+--------------+---------+------
------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------
------------

-- Add a new partition...
alter table item_partitioned add partition (i_category='Camping');
-- Add or replace files in HDFS outside of Impala,
-- rendering the stats for a partition obsolete.
!import_data_into_sports_partition.sh
refresh item_partitioned;
drop incremental stats item_partitioned partition (i_category='Sports');
-- Now some partitions have incremental stats
-- and some do not.
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+-------
-----------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Increme
ntal stats
+-------------+-------+--------+----------+--------------+---------+-----
-------------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Camping     | -1    | 1      | 408.02KB | NOT CACHED   | PARQUET | false
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 11     | 2.65MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+-----
-------------

-- After another COMPUTE INCREMENTAL STATS,
-- all partitions have incremental stats, and only the 2
-- partitions without incremental stats were scanned.
compute incremental stats item_partitioned;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 2 partition(s) and 21 column(s). |
+-----------------------------------------+
show table stats item_partitioned;
```

```
+-------------+-------+--------+----------+--------------+---------+------
------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Increm
ental stats
+-------------+-------+--------+----------+--------------+---------+-------
----------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Camping     | 5328  | 1      | 408.02KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 11     | 2.65MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+------
------------
```

File format considerations:

The COMPUTE STATS statement works with tables created with any of the file formats supported by Impala. The following considerations apply to COMPUTE STATS depending on the file format of the table.

The COMPUTE STATS statement works with text tables with no restrictions. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with Parquet tables. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with Avro tables without restriction in Impala 2.2 and higher. In earlier releases, COMPUTE STATS worked only for Avro tables created through Hive, and required the CREATE TABLE statement to use SQL-style column names and types rather than an Avro-style schema specification.

The COMPUTE STATS statement works with RCFile tables with no restrictions. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with SequenceFile tables with no restrictions. These tables can be created through either Impala or Hive.

The COMPUTE STATS statement works with partitioned tables, whether all the partitions use the same file format, or some partitions are defined through ALTER TABLE to use different file formats.

Statement type: DDL

Cancellation: Certain multi-stage statements (CREATE TABLE AS SELECT and COMPUTE STATS) can be cancelled during some stages, when running INSERT or SELECT operations internally. To cancel this statement, use Ctrl-C from the `impala-shell` interpreter, the Cancel button from the Watch page in Hue, Actions > Cancel from the Queries list in Cloudera Manager, or Cancel from the list of in-flight queries (for a particular node) on the Queries tab in the Impala web UI (port 25000).

Restrictions:

Note: Prior to Impala 1.4.0, COMPUTE STATS counted the number of NULL values in each column and recorded that figure in the metastore database. Because Impala does not currently use the NULL count during query planning, Impala 1.4.0 and higher speeds up the COMPUTE STATS statement by skipping this NULL counting.

Internal details:

Behind the scenes, the COMPUTE STATS statement executes two statements: one to count the rows of each partition in the table (or the entire table if unpartitioned) through the COUNT(*) function, and another to count the approximate number of distinct values in each column through the NDV() function. You might see these queries in

your monitoring and diagnostic displays. The same factors that affect the performance, scalability, and execution of other queries (such as parallel execution, memory usage, admission control, and timeouts) also apply to the queries run by the COMPUTE STATS statement.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read permission for all affected files in the source directory: all files in the case of an unpartitioned table or a partitioned table in the case of COMPUTE STATS; or all the files in partitions without incremental stats in the case of COMPUTE INCREMEN TAL STATS. It must also have read and execute permissions for all relevant directories holding the data files. (Essentially, COMPUTE STATS requires the same permissions as the underlying SELECT queries it runs against the table.)

Kudu considerations:

The COMPUTE STATS statement applies to Kudu tables. Impala does not compute the number of rows for each partition for Kudu tables. Therefore, you do not need to re-run the operation when you see -1 in the # Rows column of the output from SHOW TABLE STATS. That column always shows -1 for all Kudu tables.

CREATE DATABASE statement

The CREATE DATABASE statement is used to create a new database.

In Impala, a database is both:

- A logical construct for grouping together related tables, views, and functions within their own namespace. You might use a separate database for each application, set of related tables, or round of experimentation.
- A physical construct represented by a directory tree in HDFS. Tables (internal tables), partitions, and data files are all located under this directory. You can perform HDFS-level operations such as backing it up and measuring space usage, or remove it with a DROP DATABASE statement.

Syntax:

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name[COMMENT
 'database_comment']
  [LOCATION hdfs_path];
```

Statement type: DDL

Usage notes:

A database is physically represented as a directory in HDFS, with a filename extension .db, under the main Impala data directory. If the associated HDFS directory does not exist, it is created for you. All databases and their associated directories are top-level objects, with no physical or logical nesting.

After creating a database, to make it the current database within an `impala-shell` session, use the USE statement. You can refer to tables in the current database without prepending any qualifier to their names.

When you first connect to Impala through `impala-shell`, the database you start in (before issuing any CREATE DATABASE or USE statements) is named default.

Impala includes another predefined database, _impala_builtins, that serves as the location for the built-in functions. To see the built-in functions, use a statement like the following:

```
show functions in _impala_builtins;
show functions in _impala_builtins like '*substring*';
```

After creating a database, your `impala-shell` session or another `impala-shell` connected to the same node can immediately access that database. To access the database through the Impala daemon on a different node, issue the INVALIDATE METADATA statement first while connected to that other node.

Setting the LOCATION attribute for a new database is a way to work with sets of files in an HDFS directory structure outside the default Impala data directory, as opposed to setting the LOCATION attribute for each individual table.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

Hive considerations:

When you create a database in Impala, the database can also be used by Hive. When you create a database in Hive, issue an INVALIDATE METADATA statement in Impala to make Impala permanently aware of the new database.

The SHOW DATABASES statement lists all databases, or the databases whose name matches a wildcard pattern. In Impala 2.5 and higher, the SHOW DATABASES output includes a second column that displays the associated comment, if any, for each database.

Amazon S3 considerations:

To specify that any tables created within a database reside on the Amazon S3 system, you can include an s3a:// prefix on the LOCATION attribute. In Impala 2.6 and higher, Impala automatically creates any required folders as the databases, tables, and partitions are created, and removes them when they are dropped.

In Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have write permission for the parent HDFS directory under which the database is located.

Examples:

```
create database first_db;
use first_db;
create table t1 (x int);

create database second_db;
use second_db;
-- Each database has its own namespace for tables.
-- You can reuse the same table names in each database.
create table t1 (s string);

create database temp;

-- You can either USE a database after creating it,
-- or qualify all references to the table name with the name of the datab
ase.
-- Here, tables T2 and T3 are both created in the TEMP database.

create table temp.t2 (x int, y int);
use database temp;
create table t3 (s string);

-- You cannot drop a database while it is selected by the USE statement.
drop database temp;
ERROR: AnalysisException: Cannot drop current default database: temp

-- The always-available database 'default' is a convenient one to USE
-- before dropping a database you created.
use default;
-- Before dropping a database, first drop all the tables inside it,
-- or in Impala 2.3 and higher use the CASCADE clause.
drop database temp;
```

```
ERROR: ImpalaRuntimeException: Error making 'dropDatabase' RPC to Hive Meta
store:
CAUSED BY: InvalidOperationException: Database temp is not empty
show tables in temp;
+------+
| name |
+------+
| t3   |
+------+

-- Impala 2.3 and higher:
drop database temp cascade;

-- Earlier releases:
drop table temp.t3;
drop database temp;
```

Related Information

Impala with Amazon S3

CREATE FUNCTION statement

Creates a user-defined function (UDF), which you can use to implement custom logic during SELECT or INSERT operations.

Syntax:

The syntax is different depending on whether you create a scalar UDF, which is called once for each row and implemented by a single function, or a user-defined aggregate function (UDA), which is implemented by multiple functions that compute intermediate results across sets of rows.

In Impala 2.5 and higher, the syntax is also different for creating or dropping scalar Java-based UDFs. The statements for Java UDFs use a new syntax, without any argument types or return type specified. Java-based UDFs created using the new syntax persist across restarts of the Impala catalog server, and can be shared transparently between Impala and Hive.

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'
```

To create a persistent UDA, which must be written in C++, issue a CREATE AGGREGATE FUNCTION statement:

```
CREATE [AGGREGATE] FUNCTION [IF NOT EXISTS]
 [db_name.]function_name([arg_type[, arg_type...])
  RETURNS return_type
  [INTERMEDIATE type_spec]
  LOCATION 'hdfs_path'
  [INIT_FN='function]
  UPDATE_FN='function
  MERGE_FN='function
  [PREPARE_FN='function]
  [CLOSEFN='function]
  [SERIALIZE_FN='function]
```

```
  [FINALIZE_FN='function]
```

Statement type: DDL

Varargs notation:

Note:

Variable-length argument lists are supported for C++ UDFs, but currently not for Java UDFs.

If the underlying implementation of your function accepts a variable number of arguments:

- The variable arguments must go last in the argument list.
- The variable arguments must all be of the same type.
- You must include at least one instance of the variable arguments in every function call invoked from SQL.
- You designate the variable portion of the argument list in the CREATE FUNCTION statement by including ... immediately after the type name of the first variable argument. For example, to create a function that accepts an INT argument, followed by a BOOLEAN, followed by one or more STRING arguments, your CREATE FUNCTION statement would look like:

```
CREATE FUNCTION func_name (INT, BOOLEAN, STRING ...)
  RETURNS type LOCATION 'path' SYMBOL='entry_point';
```

See #unique_9/unique_9_Connect_42_udf_varargs for how to code a C++ UDF to accept variable-length argument lists.

Scalar and aggregate functions:

The simplest kind of user-defined function returns a single scalar value each time it is called, typically once for each row in the result set. This general kind of function is what is usually meant by UDF. User-defined aggregate functions (UDAs) are a specialized kind of UDF that produce a single value based on the contents of multiple rows. You usually use UDAs in combination with a GROUP BY clause to condense a large result set into a smaller one, or even a single row summarizing column values across an entire table.

You create UDAs by using the CREATE AGGREGATE FUNCTION syntax. The clauses INIT_FN, UPDATE_FN, MERGE_FN, SERIALIZE_FN, FINALIZE_FN, and INTERMEDIATE only apply when you create a UDA rather than a scalar UDF.

The *_FN clauses specify functions to call at different phases of function processing.

- Initialize: The function you specify with the INIT_FN clause does any initial setup, such as initializing member variables in internal data structures. This function is often a stub for simple UDAs. You can omit this clause and a default (no-op) function will be used.
- Update: The function you specify with the UPDATE_FN clause is called once for each row in the original result set, that is, before any GROUP BY clause is applied. A separate instance of the function is called for each different value returned by the GROUP BY clause. The final argument passed to this function is a pointer, to which you write an updated value based on its original value and the value of the first argument.
- Merge: The function you specify with the MERGE_FN clause is called an arbitrary number of times, to combine intermediate values produced by different nodes or different threads as Impala reads and processes data files in parallel. The final argument passed to this function is a pointer, to which you write an updated value based on its original value and the value of the first argument.
- Serialize: The function you specify with the SERIALIZE_FN clause frees memory allocated to intermediate results. It is required if any memory was allocated by the Allocate function in the Init, Update, or Merge functions, or if the intermediate type contains any pointers. See the *UDA code samples* for details.
- Finalize: The function you specify with the FINALIZE_FN clause does any required teardown for resources acquired by your UDF, such as freeing memory, closing file handles if you explicitly opened any files, and so on. This function is often a stub for simple UDAs. You can omit this clause and a default (no-op) function will be used. It is required in UDAs where the final return type is different than the intermediate type. or if any memory was allocated by the Allocate function in the Init, Update, or Merge functions. See the *UDA code samples* for details.

If you use a consistent naming convention for each of the underlying functions, Impala can automatically determine the names based on the first such clause, so the others are optional.

For end-to-end examples of UDAs, see User-defined functions (UDFs) on page 405.

Complex type considerations:

Currently, Impala UDFs cannot accept arguments or return values of the Impala complex types (STRUCT, ARRAY, or MAP).

Usage notes:

- When authorization is enabled, the CREATE FUNCTION statement requires:
  - The CREATE privilege on the database.
  - The ALL privilege on two URIs where the URIs are:
    - The JAR file on the file system. For example:

      ```
      GRANT ALL ON URI 'file:///path_to_my.jar' TO ROLE my_role;
      ```

    - The JAR on HDFS. For example:

      ```
      GRANT ALL ON URI 'hdfs:///path/to/jar' TO ROLE my_role
      ```
- You can write Impala UDFs in either C++ or Java. C++ UDFs are new to Impala, and are the recommended format for high performance utilizing native code. Java-based UDFs are compatible between Impala and Hive, and are most suited to reusing existing Hive UDFs. (Impala can run Java-based Hive UDFs but not Hive UDAs.)
- Impala 2.5 introduces UDF improvements to persistence for both C++ and Java UDFs, and better compatibility between Impala and Hive for Java UDFs.
- The body of the UDF is represented by a .so or .jar file, which you store in HDFS and the CREATE FUNCTION statement distributes to each Impala node.
- Impala calls the underlying code during SQL statement evaluation, as many times as needed to process all the rows from the result set. All UDFs are assumed to be deterministic, that is, to always return the same result when passed the same argument values. Impala might or might not skip some invocations of a UDF if the result value is already known from a previous call. Therefore, do not rely on the UDF being called a specific number of times, and do not return different result values based on some external factor such as the current time, a random number function, or an external data source that could be updated while an Impala query is in progress.
- The names of the function arguments in the UDF are not significant, only their number, positions, and data types.
- You can overload the same function name by creating multiple versions of the function, each with a different argument signature. For security reasons, you cannot make a UDF with the same name as any built-in function.
- In the UDF code, you represent the function return result as a struct. This struct contains 2 fields. The first field is a boolean representing whether the value is NULL or not. (When this field is true, the return value is interpreted as NULL.) The second field is the same type as the specified function return type, and holds the return value when the function returns something other than NULL.
- In the UDF code, you represent the function arguments as an initial pointer to a UDF context structure, followed by references to zero or more structs, corresponding to each of the arguments. Each struct has the same 2 fields as with the return value, a boolean field representing whether the argument is NULL, and a field of the appropriate type holding any non-NULL argument value.
- For sample code and build instructions for UDFs, see the *UDA code samples*.
- Because the file representing the body of the UDF is stored in HDFS, it is automatically available to all the Impala nodes. You do not need to manually copy any UDF-related files between servers.
- Because Impala currently does not have any ALTER FUNCTION statement, if you need to rename a function, move it to a different database, or change its signature or other properties, issue a DROP FUNCTION statement for the original function followed by a CREATE FUNCTION with the desired properties.
- Because each UDF is associated with a particular database, either issue a USE statement before doing any CREA TE FUNCTION statements, or specify the name of the function as *db_name.function_name*.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

Compatibility:

Impala can run UDFs that were created through Hive, as long as they refer to Impala-compatible data types (not composite or nested column types). Hive can run Java-based UDFs that were created through Impala, but not Impala UDFs written in C++.

The Hive current_user() function cannot be called from a Java UDF through Impala.

Persistence:

In Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new CREATE FUNCTION syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old CREATE FUNCTION syntax do not persist across restarts because they are held in the memory of the `catalogd` daemon. Until you re-create such Java UDFs using the new CREATE FUNCTION syntax, you must reload those Java-based UDFs by running the original CREATE FUNCTION statements again each time you restart the `catalogd` daemon. Prior to Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Examples:

For additional examples of all kinds of user-defined functions, see User-defined functions (UDFs) on page 405.

The following example shows how to take a Java jar file and make all the functions inside one of its classes into UDFs under a single (overloaded) function name in Impala. Each CREATE FUNCTION or DROP FUNCTION statement applies to all the overloaded Java functions with the same name. This example uses the signatureless syntax for CREATE FUNCTION and DROP FUNCTION, which is available in Impala 2.5 and higher.

At the start, the jar file is in the local filesystem. Then it is copied into HDFS, so that it is available for Impala to reference through the CREATE FUNCTION statement and queries that refer to the Impala function name.

```
$ jar -tvf udf-examples-cdh570.jar
     0 Mon Feb 22 04:06:50 PST 2016 META-INF/
   122 Mon Feb 22 04:06:48 PST 2016 META-INF/MANIFEST.MF
     0 Mon Feb 22 04:06:46 PST 2016 com/
     0 Mon Feb 22 04:06:46 PST 2016 com/cloudera/
     0 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/
  2460 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/IncompatibleUdfTe
st.class
   541 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/TestUdfException.cl
ass
  3438 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/JavaUdfTest.class
  5872 Mon Feb 22 04:06:46 PST 2016 com/cloudera/impala/TestUdf.class
...
$ hdfs dfs -put udf-examples-cdh570.jar /user/impala/udfs
$ hdfs dfs -ls /user/impala/udfs
Found 2 items
-rw-r--r--   3 jrussell supergroup        853 2015-10-09 14:05 /user/impala/
udfs/hello_world.jar
-rw-r--r--   3 jrussell supergroup       7366 2016-06-08 14:25 /user/impala/
udfs/udf-examples-cdh570.jar
```

In `impala-shell`, the CREATE FUNCTION refers to the HDFS path of the jar file and the fully qualified class name inside the jar. Each of the functions inside the class becomes an Impala function, each one overloaded under the specified Impala function name.

```
[localhost:21000] > create function testudf location '/user/impala/udfs/udf-
examples-cdh570.jar' symbol='com.cloudera.impala.TestUdf';
[localhost:21000] > show functions;
+-------------+---------------------------------------+-------------+-------
--------+
| return type | signature                             | binary type | is p
ersistent |
+-------------+---------------------------------------+-------------+-------
--------+
| BIGINT      | testudf(BIGINT)                       | JAVA        | true
        |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true
        |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true
        |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN, BOOLEAN)    | JAVA        | true
        |
| DOUBLE      | testudf(DOUBLE)                       | JAVA        | true
        |
| DOUBLE      | testudf(DOUBLE, DOUBLE)               | JAVA        | true
        |
| DOUBLE      | testudf(DOUBLE, DOUBLE, DOUBLE)       | JAVA        | true
        |
| FLOAT       | testudf(FLOAT)                        | JAVA        | true
        |
| FLOAT       | testudf(FLOAT, FLOAT)                 | JAVA        | true
        |
| FLOAT       | testudf(FLOAT, FLOAT, FLOAT)          | JAVA        | true
        |
| INT         | testudf(INT)                          | JAVA        | true
        |
| DOUBLE      | testudf(INT, DOUBLE)                  | JAVA        | true
        |
| INT         | testudf(INT, INT)                     | JAVA        | true
        |
| INT         | testudf(INT, INT, INT)                | JAVA        | true
        |
| SMALLINT    | testudf(SMALLINT)                     | JAVA        | true
        |
| SMALLINT    | testudf(SMALLINT, SMALLINT)           | JAVA        | true
        |
| SMALLINT    | testudf(SMALLINT, SMALLINT, SMALLINT) | JAVA        | true
        |
| STRING      | testudf(STRING)                       | JAVA        | true
        |
| STRING      | testudf(STRING, STRING)               | JAVA        | true
        |
| STRING      | testudf(STRING, STRING, STRING)       | JAVA        | true
        |
| TINYINT     | testudf(TINYINT)                      | JAVA        | true
        |
+-------------+---------------------------------------+-------------+-------
--------+
```

These are all simple functions that return their single arguments, or sum, concatenate, and so on their multiple arguments. Impala determines which overloaded function to use based on the number and types of the arguments.

```
insert into bigint_x values (1), (2), (4), (3);
select testudf(x) from bigint_x;
+-----------------+
| udfs.testudf(x) |
+-----------------+
| 1               |
| 2               |
| 4               |
| 3               |
+-----------------+

insert into int_x values (1), (2), (4), (3);
select testudf(x, x+1, x*x) from int_x;
+-------------------------------+
| udfs.testudf(x, x + 1, x * x) |
+-------------------------------+
| 4                             |
| 9                             |
| 25                            |
| 16                            |
+-------------------------------+
select testudf(x) from string_x;
+-----------------+
| udfs.testudf(x) |
+-----------------+
| one             |
| two             |
| four            |
| three           |
+-----------------+
select testudf(x,x) from string_x;
+--------------------+
| udfs.testudf(x, x) |
+--------------------+
| oneone             |
| twotwo             |
| fourfour           |
| threethree         |
+--------------------+
```

The previous example used the same Impala function name as the name of the class. This example shows how the Impala function name is independent of the underlying Java class or function names. A second CREATE FUNCTION statement results in a set of overloaded functions all named my_func, to go along with the overloaded functions all named testudf.

```
create function my_func location '/user/impala/udfs/udf-examples-cdh570.jar'
  symbol='com.cloudera.impala.TestUdf';

show functions;
+-------------+---------------------------------------+-------------+----
-----------+
| return type | signature                             | binary type | is pe
rsistent |
+-------------+---------------------------------------+-------------+----
-----------+
| BIGINT      | my_func(BIGINT)                       | JAVA        | true
         |
```

```
| BOOLEAN     | my_func(BOOLEAN)                          | JAVA        | true
        |
| BOOLEAN     | my_func(BOOLEAN, BOOLEAN)                 | JAVA        | true
        |
...
| BIGINT      | testudf(BIGINT)                           | JAVA        | true
        |
| BOOLEAN     | testudf(BOOLEAN)                          | JAVA        | true
        |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)                 | JAVA        | true
        |
...
```

The corresponding DROP FUNCTION statement with no signature drops all the overloaded functions with that name.

```
drop function my_func;
show functions;
+-------------+-----------------------------------------+-------------+-------
--------+
| return type | signature                               | binary type | is p
ersistent |
+-------------+-----------------------------------------+-------------+-------
--------+
| BIGINT      | testudf(BIGINT)                         | JAVA        | true
        |
| BOOLEAN     | testudf(BOOLEAN)                        | JAVA        | true
        |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)               | JAVA        | true
        |
...
```

The signatureless CREATE FUNCTION syntax for Java UDFs ensures that the functions shown in this example remain available after the Impala service (specifically, the Catalog Server) are restarted.

Related Information

UDA sample code

CREATE TABLE statement

The CREATE TABLE statement creates a new table with the specified properties.

While creating a table, you can optionally specify aspects such as:

- Whether the table is internal or external.
- The columns and associated data types.
- The columns used for physically partitioning the data.
- The file format for data files.
- The HDFS directory where the data files are located.

Syntax:

The general syntax for creating a table and specifying its columns is as follows:

Explicit column definitions:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  (col_name data_type
    [COMMENT 'col_comment']
    [, ...]
  )
  [PARTITIONED BY (col_name data_type [COMMENT 'col_comment'], ...)]
  [SORT BY ([column [, column ...]])]
```

```
  [COMMENT 'table_comment']
  [ROW FORMAT row_format]
  [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [STORED AS file_format]
  [LOCATION 'hdfs_path']
  [CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED]
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
```

CREATE TABLE AS SELECT:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] db_name.]table_name
    [PARTITIONED BY (col_name[, ...])]
    [SORT BY ([column [, column ...]])]
    [COMMENT 'table_comment']
+   [ROW FORMAT row_format]
    [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
+   [STORED AS ctas_file_format]
    [LOCATION 'hdfs_path']
+     [CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED]
    [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  AS
    select_statement
```

Note: If creating a partitioned table, the partition key columns must be listed last in the SELECT columns list, in the same order as in the PARTITIONED BY clause. Otherwise, you will receive an error about a column name mismatch.

```
primitive_type:
    TINYINT
  | SMALLINT
  | INT
  | BIGINT
  | BOOLEAN
  | FLOAT
  | DOUBLE
  | DECIMAL
  | STRING
  | CHAR
  | VARCHAR
  | TIMESTAMP

complex_type:
    struct_type
  | array_type
  | map_type

struct_type: STRUCT < name : primitive_or_complex_type [COMMENT
 'comment_string'], ... >

array_type: ARRAY < primitive_or_complex_type >

map_type: MAP < primitive_type, primitive_or_complex_type >

row_format:
  DELIMITED [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']]
  [LINES TERMINATED BY 'char']

file_format:
    PARQUET
  | TEXTFILE
  | AVRO
  | SEQUENCEFILE
```

```
  | RCFILE

ctas_file_format:
    PARQUET
  | TEXTFILE
```

Column definitions inferred from data file:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  LIKE PARQUET 'hdfs_path_of_parquet_file'
  [PARTITIONED BY (col_name data_type [COMMENT 'col_comment'], ...)]
  [SORT BY ([column [, column ...]])]
  [COMMENT 'table_comment']
  [ROW FORMAT row_format]
  [WITH SERDEPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
  [STORED AS file_format]
  [LOCATION 'hdfs_path']
  [CACHED IN 'pool_name' [WITH REPLICATION = integer] | UNCACHED]
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
data_type:
    primitive_type
  | array_type
  | map_type
  | struct_type
```

Kudu tables:

```
CREATE TABLE [IF NOT EXISTS] [db_name.]table_name
  (col_name data_type
    [kudu_column_attribute ...]
    [COMMENT 'col_comment']
    [, ...]
    [PRIMARY KEY (col_name[, ...])]
  )
  [PARTITION BY kudu_partition_clause]
  [COMMENT 'table_comment']
  STORED AS KUDU
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
```

Kudu column attributes:

```
  PRIMARY KEY
| [NOT] NULL
| ENCODING codec
| COMPRESSION algorithm
| DEFAULT constant
| BLOCK_SIZE number
```

kudu_partition_clause:

```
kudu_partition_clause ::= [ hash_clause [, ...]] [, range_clause ]

hash_clause ::=
  HASH [ (pk_col [, ...]) ]
    PARTITIONS n

range_clause ::=
  RANGE [ (pk_col [, ...]) ]
  (
    {
```

```
      PARTITION constant_expression range_comparison_operator VAL
UES range_comparison_operator constant_expression
      | PARTITION VALUE = constant_expression_or_tuple
    }
   [, ...]
  )

range_comparison_operator ::= { < | <= }
```

External Kudu tables:

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] [db_name.]table_name
  [COMMENT 'table_comment']
  STORED AS KUDU
  [TBLPROPERTIES ('kudu.table_name'='internal_kudu_name')]
```

CREATE TABLE AS SELECT for Kudu tables:

```
CREATE TABLE [IF NOT EXISTS] db_name.]table_name
  [PRIMARY KEY (col_name[, ...])]
  [PARTITION BY kudu_partition_clause]
  [COMMENT 'table_comment']
  STORED AS KUDU
  [TBLPROPERTIES ('key1'='value1', 'key2'='value2', ...)]
AS
  select_statement
```

Statement type: DDL

Column definitions:

Depending on the form of the CREATE TABLE statement, the column definitions are required or not allowed.

With the CREATE TABLE AS SELECT and CREATE TABLE LIKE syntax, you do not specify the columns at all; the column names and types are derived from the source table, query, or data file.

With the basic CREATE TABLE syntax, you must list one or more columns, its name, type, and optionally a comment, in addition to any columns used as partitioning keys. There is one exception where the column list is not required: when creating an Avro table with the STORED AS AVRO clause, you can omit the list of columns and specify the same metadata as part of the TBLPROPERTIES clause.

Complex type considerations:

The Impala complex types (STRUCT, ARRAY, or MAP) are available in Impala 2.3 and higher. Because you can nest these types (for example, to make an array of maps or a struct with an array field), these types are also sometimes referred to as nested types.

Impala can create tables containing complex type columns, with any supported file format. Because currently Impala can only query complex type columns in Parquet tables, creating tables with complex type columns and other file formats such as text is of limited use. For example, you might create a text table including some columns with complex types with Impala, and use Hive as part of your to ingest the nested type data and copy it to an identical Parquet table. Or you might create a partitioned table containing complex type columns using one file format, and use ALTER TABLE to change the file format of individual partitions to Parquet; Impala can then query only the Parquet-format partitions in that table.

Partitioned tables can contain complex type columns. All the partition key columns must be scalar types.

Internal and external tables (EXTERNAL and LOCATION clauses):

By default, Impala creates an “internal” table, where Impala manages the underlying data files for the table, and physically deletes the data files when you drop the table. If you specify the EXTERNAL clause, Impala treats the table as an “external” table, where the data files are typically produced outside Impala and queried from their original locations in HDFS, and Impala leaves the data files in place when you drop the table.

Typically, for an external table you include a LOCATION clause to specify the path to the HDFS directory where Impala reads and writes files for the table. For example, if your data pipeline produces Parquet files in the HDFS directory /user/etl/destination, you might create an external table as follows:

```
CREATE EXTERNAL TABLE external_parquet (c1 INT, c2 STRING, c3 TIMESTAMP)
  STORED AS PARQUET LOCATION '/user/etl/destination';
```

Although the EXTERNAL and LOCATION clauses are often specified together, LOCATION is optional for external tables, and you can also specify LOCATION for internal tables. The difference is all about whether Impala "takes control" of the underlying data files and moves them when you rename the table, or deletes them when you drop the table. For more about internal and external tables and how they interact with the LOCATION attribute, see *Overview of Impala tables*.

Partitioned tables (PARTITIONED BY clause):

The PARTITIONED BY clause divides the data files based on the values from one or more specified columns. Impala queries can use the partition metadata to minimize the amount of data that is read from disk or transmitted across the network, particularly during join queries.

Note:

All Kudu tables require partitioning, which involves different syntax than non-Kudu tables. See the PART ITION BY clause, rather than PARTITIONED BY, for Kudu tables.

In Impala 2.10 and higher, the PARTITION BY clause is optional for Kudu tables. If the clause is omitted, Impala automatically constructs a single partition that is not connected to any column. Because such a table cannot take advantage of Kudu features for parallelized queries and query optimizations, omitting the PART ITION BY clause is only appropriate for small lookup tables.

Prior to Impala 2.5, you could use a partitioned table as the source and copy data from it, but could not specify any partitioning clauses for the new table. In Impala 2.5 and higher, you can now use the PARTITIONED BY clause with a CREATE TABLE AS SELECT statement. See the examples under the following discussion of the CREATE T ABLE AS SELECT syntax variation.

Sorted tables (SORT BY clause):

The optional SORT BY clause lets you specify zero or more columns that are sorted in the data files created by each Impala INSERT or CREATE TABLE AS SELECT operation. Creating data files that are sorted is most useful for Parquet tables, where the metadata stored inside each file includes the minimum and maximum values for each column in the file. (The statistics apply to each row group within the file; for simplicity, Impala writes a single row group in each file.) Grouping data values together in relatively narrow ranges within each data file makes it possible for Impala to quickly skip over data files that do not contain value ranges indicated in the WHERE clause of a query, and can improve the effectiveness of Parquet encoding and compression.

This clause is not applicable for Kudu tables or HBase tables. Although it works for other HDFS file formats besides Parquet, the more efficient layout is most evident with Parquet tables, because each Parquet data file includes statistics about the data values in that file.

The SORT BY columns cannot include any partition key columns for a partitioned table, because those column values are not represented in the underlying data files.

Because data files can arrive in Impala tables by mechanisms that do not respect the SORT BY clause, such as LOAD DATA or ETL tools that create HDFS files, Impala does not guarantee or rely on the data being sorted. The sorting aspect is only used to create a more efficient layout for Parquet files generated by Impala, which helps to optimize the processing of those Parquet files during Impala queries. During an INSERT or CREATE TABLE AS SELECT operation, the sorting occurs when the SORT BY clause applies to the destination table for the data, regardless of whether the source table has a SORT BY clause.

For example, when creating a table intended to contain census data, you might define sort columns such as last name and state. If a data file in this table contains a narrow range of last names, for example from Smith to Smythe, Impala

can quickly detect that this data file contains no matches for a WHERE clause such as WHERE last_name = 'Jones' and avoid reading the entire file.

```
CREATE TABLE census_data (last_name STRING, first_name STRING, state STRING,
 address STRING)
  SORT BY (last_name, state)
  STORED AS PARQUET;
```

Likewise, if an existing table contains data without any sort order, you can reorganize the data in a more efficient way by using INSERT or CREATE TABLE AS SELECT to copy that data into a new table with a SORT BY clause:

```
CREATE TABLE sorted_census_data
  SORT BY (last_name, state)
  STORED AS PARQUET
  AS SELECT last_name, first_name, state, address
    FROM unsorted_census_data;
```

The metadata for the SORT BY clause is stored in the TBLPROPERTIES fields for the table. Other SQL engines that can interoperate with Impala tables, such as Hive and Spark SQL, do not recognize this property when inserting into a table that has a SORT BY clause.

Transactional tables:

In the version 3.3 and higher, when integrated with Hive 3, Impala can create, read, and insert into transactional tables.

To create a table that supports transactions, use the TBLPROPERTIES clause and set the 'transactional' and 'transac tional_properties' as below. Currently, Impala only supports insert-only transactional tables.

```
TBLPROPERTIES('transactional'='true', 'transactional_properties'='insert_onl
y')
```

When integrated with Hive3 and the DEFAULT_TRANSACTIONAL_TYPE query option is set to INSERT_ONLY, tables are created as insert-only transactional table by default.

Transactional tables are not supported for Kudu and HBase.

Kudu considerations:

Because Kudu tables do not support clauses related to HDFS and S3 data files and partitioning mechanisms, the syntax associated with the STORED AS KUDU clause is shown separately in the above syntax descriptions. Kudu tables have their own syntax for CREATE TABLE, CREATE EXTERNAL TABLE, and CREATE TABLE AS SELECT. Prior to Impala 2.10, all internal Kudu tables require a PARTITION BY clause, different than the PART ITIONED BY clause for HDFS-backed tables.

Here are some examples of creating empty Kudu tables:

```

-- Single partition. Only for  Impala 2.10 and higher.
-- Only suitable for small lookup tables.
CREATE TABLE kudu_no_partition_by_clause
  (
    id bigint PRIMARY KEY, s STRING, b BOOLEAN
  )
  STORED AS KUDU;

-- Single-column primary key.
CREATE TABLE kudu_t1 (id BIGINT PRIMARY key, s STRING, b BOOLEAN)
  PARTITION BY HASH (id) PARTITIONS 20 STORED AS KUDU;

-- Multi-column primary key.
CREATE TABLE kudu_t2 (id BIGINT, s STRING, b BOOLEAN, PRIMARY KEY (id,s))
  PARTITION BY HASH (s) PARTITIONS 30 STORED AS KUDU;
```

```
-- Meaningful primary key column is good for range partitioning.
CREATE TABLE kudu_t3 (id BIGINT, year INT, s STRING,
    b BOOLEAN, PRIMARY KEY (id,year))
  PARTITION BY HASH (id) PARTITIONS 20,
  RANGE (year) (PARTITION 1980 <= VALUES < 1990,
    PARTITION 1990 <= VALUES < 2000,
    PARTITION VALUE = 2001,
    PARTITION 2001 < VALUES)
  STORED AS KUDU;
```

Here is an example of creating an external Kudu table:

```
-- Inherits column definitions from original table.
-- For tables created through Impala, the kudu.table_name property
-- comes from DESCRIBE FORMATTED output from the original table.
CREATE EXTERNAL TABLE external_t1 STORED AS KUDU
  TBLPROPERTIES ('kudu.table_name'='kudu_tbl_created_via_api');
```

Here is an example of CREATE TABLE AS SELECT syntax for a Kudu table:

```
-- The CTAS statement defines the primary key and partitioning scheme.
-- The rest of the column definitions are derived from the select list.
CREATE TABLE ctas_t1
  PRIMARY KEY (id) PARTITION BY HASH (id) PARTITIONS 10
  STORED AS KUDU
  AS SELECT id, s FROM kudu_t1;
```

The following CREATE TABLE clauses are not supported for Kudu tables:

- PARTITIONED BY (Kudu tables use the clause PARTITION BY instead)
- LOCATION
- ROWFORMAT
- CACHED IN | UNCACHED
- WITH SERDEPROPERTIES

Partitioning for Kudu tables (PARTITION BY clause)

For Kudu tables, you specify logical partitioning across one or more columns using the PARTITION BY clause. In contrast to partitioning for HDFS-based tables, multiple values for a partition key column can be located in the same partition. The optional HASH clause lets you divide one or a set of partition key columns into a specified number of buckets. You can use more than one HASH clause, specifying a distinct set of partition key columns for each. The optional RANGE clause further subdivides the partitions, based on a set of comparison operations for the partition key columns.

Here are some examples of the PARTITION BY HASH syntax:

```
-- Apply hash function to 1 primary key column.
create table hash_t1 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (x) partitions 10
  stored as kudu;
-- Apply hash function to a different primary key column.
create table hash_t2 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (y) partitions 10
  stored as kudu;
```

```
-- Apply hash function to both primary key columns.
-- In this case, the total number of partitions is 10.
create table hash_t3 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (x,y) partitions 10
  stored as kudu;

-- When the column list is omitted, apply hash function to all primary key c
olumns.
create table hash_t4 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash partitions 10
  stored as kudu;

-- Hash the X values independently from the Y values.
-- In this case, the total number of partitions is 10 x 20.
create table hash_t5 (x bigint, y bigint, s string, primary key (x,y))
  partition by hash (x) partitions 10, hash (y) partitions 20
  stored as kudu;
```

Here are some examples of the PARTITION BY RANGE syntax:

```
-- Create partitions that cover every possible value of X.
-- Ranges that span multiple values use the keyword VALUES between
-- a pair of < and <= comparisons.
create table range_t1 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (x)
    (
      partition 0 <= values <= 49, partition 50 <= values <= 100,
      partition values < 0, partition 100 < values
    )
  stored as kudu;
-- Create partitions that cover some possible values of X.
-- Values outside the covered range(s) are rejected.
-- New range partitions can be added through ALTER TABLE.
create table range_t2 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (x)
    (
      partition 0 <= values <= 49, partition 50 <= values <= 100
    )
  stored as kudu;

-- A range can also specify a single specific value, using the keyword VALUE
-- with an = comparison.
create table range_t3 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (s)
    (
      partition value = 'Yes', partition value = 'No', partition value = 'M
aybe'
    )
  stored as kudu;

-- Using multiple columns in the RANGE clause and tuples inside the parti
tion spec
-- only works for partitions specified with the VALUE= syntax.
create table range_t4 (x bigint, s string, s2 string, primary key (x, s))
  partition by range (x,s)
    (
      partition value = (0,'zero'), partition value = (1,'one'), partition v
alue = (2,'two')
    )
  stored as kudu;
```

Here are some examples combining both HASH and RANGE syntax for the PARTITION BY clause:

```
-- Values from each range partition are hashed into 10 associated buckets.
-- Total number of partitions in this case is 10 x 2.
create table combined_t1 (x bigint, s string, s2 string, primary key (x, s))
  partition by hash (x) partitions 10, range (x)
    (
      partition 0 <= values <= 49, partition 50 <= values <= 100
    )
  stored as kudu;

-- The hash partitioning and range partitioning can apply to different colum
ns.
-- But all the columns used in either partitioning scheme must be from the p
rimary key.
create table combined_t2 (x bigint, s string, s2 string, primary key (x, s))
  partition by hash (s) partitions 10, range (x)
    (
      partition 0 <= values <= 49, partition 50 <= values <= 100
    )
  stored as kudu;
```

Specifying file format (STORED AS and ROW FORMAT clauses):

The STORED AS clause identifies the format of the underlying data files. Currently, Impala can query more types of file formats than it can create or insert into. Use Hive to perform any create or data load operations that are not currently available in Impala. For example, Impala can create an Avro, SequenceFile, or RCFile table but cannot insert data into it. There are also Impala-specific procedures for using compression with each kind of file format. For details about working with data files of various formats, see *How Impala works with Hadoop fle formats*.

Note: In Impala 1.4.0 and higher, Impala can create Avro tables, which formerly required doing the CREATE TABLE statement in Hive. See *Using the Avro file format with Impala Tables* for details and examples.

By default (when no STORED AS clause is specified), data files in Impala tables are created as text files with Ctrl-A (hex 01) characters as the delimiter. Specify the ROW FORMAT DELIMITED clause to produce or ingest data files that use a different delimiter character such as tab or |, or a different line end character such as carriage return or newline. When specifying delimiter and line end characters with the FIELDS TERMINATED BY and LINES TERMINATED BY clauses, use '\t' for tab, '\n' for newline or linefeed, '\r' for carriage return, and \0 for ASCII nul (hex 00).

The ESCAPED BY clause applies both to text files that you create through an INSERT statement to an Impala TEXT FILE table, and to existing data files that you put into an Impala table directory. (You can ingest existing data files either by creating the table with CREATE EXTERNAL TABLE ... LOCATION, the LOAD DATA statement, or through an HDFS operation such as hdfs dfs -put *file hdfs_path*.) Choose an escape character that is not used anywhere else in the file, and put it in front of each instance of the delimiter character that occurs within a field value. Surrounding field values with quotation marks does not help Impala to parse fields with embedded delimiter characters; the quotation marks are considered to be part of the column value. If you want to use \ as the escape character, specify the clause in `impala-shell` as ESCAPED BY '\\'.

Note: The CREATE TABLE clauses FIELDS TERMINATED BY, ESCAPED BY, and LINES TERMINATED BY have special rules for the string literal used for their argument, because they all require a single character. You can use a regular character surrounded by single or double quotation marks, an octal sequence such as '\054' (representing a comma), or an integer in the range '-127'..'128' (with quotation marks but no backslash), which is interpreted as a single-byte ASCII character. Negative values are subtracted from 256; for example, FIELDS TERMINATED BY '-2' sets the field delimiter to ASCII code 254, the “Icelandic Thorn” character used as a delimiter by some data formats.

Cloning tables (LIKE clause):

To create an empty table with the same columns, comments, and other attributes as another table, use the following variation. The CREATE TABLE ... LIKE form allows a restricted set of clauses, currently only the LOCATION, COMMENT, and STORED AS clauses.

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  LIKE { [db_name.]table_name | PARQUET 'hdfs_path_of_parquet_file' }
  [COMMENT 'table_comment']
  [STORED AS file_format]
  [LOCATION 'hdfs_path']
```

Note:

To clone the structure of a table and transfer data into it in a single operation, use the CREATE TABLE AS SELECT syntax described in the next subsection.

When you clone the structure of an existing table using the CREATE TABLE ... LIKE syntax, the new table keeps the same file format as the original one, so you only need to specify the STORED AS clause if you want to use a different file format, or when specifying a view as the original table. (Creating a table “like” a view produces a text table by default.)

Although normally Impala cannot create an HBase table directly, Impala can clone the structure of an existing HBase table with the CREATE TABLE ... LIKE syntax, preserving the file format and metadata from the original table.

There are some exceptions to the ability to use CREATE TABLE ... LIKE with an Avro table. For example, you cannot use this technique for an Avro table that is specified with an Avro schema but no columns. When in doubt, check if a CREATE TABLE ... LIKE operation works in Hive; if not, it typically will not work in Impala either.

If the original table is partitioned, the new table inherits the same partition key columns. Because the new table is initially empty, it does not inherit the actual partitions that exist in the original one. To create partitions in the new table, insert data or issue ALTER TABLE ... ADD PARTITION statements.

Prior to Impala 1.4.0, it was not possible to use the CREATE TABLE LIKE *view_name* syntax. In Impala 1.4.0 and higher, you can create a table with the same column definitions as a view using the CREATE TABLE LIKE technique. Although CREATE TABLE LIKE normally inherits the file format of the original table, a view has no underlying file format, so CREATE TABLE LIKE *view_name* produces a text table by default. To specify a different file format, include a STORED AS *file_format* clause at the end of the CREATE TABLE LIKE statement.

Because CREATE TABLE ... LIKE only manipulates table metadata, not the physical data of the table, issue INSERT INTO TABLE statements afterward to copy any data from the original table into the new one, optionally converting the data to a new file format. For some file formats, Impala can do a CREATE TABLE ... LIKE to create the table, but Impala cannot insert data in that file format; in these cases, you must load the data in Hive.

CREATE TABLE AS SELECT:

The CREATE TABLE AS SELECT syntax is a shorthand notation to create a table based on column definitions from another table, and copy data from the source table to the destination table without issuing any separate INSERT statement. This idiom is so popular that it has its own acronym, “CTAS”.

The following examples show how to copy data from a source table T1 to a variety of destinations tables, applying various transformations to the table properties, table layout, or the data itself as part of the operation:

```
-- Sample table to be the source of CTAS operations.
CREATE TABLE t1 (x INT, y STRING);
INSERT INTO t1 VALUES (1, 'one'), (2, 'two'), (3, 'three');

-- Clone all the columns and data from one table to another.
CREATE TABLE clone_of_t1 AS SELECT * FROM t1;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+
```

```
-- Clone the columns and data, and convert the data to a different file form
at.
CREATE TABLE parquet_version_of_t1 STORED AS PARQUET AS SELECT * FROM t1;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+
-- Copy only some rows to the new table.
CREATE TABLE subset_of_t1 AS SELECT * FROM t1 WHERE x >= 2;
+-------------------+
| summary           |
+-------------------+
| Inserted 2 row(s) |
+-------------------+
-- Same idea as CREATE TABLE LIKE: clone table layout but do not copy any d
ata.
CREATE TABLE empty_clone_of_t1 AS SELECT * FROM t1 WHERE 1=0;
+-------------------+
| summary           |
+-------------------+
| Inserted 0 row(s) |
+-------------------+

-- Reorder and rename columns and transform the data.
CREATE TABLE t5 AS SELECT upper(y) AS s, x+1 AS a, 'Entirely new column' AS
n FROM t1;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+
SELECT * FROM t5;
+-------+---+---------------------+
| s     | a | n                   |
+-------+---+---------------------+
| ONE   | 2 | Entirely new column |
| TWO   | 3 | Entirely new column |
| THREE | 4 | Entirely new column |
+-------+---+---------------------+
```

See the SELECT statement on page 194 topic for details about query syntax for the SELECT portion of a CREATE TABLE AS SELECT statement.

The newly created table inherits the column names that you select from the original table, which you can override by specifying column aliases in the query. Any column or table comments from the original table are not carried over to the new table.

Note: When using the STORED AS clause with a CREATE TABLE AS SELECT statement, the destination table must be a file format that Impala can write to: currently, text or Parquet. You cannot specify an Avro, SequenceFile, or RCFile table as the destination table for a CTAS operation.

Prior to Impala 2.5 you could use a partitioned table as the source and copy data from it, but could not specify any partitioning clauses for the new table. In Impala 2.5 and higher, you can now use the PARTITIONED BY clause with a CREATE TABLE AS SELECT statement. The following example demonstrates how you can copy data from an unpartitioned table in a CREATE TABLE AS SELECT operation, creating a new partitioned table in the process. The main syntax consideration is the column order in the PARTITIONED BY clause and the select list: the partition key columns must be listed last in the select list, in the same order as in the PARTITIONED BY clause. Therefore, in this

case, the column order in the destination table is different from the source table. You also only specify the column names in the PARTITIONED BY clause, not the data types or column comments.

```
create table partitions_no (year smallint, month tinyint, s string);
insert into partitions_no values (2016, 1, 'January 2016'),
  (2016, 2, 'February 2016'), (2016, 3, 'March 2016');

-- Prove that the source table is not partitioned.
show partitions partitions_no;
ERROR: AnalysisException: Table is not partitioned: ctas_partition_by.partit
ions_no
-- Create new table with partitions based on column values from source tabl
e.
create table partitions_yes partitioned by (year, month)
  as select s, year, month from partitions_no;
+-------------------+
| summary           |
+-------------------+
| Inserted 3 row(s) |
+-------------------+

-- Prove that the destination table is partitioned.
show partitions partitions_yes;
+-------+-------+-------+--------+------+...
| year  | month | #Rows | #Files | Size |...
+-------+-------+-------+--------+------+...
| 2016  | 1     | -1    | 1      | 13B  |...
| 2016  | 2     | -1    | 1      | 14B  |...
| 2016  | 3     | -1    | 1      | 11B  |...
| Total |       | -1    | 3      | 38B  |...
+-------+-------+-------+--------+------+...
```

The most convenient layout for partitioned tables is with all the partition key columns at the end. The CTAS PARTITIONED BY syntax requires that column order in the select list, resulting in that same column order in the destination table.

```
describe partitions_no;
+-------+----------+---------+
| name  | type     | comment |
+-------+----------+---------+
| year  | smallint |         |
| month | tinyint  |         |
| s     | string   |         |
+-------+----------+---------+

-- The CTAS operation forced us to put the partition key columns last.
-- Having those columns last works better with idioms such as SELECT *
-- for partitioned tables.
describe partitions_yes;
+-------+----------+---------+
| name  | type     | comment |
+-------+----------+---------+
| s     | string   |         |
| year  | smallint |         |
| month | tinyint  |         |
+-------+----------+---------+
```

Attempting to use a select list with the partition key columns not at the end results in an error due to a column name mismatch:

```
-- We expect this CTAS to fail because non-key column S
-- comes after key columns YEAR and MONTH in the select list.
create table partitions_maybe partitioned by (year, month)
  as select year, month, s from partitions_no;
ERROR: AnalysisException: Partition column name mismatch: year != month
```

As part of a CTAS operation, you can convert the data to any file format that Impala can write (currently, TEXTFILE and PARQUET). You cannot specify the lower-level properties of a text table, such as the delimiter.

Sorting considerations: Although you can specify an ORDER BY clause in an INSERT ... SELECT statement, any ORDER BY clause is ignored and the results are not necessarily sorted. An INSERT ... SELECT operation potentially creates many different data files, prepared by different executor Impala daemons, and therefore the notion of the data being stored in sorted order is impractical.

CREATE TABLE LIKE PARQUET:

The variation CREATE TABLE ... LIKE PARQUET '*hdfs_path_of_parquet_file*' lets you skip the column definitions of the CREATE TABLE statement. The column names and data types are automatically configured based on the organization of the specified Parquet data file, which must already reside in HDFS. You can use a data file located outside the Impala database directories, or a file from an existing Impala Parquet table; either way, Impala only uses the column definitions from the file and does not use the HDFS location for the LOCATION attribute of the new table. (Although you can also specify the enclosing directory with the LOCATION attribute, to both use the same schema as the data file and point the Impala table at the associated directory for querying.)

The following considerations apply when you use the CREATE TABLE LIKE PARQUET technique:

- Any column comments from the original table are not preserved in the new table. Each column in the new table has a comment stating the low-level Parquet field type used to deduce the appropriate SQL column type.
- If you use a data file from a partitioned Impala table, any partition key columns from the original table are left out of the new table, because they are represented in HDFS directory names rather than stored in the data file. To preserve the partition information, repeat the same PARTITION clause as in the original CREATE TABLE statement.
- The file format of the new table defaults to text, as with other kinds of CREATE TABLE statements. To make the new table also use Parquet format, include the clause STORED AS PARQUET in the CREATE TABLE LIKE PARQUET statement.
- If the Parquet data file comes from an existing Impala table, currently, any TINYINT or SMALLINT columns are turned into INT columns in the new table. Internally, Parquet stores such values as 32-bit integers.
- When the destination table uses the Parquet file format, the CREATE TABLE AS SELECT and INSERT ... SELECT statements always create at least one data file, even if the SELECT part of the statement does not match any rows. You can use such an empty Parquet data file as a template for subsequent CREATE TABLE LIKE PARQUET statements.

For more details about creating Parquet tables, and examples of the CREATE TABLE LIKE PARQUET syntax, see *Using the Parquet file format with Impala tables*.

Visibility and Metadata (TBLPROPERTIES and WITH SERDEPROPERTIES clauses):

You can associate arbitrary items of metadata with a table by specifying the TBLPROPERTIES clause. This clause takes a comma-separated list of key-value pairs and stores those items in the metastore database. You can also change the table properties later with an ALTER TABLE statement. You can observe the table properties for different delimiter and escape characters using the DESCRIBE FORMATTED command, and change those settings for an existing table with ALTER TABLE ... SET TBLPROPERTIES.

You can also associate SerDes properties with the table by specifying key-value pairs through the WITH SERDEPROPERTIES clause. This metadata is not used by Impala, which has its own built-in serializer and deserializer for the file formats it supports. Particular property values might be needed for Hive compatibility with certain variations of file formats, particularly Avro.

Some DDL operations that interact with other Hadoop components require specifying particular values in the SERD EPROPERTIES or TBLPROPERTIES fields, such as creating an Avro table or an HBase table. (You typically create HBase tables in Hive, because they require additional clauses not currently available in Impala.)

To see the column definitions and column comments for an existing table, for example before issuing a CREATE T ABLE ... LIKE or a CREATE TABLE ... AS SELECT statement, issue the statement DESCRIBE *table_name*. To see even more detail, such as the location of data files and the values for clauses such as ROW FORMAT and STOR ED AS, issue the statement DESCRIBE FORMATTED *table_name*. DESCRIBE FORMATTED is also needed to see any overall table comment (as opposed to individual column comments).

After creating a table, your `impala-shell` session or another `impala-shell` connected to the same node can immediately query that table. There might be a brief interval (one statestore heartbeat) before the table can be queried through a different Impala node. To make the CREATE TABLE statement return only when the table is recognized by all Impala nodes in the cluster, enable the SYNC_DDL query option.

HDFS caching (CACHED IN clause):

If you specify the CACHED IN clause, any existing or future data files in the table directory or the partition subdirectories are designated to be loaded into memory with the HDFS caching mechanism.

In Impala 2.2 and higher, the optional WITH REPLICATION clause for CREATE TABLE and ALTER TABLE lets you specify a *replication factor*, the number of hosts on which to cache the same data blocks. When Impala processes a cached data block, where the cache replication factor is greater than 1, Impala randomly selects a host that has a cached copy of that data block. This optimization avoids excessive CPU usage on a single host when the same cached data block is processed multiple times. Cloudera recommends specifying a value greater than or equal to the HDFS block replication factor.

Column order:

If you intend to use the table to hold data files produced by some external source, specify the columns in the same order as they appear in the data files.

If you intend to insert or copy data into the table through Impala, or if you have control over the way externally produced data files are arranged, use your judgment to specify columns in the most convenient order:

- If certain columns are often NULL, specify those columns last. You might produce data files that omit these trailing columns entirely. Impala automatically fills in the NULL values if so.
- If an unpartitioned table will be used as the source for an INSERT ... SELECT operation into a partitioned table, specify last in the unpartitioned table any columns that correspond to partition key columns in the partitioned table, and in the same order as the partition key columns are declared in the partitioned table. This technique lets you use INSERT ... SELECT * when copying data to the partitioned table, rather than specifying each column name individually.
- If you specify columns in an order that you later discover is suboptimal, you can sometimes work around the problem without recreating the table. You can create a view that selects columns from the original table in a permuted order, then do a SELECT * from the view. When inserting data into a table, you can specify a permuted order for the inserted columns to match the order in the destination table.

Hive considerations:

Impala queries can make use of metadata about the table and columns, such as the number of rows in a table or the number of different values in a column. Prior to Impala 1.2.2, to create this metadata, you issued the ANALYZE TABLE statement in Hive to gather this information, after creating the table and loading representative data into it. In Impala 1.2.2 and higher, the COMPUTE STATS statement produces these statistics within Impala, without needing to use Hive at all.

HBase considerations:

Note:

The Impala CREATE TABLE statement cannot create an HBase table, because it currently does not support the STORED BY clause needed for HBase tables. Create such tables in Hive, then query them through Impala.

Amazon S3 considerations:

To create a table where the data resides in the Amazon Simple Storage Service (S3), specify a s3a:// prefix LOCATION attribute pointing to the data files in S3.

In Impala 2.6 and higher, you can use this special LOCATION syntax as part of a CREATE TABLE AS SELECT statement.

In Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DATABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed.

Sorting considerations: Although you can specify an ORDER BY clause in an INSERT ... SELECT statement, any ORDER BY clause is ignored and the results are not necessarily sorted. An INSERT ... SELECT operation potentially creates many different data files, prepared by different executor Impala daemons, and therefore the notion of the data being stored in sorted order is impractical.

HDFS considerations:

The CREATE TABLE statement for an internal table creates a directory in HDFS. The CREATE EXTERNAL TABLE statement associates the table with an existing HDFS directory, and does not create any new directory in HDFS. To locate the HDFS data directory for a table, issue a DESCRIBE FORMATTED *table* statement. To examine the contents of that HDFS directory, use an OS command such as hdfs dfs -ls hdfs://*path*, either from the OS command line or through the shell or ! commands in `impala-shell`.

The CREATE TABLE AS SELECT syntax creates data files under the table data directory to hold any data copied by the INSERT portion of the statement. (Even if no data is copied, Impala might create one or more empty data files.)

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have both execute and write permission for the database directory where the table is being created.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts.

Cancellation: Certain multi-stage statements (CREATE TABLE AS SELECT and COMPUTE STATS) can be cancelled during some stages, when running INSERT or SELECT operations internally. To cancel this statement, use Ctrl-C from the `impala-shell` interpreter, the Cancel button from the Watch page in Hue, Actions > Cancel from the Queries list in Cloudera Manager, or Cancel from the list of in-flight queries (for a particular node) on the Queries tab in the Impala web UI (port 25000).

Related Information

Impala tables
Partitioning
Hadoop file formats supported
Impala with Amazon S3
SQL transactions in Impala

CREATE VIEW statement

The CREATE VIEW statement lets you create a shorthand abbreviation for a more complicated query. The base query can involve joins, expressions, reordered columns, column aliases, and other SQL features that can make a query hard to understand or maintain.

Because a view is purely a logical construct (an alias for a query) with no physical data behind it, ALTER VIEW only involves changes to metadata in the metastore database, not any data files in HDFS.

Syntax:

```
CREATE VIEW [IF NOT EXISTS] view_name
    [(column_name [COMMENT 'column_comment'][, ...])]
    [COMMENT 'view_comment']
  AS select_statement
```

Statement type: DDL

Usage notes:

The CREATE VIEW statement can be useful in scenarios such as the following:

- To turn even the most lengthy and complicated SQL query into a one-liner. You can issue simple queries against the view from applications, scripts, or interactive queries in `impala-shell`. For example:

  ```
  select * from view_name;
  select * from view_name order by c1 desc limit 10;
  ```

  The more complicated and hard-to-read the original query, the more benefit there is to simplifying the query using a view.
- To hide the underlying table and column names, to minimize maintenance problems if those names change. In that case, you re-create the view using the new names, and all queries that use the view rather than the underlying tables keep running with no changes.
- To experiment with optimization techniques and make the optimized queries available to all applications. For example, if you find a combination of WHERE conditions, join order, join hints, and so on that works the best for a class of queries, you can establish a view that incorporates the best-performing techniques. Applications can then make relatively simple queries against the view, without repeating the complicated and optimized logic over and over. If you later find a better way to optimize the original query, when you re-create the view, all the applications immediately take advantage of the optimized base query.
- To simplify a whole class of related queries, especially complicated queries involving joins between multiple tables, complicated expressions in the column list, and other SQL syntax that makes the query difficult to understand and debug. For example, you might create a view that joins several tables, filters using several WHERE conditions, and selects several columns from the result set. Applications might issue queries against this view that only vary in their LIMIT, ORDER BY, and similar simple clauses.

For queries that require repeating complicated clauses over and over again, for example in the select list, ORDER BY, and GROUP BY clauses, you can use the WITH clause as an alternative to creating a view.

You can optionally specify the table-level and the column-level comments as in the CREATE TABLE statement.

Complex type considerations:

For tables containing complex type columns (ARRAY, STRUCT, or MAP), you typically use join queries to refer to the complex values. You can use views to hide the join notation, making such tables seem like traditional denormalized tables, and making those tables queryable by business intelligence tools that do not have built-in support for those complex types.

Because you cannot directly issue SELECT *col_name* against a column of complex type, you cannot use a view or a WITH clause to “rename” a column by selecting it with a column alias.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Examples:

```
-- Create a view that is exactly the same as the underlying table.
CREATE VIEW v1 AS SELECT * FROM t1;

-- Create a view that includes only certain columns from the underlying tabl
e.
CREATE VIEW v2 AS SELECT c1, c3, c7 FROM t1;
-- Create a view that filters the values from the underlying table.
CREATE VIEW v3 AS SELECT DISTINCT c1, c3, c7 FROM t1 WHERE c1 IS NOT NULL A
ND c5 > 0;

-- Create a view that that reorders and renames columns from the underlying
table.
CREATE VIEW v4 AS SELECT c4 AS last_name, c6 AS address, c2 AS birth_date
 FROM t1;

-- Create a view that runs functions to convert or transform certain colu
mns.
CREATE VIEW v5 AS SELECT c1, CAST(c3 AS STRING) c3, CONCAT(c4,c5) c5, TRIM(
c6) c6, "Constant" c8 FROM t1;

-- Create a view that hides the complexity of a view query.
CREATE VIEW v6 AS SELECT t1.c1, t2.c2 FROM t1 JOIN t2 ON t1.id = t2.id;
-- Create a view with a column comment and a table comment.
CREATE VIEW v7 (c1 COMMENT 'Comment for c1', c2) COMMENT 'Comment for v7' AS
 SELECT t1.c1, t1.c2 FROM t1;
```

DELETE statement

The DELETE statement deletes an arbitrary number of rows from a Kudu table. This statement only works for Impala tables that use the Kudu storage engine.

Syntax:

```
DELETE [FROM] [database_name.]table_name [ WHERE where_conditions ]

DELETE table_ref FROM [joined_table_refs] [ WHERE where_conditions ]
```

The first form evaluates rows from one table against an optional WHERE clause, and deletes all the rows that match the WHERE conditions, or all rows if WHERE is omitted.

The second form evaluates one or more join clauses, and deletes all matching rows from one of the tables. The join clauses can include non-Kudu tables, but the table from which the rows are deleted must be a Kudu table. The FROM keyword is required in this case, to separate the name of the table whose rows are being deleted from the table names of the join clauses.

Usage notes:

The conditions in the WHERE clause are the same ones allowed for the SELECT statement.

The conditions in the WHERE clause can refer to any combination of primary key columns or other columns. Referring to primary key columns in the WHERE clause is more efficient than referring to non-primary key columns.

If the WHERE clause is omitted, all rows are removed from the table.

Because Kudu currently does not enforce strong consistency during concurrent DML operations, be aware that the results after this statement finishes might be different than you intuitively expect:

- If some rows cannot be deleted because their some primary key columns are not found, due to their being deleted by a concurrent DELETE operation, the statement succeeds but returns a warning.
- A DELETE statement might also overlap with INSERT, UPDATE, or UPSERT statements running concurrently on the same table. After the statement finishes, there might be more or fewer rows than expected in the table because it is undefined whether the DELETE applies to rows that are inserted or updated while the DELETE is in progress.

The number of affected rows is reported in an `impala-shell` message and in the query profile.

Statement type: DML

Important: After adding or replacing data in a table used in performance-critical queries, issue a COMP UTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following examples show how to delete rows from a specified table, either all rows or rows that match a WHERE clause:

```
-- Deletes all rows. The FROM keyword is optional.
DELETE FROM kudu_table;
DELETE kudu_table;

-- Deletes 0, 1, or more rows.
-- (If c1 is a single-column primary key, the statement could only
-- delete 0 or 1 rows.)
DELETE FROM kudu_table WHERE c1 = 100;

-- Deletes all rows that match all the WHERE conditions.
DELETE FROM kudu_table WHERE
  (c1 > c2 OR c3 IN ('hello','world')) AND c4 IS NOT NULL;
DELETE FROM t1 WHERE
  (c1 IN (1,2,3) AND c2 > c3) OR c4 IS NOT NULL;
DELETE FROM time_series WHERE
  year = 2016 AND month IN (11,12) AND day > 15;
-- WHERE condition with a subquery.
DELETE FROM t1 WHERE
  c5 IN (SELECT DISTINCT other_col FROM other_table);
-- Does not delete any rows, because the WHERE condition is always false.
DELETE FROM kudu_table WHERE 1 = 0;
```

The following examples show how to delete rows that are part of the result set from a join:

```
-- Remove _all_ rows from t1 that have a matching X value in t2.
DELETE t1 FROM t1 JOIN t2 ON t1.x = t2.x;
-- Remove _some_ rows from t1 that have a matching X value in t2.
DELETE t1 FROM t1 JOIN t2 ON t1.x = t2.x
  WHERE t1.y = FALSE and t2.z > 100;

-- Delete from a Kudu table based on a join with a non-Kudu table.
DELETE t1 FROM kudu_table t1 JOIN non_kudu_table t2 ON t1.x = t2.x;

-- The tables can be joined in any order as long as the Kudu table
-- is specified as the deletion target.
DELETE t2 FROM non_kudu_table t1 JOIN kudu_table t2 ON t1.x = t2.x;
```

Related Information

SELECT statement

DESCRIBE statement

The DESCRIBE statement displays metadata about a table, such as the column names and their data types. In Impala 2.3 and higher, you can specify the name of a complex type column, which takes the form of a dotted path. The path might include multiple components in the case of a nested type definition. In Impala 2.5 and higher, the DESCRIBE DATABASE form can display information about a database.

Syntax:

```
DESCRIBE [DATABASE] [FORMATTED|EXTENDED] object_name

object_name ::=
    [db_name.]table_name[.complex_col_name ...]
  | db_name
```

You can use the abbreviation DESC for the DESCRIBE statement.

The DESCRIBE FORMATTED variation displays additional information, in a format familiar to users of Apache Hive. The extra information includes low-level details such as whether the table is internal or external, when it was created, the file format, the location of the data in HDFS, whether the object is a table or a view, and (for views) the text of the query from the view definition.

Note: The Compressed field is not a reliable indicator of whether the table contains compressed data. It typically always shows No, because the compression settings only apply during the session that loads data and are not stored persistently with the table metadata.

Describing databases:

By default, the DESCRIBE output for a database includes the location and the comment, which can be set by the LOCATION and COMMENT clauses on the CREATE DATABASE statement.

The additional information displayed by the FORMATTED or EXTENDED keyword includes the HDFS user ID that is considered the owner of the database, and any optional database properties. The properties could be specified by the WITH DBPROPERTIES clause if the database is created using a Hive CREATE DATABASE statement. Impala currently does not set or do any special processing based on those properties.

The following examples show the variations in syntax and output for describing databases. This feature is available in Impala 2.5 and higher.

```
describe database default;
+---------+----------------------+-----------------------+
| name    | location             | comment               |
+---------+----------------------+-----------------------+
| default | /user/hive/warehouse | Default Hive database |
+---------+----------------------+-----------------------+

describe database formatted default;
+---------+----------------------+-----------------------+
| name    | location             | comment               |
+---------+----------------------+-----------------------+
| default | /user/hive/warehouse | Default Hive database |
| Owner:  |                      |                       |
|         | public               | ROLE                  |
+---------+----------------------+-----------------------+

describe database extended default;
+---------+----------------------+-----------------------+
| name    | location             | comment               |
+---------+----------------------+-----------------------+
```

```
| default | /user/hive/warehouse | Default Hive database |
| Owner:  |                      |                       |
|         | public               | ROLE                  |
+---------+----------------------+-----------------------+
```

Describing tables:

If the DATABASE keyword is omitted, the default for the DESCRIBE statement is to refer to a table.

If you have the SELECT privilege on a subset of the table columns and no other relevant table/database/server-level privileges, DESCRIBE returns the data from the columns you have access to.

If you have the SELECT privilege on a subset of the table columns and no other relevant table/database/server-level privileges, DESCRIBE FORMATTED/EXTENDED does not return the LOCATION field. The LOCATION data is shown if you have any privilege on the table, the containing database or the server.

```
-- By default, the table is assumed to be in the current database.
describe my_table;
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| s    | string |         |
+------+--------+---------+

-- Use a fully qualified table name to specify a table in any database.
describe my_database.my_table;
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| s    | string |         |
+------+--------+---------+

-- The formatted or extended output includes additional useful information.
-- The LOCATION field is especially useful to know for DDL statements and HD
FS commands
-- during ETL jobs. (The LOCATION includes a full hdfs:// URL, omitted here
for readability.)
describe formatted my_table;
+------------------------------+----------------------------------------
-----+----------------------+
| name                         | type
  | comment              |
+------------------------------+----------------------------------------
---+----------------------+
| # col_name                   | data_type
  | comment              |
|                              | NULL
   | NULL                  |
| x                            | int
  | NULL                 |
| s                            | string
  | NULL                 |
|                              | NULL
  | NULL                 |
| # Detailed Table Information | NULL
   | NULL                  |
| Database:                    | my_database
  | NULL                 |
| Owner:                       | jrussell
  | NULL                 |
```

```
| CreateTime:                   | Fri Mar 18 15:58:00 PDT 2016
  | NULL                  |
| LastAccessTime:               | UNKNOWN
   | NULL                  |
| Protect Mode:                 | None
  | NULL                  |
| Retention:                    | 0
  | NULL                  |
| Location:                     | /user/hive/warehouse/my_database.db/my_tabl
e | NULL                  |
| Table Type:                   | MANAGED_TABLE
   | NULL                  |
| Table Parameters:             | NULL
  | NULL                  |
|                               | transient_lastDdlTime
  | 1458341880            |
|                               | NULL
  | NULL                  |
| # Storage Information         | NULL
   | NULL                  |
| SerDe Library:                | org. ... .LazySimpleSerDe
  | NULL                  |
| InputFormat:                  | org.apache.hadoop.mapred.TextInputFormat
  | NULL                  |
| OutputFormat:                 | org. ... .HiveIgnoreKeyTextOutputFormat
  | NULL                  |
| Compressed:                   | No
   | NULL                  |
| Num Buckets:                  | 0
  | NULL                  |
| Bucket Columns:               | []
  | NULL                  |
| Sort Columns:                 | []
  | NULL                  |
+-------------------------------+-----------------------------------------
-----+----------------------+
```

Complex type considerations:

Because the column definitions for complex types can become long, particularly when such types are nested, the DESCRIBE statement uses special formatting for complex type columns to make the output readable.

For the ARRAY, STRUCT, and MAP types available in Impala 2.3 and higher, the DESCRIBE output is formatted to avoid excessively long lines for multiple fields within a STRUCT, or a nested sequence of complex types.

You can pass a multi-part qualified name to DESCRIBE to specify an ARRAY, STRUCT, or MAP column and visualize its structure as if it were a table. For example, if table T1 contains an ARRAY column A1, you could issue the statement DESCRIBE t1.a1. If table T1 contained a STRUCT column S1, and a field F1 within the STRUCT was a MAP, you could issue the statement DESCRIBE t1.s1.f1. An ARRAY is shown as a two-column table, with ITEM and POS columns. A STRUCT is shown as a table with each field representing a column in the table. A MAP is shown as a two-column table, with KEY and VALUE columns.

For example, here is the DESCRIBE output for a table containing a single top-level column of each complex type:

```
create table t1 (x int, a array<int>, s struct<f1: string, f2: bigint>, m ma
p<string,int>) stored as parquet;

describe t1;
+------+-----------------+---------+
| name | type            | comment |
+------+-----------------+---------+
| x    | int             |         |
| a    | array<int>      |         |
```

```
| s    | struct<          |         |
|      |   f1:string,     |         |
|      |   f2:bigint      |         |
|      | >                |         |
| m    | map<string,int>  |         |
+------+------------------+---------+
```

Here are examples showing how to “drill down” into the layouts of complex types, including using multi-part names to examine the definitions of nested types. The < > delimiters identify the columns with complex types; these are the columns where you can descend another level to see the parts that make up the complex type. This technique helps you to understand the multi-part names you use as table references in queries involving complex types, and the corresponding column names you refer to in the SELECT list. These tables are from the “nested TPC-H” schema, shown in detail in *Sample schema and data for experimenting with Impala complex types*.

The REGION table contains an ARRAY of STRUCT elements:

- The first DESCRIBE specifies the table name, to display the definition of each top-level column.
- The second DESCRIBE specifies the name of a complex column, REGION.R_NATIONS, showing that when you include the name of an ARRAY column in a FROM clause, that table reference acts like a two-column table with columns ITEM and POS.
- The final DESCRIBE specifies the fully qualified name of the ITEM field, to display the layout of its underlying STRUCT type in table format, with the fields mapped to column names.

```
-- #1: The overall layout of the entire table.
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

-- #2: The ARRAY column within the table.
describe region.r_nations;
+------+-------------------------+---------+
| name | type                    | comment |
+------+-------------------------+---------+
| item | struct<                 |         |
|      |   n_nationkey:smallint, |         |
|      |   n_name:string,        |         |
|      |   n_comment:string      |         |
|      | >                       |         |
| pos  | bigint                  |         |
+------+-------------------------+---------+

-- #3: The STRUCT that makes up each ARRAY element.
--     The fields of the STRUCT act like columns of a table.
describe region.r_nations.item;
+-------------+----------+---------+
| name        | type     | comment |
+-------------+----------+---------+
| n_nationkey | smallint |         |
| n_name      | string   |         |
| n_comment   | string   |         |
```

```
+-------------+----------+---------+
```

The CUSTOMER table contains an ARRAY of STRUCT elements, where one field in the STRUCT is another ARRAY of STRUCT elements:

- Again, the initial DESCRIBE specifies only the table name.
- The second DESCRIBE specifies the qualified name of the complex column, CUSTOMER.C_ORDERS, showing how an ARRAY is represented as a two-column table with columns ITEM and POS.
- The third DESCRIBE specifies the qualified name of the ITEM of the ARRAY column, to see the structure of the nested ARRAY. Again, it has has two parts, ITEM and POS. Because the ARRAY contains a STRUCT, the layout of the STRUCT is shown.
- The fourth and fifth DESCRIBE statements drill down into a STRUCT field that is itself a complex type, an ARRAY of STRUCT. The ITEM portion of the qualified name is only required when the ARRAY elements are anonymous. The fields of the STRUCT give names to any other complex types nested inside the STRUCT. Therefore, the DESCRIBE parameters CUSTOMER.C_ORDERS.ITEM.O_LINEITEMS and CUSTOMER.C_ORDERS.O_LINEITEMS are equivalent. (For brevity, leave out the ITEM portion of a qualified name when it is not required.)
- The final DESCRIBE shows the layout of the deeply nested STRUCT type. Because there are no more complex types nested inside this STRUCT, this is as far as you can drill down into the layout for this table.

```
-- #1: The overall layout of the entire table.
describe customer;
+--------------+-----------------------------------+
| name         | type                              |
+--------------+-----------------------------------+
| c_custkey    | bigint                            |
... more scalar columns ...
| c_orders     | array<struct<                     |
|              |   o_orderkey:bigint,              |
|              |   o_orderstatus:string,           |
|              |   o_totalprice:decimal(12,2),     |
|              |   o_orderdate:string,             |
|              |   o_orderpriority:string,         |
|              |   o_clerk:string,                 |
|              |   o_shippriority:int,             |
|              |   o_comment:string,               |
|              |   o_lineitems:array<struct<       |
|              |     l_partkey:bigint,             |
|              |     l_suppkey:bigint,             |
|              |     l_linenumber:int,             |
|              |     l_quantity:decimal(12,2),     |
|              |     l_extendedprice:decimal(12,2), |
|              |     l_discount:decimal(12,2),     |
|              |     l_tax:decimal(12,2),          |
|              |     l_returnflag:string,          |
|              |     l_linestatus:string,          |
|              |     l_shipdate:string,            |
|              |     l_commitdate:string,          |
|              |     l_receiptdate:string,         |
|              |     l_shipinstruct:string,        |
|              |     l_shipmode:string,            |
|              |     l_comment:string              |
|              |   >>                              |
|              | >>                                |
+--------------+-----------------------------------+

-- #2: The ARRAY column within the table.
describe customer.c_orders;
+------+-----------------------------------+
| name | type                              |
```

```
+------+-----------------------------------+
| item | struct<                           |
|      |   o_orderkey:bigint,              |
|      |   o_orderstatus:string,           |
... more struct fields ...
|      |   o_lineitems:array<struct<       |
|      |     l_partkey:bigint,             |
|      |     l_suppkey:bigint,             |
... more nested struct fields ...
|      |     l_comment:string              |
|      |   >>                              |
|      | >                                 |
| pos  | bigint                            |
+------+-----------------------------------+

-- #3: The STRUCT that makes up each ARRAY element.
--     The fields of the STRUCT act like columns of a table.
describe customer.c_orders.item;
+-----------------+----------------------------------+
| name            | type                             |
+-----------------+----------------------------------+
| o_orderkey      | bigint                           |
| o_orderstatus   | string                           |
| o_totalprice    | decimal(12,2)                    |
| o_orderdate     | string                           |
| o_orderpriority | string                           |
| o_clerk         | string                           |
| o_shippriority  | int                              |
| o_comment       | string                           |
| o_lineitems     | array<struct<                    |
|                 |   l_partkey:bigint,              |
|                 |   l_suppkey:bigint,              |
... more struct fields ...
|                 |   l_comment:string               |
|                 | >>                               |
+-----------------+----------------------------------+

-- #4: The ARRAY nested inside the STRUCT elements of the first ARRAY.
describe customer.c_orders.item.o_lineitems;
+------+-----------------------------------+
| name | type                              |
+------+-----------------------------------+
| item | struct<                           |
|      |   l_partkey:bigint,               |
|      |   l_suppkey:bigint,               |
... more struct fields ...
|      |   l_comment:string                |
|      | >                                 |
| pos  | bigint                            |
+------+-----------------------------------+

-- #5: Shorter form of the previous DESCRIBE. Omits the .ITEM portion of the
 name
--     because O_LINEITEMS and other field names provide a way to refer to
 things
--     inside the ARRAY element.
describe customer.c_orders.o_lineitems;
+------+-----------------------------------+
| name | type                              |
+------+-----------------------------------+
| item | struct<                           |
|      |   l_partkey:bigint,               |
|      |   l_suppkey:bigint,               |
... more struct fields ...
```

```
|      |   l_comment:string                |
|      | >                                 |
| pos  | bigint                            |
+------+-----------------------------------+
-- #6: The STRUCT representing ARRAY elements nested inside
--     another ARRAY of STRUCTs. The lack of any complex types
--     in this output means this is as far as DESCRIBE can
--     descend into the table layout.
describe customer.c_orders.o_lineitems.item;
+-----------------+---------------+
| name            | type          |
+-----------------+---------------+
| l_partkey       | bigint        |
| l_suppkey       | bigint        |
... more scalar columns ...
| l_comment       | string        |
+-----------------+---------------+
```

Usage notes:

After the `impalad` daemons are restarted, the first query against a table can take longer than subsequent queries, because the metadata for the table is loaded before the query is processed. This one-time delay for each table can cause misleading results in benchmark tests or cause unnecessary concern. To “warm up” the Impala metadata cache, you can issue a DESCRIBE statement in advance for each table you intend to access later.

When you are dealing with data files stored in HDFS, sometimes it is important to know details such as the path of the data files for an Impala table, and the hostname for the namenode. You can get this information from the DESC RIBE FORMATTED output. You specify HDFS URIs or path specifications with statements such as LOAD DATA and the LOCATION clause of CREATE TABLE or ALTER TABLE. You might also use HDFS URIs or paths with Linux commands such as `hadoop` and `hdfs` to copy, rename, and so on, data files in HDFS.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

Each table can also have associated table statistics and column statistics. To see these categories of information, use the SHOW TABLE STATS *table_name* and SHOW COLUMN STATS *table_name* statements. See the *SHOW statement* topic for details.

Important: After adding or replacing data in a table used in performance-critical queries, issue a COMP UTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following example shows the results of both a standard DESCRIBE and DESCRIBE FORMATTED for different kinds of schema objects:

- DESCRIBE for a table or a view returns the name, type, and comment for each of the columns. For a view, if the column value is computed by an expression, the column name is automatically generated as _c0, _c1, and so on depending on the ordinal number of the column.
- A table created with no special format or storage clauses is designated as a MANAGED_TABLE (an “internal table” in Impala terminology). Its data files are stored in an HDFS directory under the default Hive data directory. By default, it uses Text data format.
- A view is designated as VIRTUAL_VIEW in DESCRIBE FORMATTED output. Some of its properties are NULL or blank because they are inherited from the base table. The text of the query that defines the view is part of the DESCRIBE FORMATTED output.

- A table with additional clauses in the CREATE TABLE statement has differences in DESCRIBE FORMATTED output. The output for T2 includes the EXTERNAL_TABLE keyword because of the CREATE EXTERNAL TABLE syntax, and different InputFormat and OutputFormat fields to reflect the Parquet file format.

```
[localhost:21000] > create table t1 (x int, y int, s string);
Query: create table t1 (x int, y int, s string)
[localhost:21000] > describe t1;
Query: describe t1
Query finished, fetching results ...
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| y    | int    |         |
| s    | string |         |
+------+--------+---------+
Returned 3 row(s) in 0.13s
[localhost:21000] > describe formatted t1;
Query: describe formatted t1
Query finished, fetching results ...
+------------------------------+----------------------------------------------
+------------+
| name                         | type
 | comment    |
+------------------------------+----------------------------------------------
+------------+
| # col_name                   | data_type
| comment    |
|                              | NULL
 | NULL       |
| x                            | int
 | None       |
| y                            | int
 | None       |
| s                            | string
| None       |
|                              | NULL
 | NULL       |
| # Detailed Table Information | NULL
 | NULL       |
| Database:                    | describe_formatted
 | NULL       |
| Owner:                       | doc_demo
| NULL       |
| CreateTime:                  | Mon Jul 22 17:03:16 EDT 2013
 | NULL       |
| LastAccessTime:              | UNKNOWN
 | NULL       |
| Protect Mode:                | None
 | NULL       |
| Retention:                   | 0
| NULL       |
| Location:                    | hdfs://127.0.0.1:8020/user/hive/warehouse/
 |            |
|                              |   describe_formatted.db/t1
 | NULL       |
| Table Type:                  | MANAGED_TABLE
 | NULL       |
| Table Parameters:            | NULL
| NULL       |
|                              | transient_lastDdlTime
 | 1374526996 |
```

```
|                               | NULL
 | NULL       |
| # Storage Information        | NULL
 | NULL       |
| SerDe Library:               | org.apache.hadoop.hive.serde2.lazy.
|            |
|                               |   LazySimpleSerDe
 | NULL       |
| InputFormat:                 | org.apache.hadoop.mapred.TextInputFormat
 | NULL       |
| OutputFormat:                | org.apache.hadoop.hive.ql.io.
 |            |
|                               |   HiveIgnoreKeyTextOutputFormat
| NULL       |
| Compressed:                  | No
 | NULL       |
| Num Buckets:                 | 0
 | NULL       |
| Bucket Columns:              | []
 | NULL       |
| Sort Columns:                | []
| NULL       |
+------------------------------+-----------------------------------------
+------------+
Returned 26 row(s) in 0.03s
[localhost:21000] > create view v1 as select x, upper(s) from t1;
Query: create view v1 as select x, upper(s) from t1
[localhost:21000] > describe v1;
Query: describe v1
Query finished, fetching results ...
+------+--------+---------+
| name | type   | comment |
+------+--------+---------+
| x    | int    |         |
| _c1  | string |         |
+------+--------+---------+
Returned 2 row(s) in 0.10s
[localhost:21000] > describe formatted v1;
Query: describe formatted v1
Query finished, fetching results ...
+------------------------------+------------------------------+------------
----------+
| name                         | type                         | comment
        |
+------------------------------+------------------------------+----------
------------+
| # col_name                   | data_type                    | comment
        |
|                              | NULL                         | NULL
        |
| x                            | int                          | None
        |
| _c1                          | string                       | None
         |
|                              | NULL                         | NULL
        |
| # Detailed Table Information | NULL                         | NULL
        |
| Database:                    | describe_formatted           | NULL
        |
| Owner:                       | doc_demo                     | NULL
         |
| CreateTime:                  | Mon Jul 22 16:56:38 EDT 2013 | NULL
        |
```

```
| LastAccessTime:               | UNKNOWN                       | NULL
        |
| Protect Mode:                 | None                          | NULL
        |
| Retention:                    | 0                             | NULL
         |
| Table Type:                   | VIRTUAL_VIEW                  | NULL
        |
| Table Parameters:             | NULL                          | NULL
        |
|                               | transient_lastDdlTime         | 1374526598
        |
|                               | NULL                          | NULL
         |
| # Storage Information         | NULL                          | NULL
        |
| SerDe Library:                | null                          | NULL
        |
| InputFormat:                  | null                          | NULL
        |
| OutputFormat:                 | null                          | NULL
         |
| Compressed:                   | No                            | NULL
        |
| Num Buckets:                  | 0                             | NULL
        |
| Bucket Columns:               | []                            | NULL
        |
| Sort Columns:                 | []                            | NULL
         |
|                               | NULL                          | NULL
        |
| # View Information            | NULL                          | NULL
        |
| View Original Text:           | SELECT x, upper(s) FROM t1    | NULL
        |
| View Expanded Text:           | SELECT x, upper(s) FROM t1    | NULL
         |
+-------------------------------+-------------------------------+-----------
-----------+
Returned 28 row(s) in 0.03s
[localhost:21000] > create external table t2 (x int, y int, s string) stored
 as parquet location '/user/doc_demo/sample_data';
[localhost:21000] > describe formatted t2;
Query: describe formatted t2
Query finished, fetching results ...
+-------------------------------+-----------------------------------------
---------+------------+
| name                          | type
        | comment    |
+-------------------------------+-----------------------------------------
-----------+------------+
| # col_name                    | data_type
        | comment    |
|                               | NULL
        | NULL       |
| x                             | int
        | None       |
| y                             | int
         | None       |
| s                             | string
        | None       |
|                               | NULL
        | NULL       |
```

```
| # Detailed Table Information | NULL
        | NULL       |
| Database:                   | describe_formatted
         | NULL       |
| Owner:                      | doc_demo
        | NULL       |
| CreateTime:                 | Mon Jul 22 17:01:47 EDT 2013
        | NULL       |
| LastAccessTime:             | UNKNOWN
        | NULL       |
| Protect Mode:               | None
         | NULL       |
| Retention:                  | 0
        | NULL       |
| Location:                   | hdfs://127.0.0.1:8020/user/doc_demo/sample
_data    | NULL       |
| Table Type:                 | EXTERNAL_TABLE
        | NULL       |
| Table Parameters:           | NULL
         | NULL       |
|                             | EXTERNAL
        | TRUE       |
|                             | transient_lastDdlTime
        | 1374526907 |
|                             | NULL
        | NULL       |
| # Storage Information        | NULL
         | NULL       |
| SerDe Library:              | org.apache.hadoop.hive.serde2.lazy.LazySi
mpleSerDe | NULL       |
| InputFormat:                | com.cloudera.impala.hive.serde.ParquetInpu
tFormat  | NULL       |
| OutputFormat:               | com.cloudera.impala.hive.serde.ParquetOutpu
tFormat | NULL       |
| Compressed:                 | No
         | NULL       |
| Num Buckets:                | 0
        | NULL       |
| Bucket Columns:             | []
        | NULL       |
| Sort Columns:               | []
        | NULL       |
+-----------------------------+-----------------------------------------
-----------+------------+
Returned 27 row(s) in 0.17s
```

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its LOCATION attribute.)

Cloudera Manager considerations:

A change in the behavior of metadata loading in Impala 2.9 could lead to certain long-running statements being left out of the Cloudera Manager list of Impala queries until the statements are completed.

Prior to Impala 2.9, statements such as DESCRIBE could cause the Impala web UI, and Cloudera Manager monitoring pages that rely on information from the web UI, to become unresponsive while they ran. The first time Impala references a large table, for example one with thousands of partitions, the statement might take longer than

normal while the metadata for the table is loaded. In Impala 2.9 and higher, the Impala web UI and associated Cloudera Manager monitoring pages are more responsive while a metadata loading operation is in progress.

Although the statement that loads the metadata shows up on the Impala web UI /queries page immediately, it does not show up in the Cloudera Manager list of queries until the metadata is finished loading. For example, the first DESC RIBE of a large partitioned table might take 30 minutes due to metadata loading, and the statement does not show up in Cloudera Manager during those 30 minutes.

Kudu considerations:

The information displayed for Kudu tables includes the additional attributes that are only applicable for Kudu tables:

- Whether or not the column is part of the primary key. Every Kudu table has a true value here for at least one column. There could be multiple true values, for tables with composite primary keys.
- Whether or not the column is nullable. Specified by the NULL or NOT NULL attributes on the CREATE TABLE statement. Columns that are part of the primary key are automatically non-nullable.
- The default value, if any, for the column. Specified by the DEFAULT attribute on the CREATE TABLE statement. If the default value is NULL, that is not indicated in this column. It is implied by nullable being true and no other default value specified.
- The encoding used for values in the column. Specified by the ENCODING attribute on the CREATE TABLE statement.
- The compression used for values in the column. Specified by the COMPRESSION attribute on the CREATE T ABLE statement.
- The block size (in bytes) used for the underlying Kudu storage layer for the column. Specified by the BLOCK_SI ZE attribute on the CREATE TABLE statement.

The following example shows DESCRIBE output for a simple Kudu table, with a single-column primary key and all column attributes left with their default values:

```
describe million_rows;
+------+--------+---------+-------------+----------+---------------+--------
-------+---------------------+------------+
| name | type   | comment | primary_key | nullable | default_value | encodin
g      | compression         | block_size |
+------+--------+---------+-------------+----------+---------------+--------
-------+---------------------+------------+
| id   | string |         | true        | false    |               | AUTO_EN
CODING | DEFAULT_COMPRESSION | 0          |
| s    | string |         | false       | false    |               | AUTO_EN
CODING | DEFAULT_COMPRESSION | 0          |
+------+--------+---------+-------------+----------+---------------+--------
-------+---------------------+------------+
```

The following example shows DESCRIBE output for a Kudu table with a two-column primary key, and Kudu-specific attributes applied to some columns:

```
create table kudu_describe_example
(
  c1 int, c2 int,
  c3 string, c4 string not null, c5 string default 'n/a', c6 string default
'',
  c7 bigint not null, c8 bigint null default null, c9 bigint default -1 enco
ding bit_shuffle,
  primary key(c1,c2)
)
partition by hash (c1, c2) partitions 10 stored as kudu;

describe kudu_describe_example;
```

```
+------+--------+---------+-------------+----------+---------------+-----
----------+--------------------+------------+
| name | type   | comment | primary_key | nullable | default_value | enco
ding      | compression         | block_size |
+------+--------+---------+-------------+----------+---------------+-----
----------+--------------------+------------+
| c1   | int    |         | true        | false    |               | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c2   | int    |         | true        | false    |               | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c3   | string |         | false       | true     |               | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c4   | string |         | false       | false    |               | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c5   | string |         | false       | true     | n/a           | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c6   | string |         | false       | true     |               | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c7   | bigint |         | false       | false    |               | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c8   | bigint |         | false       | true     |               | AUTO
_ENCODING | DEFAULT_COMPRESSION | 0          |
| c9   | bigint |         | false       | true     | -1            | BIT_
SHUFFLE   | DEFAULT_COMPRESSION | 0          |
+------+--------+---------+-------------+----------+---------------+-----
----------+--------------------+------------+
```

Related Information

Complex types

SHOW statement

DROP DATABASE statement

The DROP DATABASE statement removes a database from the system. The physical operations involve removing the metadata for the database from the metastore, and deleting the corresponding *.db directory from HDFS.

Syntax:

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name [RESTRICT | CASCADE];
```

Statement type: DDL

Usage notes:

By default, the database must be empty before it can be dropped, to avoid losing any data.

In Impala 2.3 and higher, you can include the CASCADE clause to make Impala drop all tables and other objects in the database before dropping the database itself. The RESTRICT clause enforces the original requirement that the database be empty before being dropped. Because the RESTRICT behavior is still the default, this clause is optional.

The automatic dropping resulting from the CASCADE clause follows the same rules as the corresponding DROP TABLE, DROP VIEW, and DROP FUNCTION statements. In particular, the HDFS directories and data files for any external tables are left behind when the tables are removed.

When you do not use the CASCADE clause, drop or move all the objects inside the database manually before dropping the database itself:

- Use the SHOW TABLES statement to locate all tables and views in the database, and issue DROP TABLE and DROP VIEW statements to remove them all.
- Use the SHOW FUNCTIONS and SHOW AGGREGATE FUNCTIONS statements to locate all user-defined functions in the database, and issue DROP FUNCTION and DROP AGGREGATE FUNCTION statements to remove them all.

- To keep tables or views contained by a database while removing the database itself, use ALTER TABLE and ALTER VIEW to move the relevant objects to a different database before dropping the original database.

You cannot drop the current database, that is, the database your session connected to either through the USE statement or the -d option of `impala-shell`. Issue a USE statement to switch to a different database first. Because the default database is always available, issuing USE default is a convenient way to leave the current database before dropping it.

Hive considerations:

When you drop a database in Impala, the database can no longer be used by Hive.

Examples:

See *CREATE DATABASE* statement for examples covering CREATE DATABASE, USE, and DROP DATABASE.

Amazon S3 considerations:

In Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DAT ABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have write permission for the directory associated with the database.

Examples:

```
create database first_db;
use first_db;
create table t1 (x int);

create database second_db;
use second_db;
-- Each database has its own namespace for tables.
-- You can reuse the same table names in each database.
create table t1 (s string);

create database temp;

-- You can either USE a database after creating it,
-- or qualify all references to the table name with the name of the datab
ase.
-- Here, tables T2 and T3 are both created in the TEMP database.

create table temp.t2 (x int, y int);
use database temp;
create table t3 (s string);

-- You cannot drop a database while it is selected by the USE statement.
drop database temp;
ERROR: AnalysisException: Cannot drop current default database: temp

-- The always-available database 'default' is a convenient one to USE
-- before dropping a database you created.
use default;
-- Before dropping a database, first drop all the tables inside it,
-- or in Impala 2.3 and higher use the CASCADE clause.
drop database temp;
ERROR: ImpalaRuntimeException: Error making 'dropDatabase' RPC to Hive Meta
store:
CAUSED BY: InvalidOperationException: Database temp is not empty
```

```
show tables in temp;
+------+
| name |
+------+
| t3   |
+------+

-- Impala 2.3 and higher:
drop database temp cascade;

-- Earlier releases:
drop table temp.t3;
drop database temp;
```

Related Information

CREATE DATABASE statement

Impala with Amazon S3

DROP FUNCTION statement

The DROP FUNCTION statement removes a user-defined function (UDF), so that it is not available for execution during Impala SELECT or INSERT operations.

Syntax:

To drop C++ UDFs and UDAs:

```
DROP [AGGREGATE] FUNCTION [IF EXISTS]
 [db_name.]function_name(type[, type...])
```

Note:

The preceding syntax, which includes the function signature, also applies to Java UDFs that were created using the corresponding CREATE FUNCTION syntax that includes the argument and return types. After upgrading to Impala 2.5 or higher, consider re-creating all Java UDFs with the CREATE FUNCTION syntax that does not include the function signature. Java UDFs created this way are now persisted in the metastore database and do not need to be re-created after an Impala restart.

To drop Java UDFs (created using the CREATE FUNCTION syntax with no function signature):

```
DROP FUNCTION [IF EXISTS] [db_name.]function_name
```

Statement type: DDL

Usage notes:

Because the same function name could be overloaded with different argument signatures, you specify the argument types to identify the exact function to drop.

Restrictions:

In Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new CREATE FUNCTION syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old CREATE FUNCTION syntax do not persist across restarts because they are held in the memory of the `catalogd` daemon. Until you re-create such Java UDFs using the new CREATE FUNCTION syntax, you must reload those Java-based UDFs by running the original CREATE FUNCTION statements again each time you restart the `catalogd` daemon. Prior to Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, does not need any particular HDFS permissions to perform this statement. All read and write operations are on the metastore database, not HDFS files and directories.

Examples:

The following example shows how to drop Java functions created with the signatureless CREATE FUNCTION syntax in Impala 2.5 and higher. Issuing DROP FUNCTION *function_name* removes all the overloaded functions under that name. See the *CREATE FUNCTION* statement for a longer example showing how to set up such functions in the first place.

```
create function my_func location '/user/impala/udfs/udf-examples-cdh570.jar'
  symbol='com.cloudera.impala.TestUdf';

show functions;
+-------------+---------------------------------------+-------------+----
-----------+
| return type | signature                             | binary type | is pe
rsistent |
+-------------+---------------------------------------+-------------+----
-----------+
| BIGINT      | my_func(BIGINT)                       | JAVA        | true
         |
| BOOLEAN     | my_func(BOOLEAN)                      | JAVA        | true
         |
| BOOLEAN     | my_func(BOOLEAN, BOOLEAN)             | JAVA        | true
         |
...
| BIGINT      | testudf(BIGINT)                       | JAVA        | true
         |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true
         |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true
         |
...

drop function my_func;
show functions;
+-------------+---------------------------------------+-------------+------
---------+
| return type | signature                             | binary type | is
persistent |
+-------------+---------------------------------------+-------------+------
---------+
| BIGINT      | testudf(BIGINT)                       | JAVA        | true
         |
| BOOLEAN     | testudf(BOOLEAN)                      | JAVA        | true
         |
| BOOLEAN     | testudf(BOOLEAN, BOOLEAN)             | JAVA        | true
         |
...
```

Related Information

CREATE FUNCTION statement

DROP STATS statement

The DROP STATS statement removes the specified statistics from a table or partition. The statistics were originally created by the COMPUTE STATS or COMPUTE INCREMENTAL STATS statement.

Syntax:

```
DROP STATS [database_name.]table_name
DROP INCREMENTAL STATS [database_name.]table_name PARTITION (partition_spec)

partition_spec ::= partition_col=constant_value
```

The PARTITION clause is only allowed in combination with the INCREMENTAL clause. It is optional for COMP UTE INCREMENTAL STATS, and required for DROP INCREMENTAL STATS. Whenever you specify partitions through the PARTITION (*partition_spec*) clause in a COMPUTE INCREMENTAL STATS or DROP INCREME NTAL STATS statement, you must include all the partitioning columns in the specification, and specify constant values for all the partition key columns.

DROP STATS removes all statistics from the table, whether created by COMPUTE STATS or COMPUTE INCR EMENTAL STATS.

DROP INCREMENTAL STATS only affects incremental statistics for a single partition, specified through the PART ITION clause. The incremental stats are marked as outdated, so that they are recomputed by the next COMPUTE INCREMENTAL STATS statement.

Usage notes:

You typically use this statement when the statistics for a table or a partition have become stale due to data files being added to or removed from the associated HDFS data directories, whether by manual HDFS operations or INSERT, INSERT OVERWRITE, or LOAD DATA statements, or adding or dropping partitions.

When a table or partition has no associated statistics, Impala treats it as essentially zero-sized when constructing the execution plan for a query. In particular, the statistics influence the order in which tables are joined in a join query. To ensure proper query planning and good query performance and scalability, make sure to run COMPUTE STATS or COMPUTE INCREMENTAL STATS on the table or partition after removing any stale statistics.

Dropping the statistics is not required for an unpartitioned table or a partitioned table covered by the original type of statistics. A subsequent COMPUTE STATS statement replaces any existing statistics with new ones, for all partitions, regardless of whether the old ones were outdated. Therefore, this statement was rarely used before the introduction of incremental statistics.

Dropping the statistics is required for a partitioned table containing incremental statistics, to make a subsequent COMPUTE INCREMENTAL STATS statement rescan an existing partition. See the *Tables and columns topic* for information about incremental statistics, a new feature available in Impala 2.1.0 and higher.

Statement type: DDL

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, does not need any particular HDFS permissions to perform this statement. All read and write operations are on the metastore database, not HDFS files and directories.

Examples:

The following example shows a partitioned table that has associated statistics produced by the COMPUTE INCR EMENTAL STATS statement, and how the situation evolves as statistics are dropped from specific partitions, then the entire table.

Initially, all table and column statistics are filled in.

```
show table stats item_partitioned;
+-------------+-------+--------+----------+--------------+---------+-----
------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Increm
ental stats
```

```
+-------------+-------+--------+----------+--------------+---------+--------
---------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | 1812  | 1      | 232.67KB | NOT CACHED   | PARQUET | true
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | 1783  | 1      | 227.97KB | NOT CACHED   | PARQUET | true
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+-----
------------
show column stats item_partitioned;
+------------------+-----------+------------------+--------+----------+----
----------
| Column           | Type      | #Distinct Values | #Nulls | Max Size | Avg
 Size
+------------------+-----------+------------------+--------+----------+-----
---------
| i_item_sk        | INT       | 19443            | -1     | 4        | 4
| i_item_id        | STRING    | 9025             | -1     | 16       | 16
| i_rec_start_date | TIMESTAMP | 4                | -1     | 16       | 16
| i_rec_end_date   | TIMESTAMP | 3                | -1     | 16       | 16
| i_item_desc      | STRING    | 13330            | -1     | 200      | 100
.302803039
| i_current_price  | FLOAT     | 2807             | -1     | 4        | 4
| i_wholesale_cost | FLOAT     | 2105             | -1     | 4        | 4
| i_brand_id       | INT       | 965              | -1     | 4        | 4
| i_brand          | STRING    | 725              | -1     | 22       | 16.
1776008605
| i_class_id       | INT       | 16               | -1     | 4        | 4
| i_class          | STRING    | 101              | -1     | 15       | 7.7
6749992370
| i_category_id    | INT       | 10               | -1     | 4        | 4
| i_manufact_id    | INT       | 1857             | -1     | 4        | 4
| i_manufact       | STRING    | 1028             | -1     | 15       | 1
1.3295001983
| i_size           | STRING    | 8                | -1     | 11       | 4.3
3459997177
| i_formulation    | STRING    | 12884            | -1     | 20       | 1
9.9799995422
| i_color          | STRING    | 92               | -1     | 10       | 5.3
8089990615
| i_units          | STRING    | 22               | -1     | 7        | 4
.18690013885
| i_container      | STRING    | 2                | -1     | 7        | 6.9
9259996414
| i_manager_id     | INT       | 105              | -1     | 4        | 4
| i_product_name   | STRING    | 19094            | -1     | 25       | 18.
0233001708
| i_category       | STRING    | 10               | 0      | -1       | -1
+------------------+-----------+------------------+--------+----------+---
-----------
```

To remove statistics for particular partitions, use the DROP INCREMENTAL STATS statement. After removing statistics for two partitions, the table-level statistics reflect that change in the #Rows and Incremental stats fields. The counts, maximums, and averages of the column-level statistics are unaffected.

Note: (It is possible that the row count might be preserved in future after a DROP INCREMENTAL STATS statement. Check the resolution of the issue IMPALA-1615.)

```
drop incremental stats item_partitioned partition (i_category='Sports');
drop incremental stats item_partitioned partition (i_category='Electronics'
);

show table stats item_partitioned
+-------------+-------+--------+----------+--------------+---------+------
------------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Increm
ental stats
+-------------+-------+--------+----------+--------------+---------+-------
---------
| Books       | 1733  | 1      | 223.74KB | NOT CACHED   | PARQUET | true
| Children    | 1786  | 1      | 230.05KB | NOT CACHED   | PARQUET | true
| Electronics | -1    | 1      | 232.67KB | NOT CACHED   | PARQUET | false
| Home        | 1807  | 1      | 232.56KB | NOT CACHED   | PARQUET | true
| Jewelry     | 1740  | 1      | 223.72KB | NOT CACHED   | PARQUET | true
| Men         | 1811  | 1      | 231.25KB | NOT CACHED   | PARQUET | true
| Music       | 1860  | 1      | 237.90KB | NOT CACHED   | PARQUET | true
| Shoes       | 1835  | 1      | 234.90KB | NOT CACHED   | PARQUET | true
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | 1790  | 1      | 226.27KB | NOT CACHED   | PARQUET | true
| Total       | 17957 | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+-------
----------
show column stats item_partitioned
+------------------+-----------+------------------+--------+----------+---
-----------
| Column           | Type      | #Distinct Values | #Nulls | Max Size | Avg
Size
+------------------+-----------+------------------+--------+----------+----
----------
| i_item_sk        | INT       | 19443            | -1     | 4        | 4
| i_item_id        | STRING    | 9025             | -1     | 16       | 16
| i_rec_start_date | TIMESTAMP | 4                | -1     | 16       | 16
| i_rec_end_date   | TIMESTAMP | 3                | -1     | 16       | 16
| i_item_desc      | STRING    | 13330            | -1     | 200      | 10
0.302803039
| i_current_price  | FLOAT     | 2807             | -1     | 4        | 4
| i_wholesale_cost | FLOAT     | 2105             | -1     | 4        | 4
| i_brand_id       | INT       | 965              | -1     | 4        | 4
| i_brand          | STRING    | 725              | -1     | 22       | 16
.1776008605
| i_class_id       | INT       | 16               | -1     | 4        | 4
| i_class          | STRING    | 101              | -1     | 15       | 7.
76749992370
| i_category_id    | INT       | 10               | -1     | 4        | 4
| i_manufact_id    | INT       | 1857             | -1     | 4        | 4
| i_manufact       | STRING    | 1028             | -1     | 15       | 11.3
295001983
| i_size           | STRING    | 8                | -1     | 11       | 4.
33459997177
| i_formulation    | STRING    | 12884            | -1     | 20       | 19.9
799995422
| i_color          | STRING    | 92               | -1     | 10       | 5.
38089990615
| i_units          | STRING    | 22               | -1     | 7        | 4.18
690013885
| i_container      | STRING    | 2                | -1     | 7        | 6.
99259996414
| i_manager_id     | INT       | 105              | -1     | 4        | 4
```

```
| i_product_name   | STRING    | 19094            | -1     | 25       | 18
.0233001708
| i_category       | STRING    | 10               | 0      | -1       | -1
+------------------+-----------+------------------+--------+----------+--
------------
```

To remove all statistics from the table, whether produced by COMPUTE STATS or COMPUTE INCREMENTAL STATS, use the DROP STATS statement without the INCREMENTAL clause). Now, both table-level and column-level statistics are reset.

```
drop stats item_partitioned;

show table stats item_partitioned
+-------------+-------+--------+----------+--------------+---------+-------
----------
| i_category  | #Rows | #Files | Size     | Bytes Cached | Format  | Incr
emental stats
+-------------+-------+--------+----------+--------------+---------+------
------------
| Books       | -1    | 1      | 223.74KB | NOT CACHED   | PARQUET | false
| Children    | -1    | 1      | 230.05KB | NOT CACHED   | PARQUET | false
| Electronics | -1    | 1      | 232.67KB | NOT CACHED   | PARQUET | false
| Home        | -1    | 1      | 232.56KB | NOT CACHED   | PARQUET | false
| Jewelry     | -1    | 1      | 223.72KB | NOT CACHED   | PARQUET | false
| Men         | -1    | 1      | 231.25KB | NOT CACHED   | PARQUET | false
| Music       | -1    | 1      | 237.90KB | NOT CACHED   | PARQUET | false
| Shoes       | -1    | 1      | 234.90KB | NOT CACHED   | PARQUET | false
| Sports      | -1    | 1      | 227.97KB | NOT CACHED   | PARQUET | false
| Women       | -1    | 1      | 226.27KB | NOT CACHED   | PARQUET | false
| Total       | -1    | 10     | 2.25MB   | 0B           |         |
+-------------+-------+--------+----------+--------------+---------+-------
----------
show column stats item_partitioned
+------------------+-----------+------------------+--------+----------+---
-------+
| Column           | Type      | #Distinct Values | #Nulls | Max Size | Avg
 Size |
+------------------+-----------+------------------+--------+----------+-----
-----+
| i_item_sk        | INT       | -1               | -1     | 4        | 4
     |
| i_item_id        | STRING    | -1               | -1     | -1       | -1
     |
| i_rec_start_date | TIMESTAMP | -1               | -1     | 16       | 16
     |
| i_rec_end_date   | TIMESTAMP | -1               | -1     | 16       | 16
     |
| i_item_desc      | STRING    | -1               | -1     | -1       | -1
     |
| i_current_price  | FLOAT     | -1               | -1     | 4        | 4
     |
| i_wholesale_cost | FLOAT     | -1               | -1     | 4        | 4
     |
| i_brand_id       | INT       | -1               | -1     | 4        | 4
     |
| i_brand          | STRING    | -1               | -1     | -1       | -1
     |
| i_class_id       | INT       | -1               | -1     | 4        | 4
     |
| i_class          | STRING    | -1               | -1     | -1       | -1
     |
| i_category_id    | INT       | -1               | -1     | 4        | 4
     |
```

```
| i_manufact_id    | INT       | -1                | -1     | 4         | 4
     |
| i_manufact       | STRING    | -1                | -1     | -1        | -1
     |
| i_size           | STRING    | -1                | -1     | -1        | -1
     |
| i_formulation    | STRING    | -1                | -1     | -1        | -1
     |
| i_color          | STRING    | -1                | -1     | -1        | -1
     |
| i_units          | STRING    | -1                | -1     | -1        | -1
     |
| i_container      | STRING    | -1                | -1     | -1        | -1
     |
| i_manager_id     | INT       | -1                | -1     | 4         | 4
     |
| i_product_name   | STRING    | -1                | -1     | -1        | -1
     |
| i_category       | STRING    | 10                | 0      | -1        | -1
     |
+------------------+-----------+-------------------+--------+-----------+----
------+
```

Related Information

Table and column statistics

DROP TABLE statement

The DROP TABLE statement removes an Impala table. Also removes the underlying HDFS data files for internal tables, although not for external tables.

Syntax:

```
DROP TABLE [IF EXISTS] [db_name.]table_name [PURGE]
```

IF EXISTS clause:

The optional IF EXISTS clause makes the statement succeed whether or not the table exists. If the table does exist, it is dropped; if it does not exist, the statement has no effect. This capability is useful in standardized setup scripts that remove existing schema objects and create new ones. By using some combination of IF EXISTS for the DROP statements and IF NOT EXISTS clauses for the CREATE statements, the script can run successfully the first time you run it (when the objects do not exist yet) and subsequent times (when some or all of the objects do already exist).

PURGE clause:

The optional PURGE keyword, available in Impala 2.3 and higher, causes Impala to remove the associated HDFS data files immediately, rather than going through the HDFS trashcan mechanism. Use this keyword when dropping a table if it is crucial to remove the data as quickly as possible to free up space, or if there is a problem with the trashcan, such as the trash cannot being configured or being in a different HDFS encryption zone than the data files.

Statement type: DDL

Usage notes:

By default, Impala removes the associated HDFS directory and data files for the table. If you issue a DROP TABLE and the data files are not deleted, it might be for the following reasons:

- If the table was created with the EXTERNAL clause, Impala leaves all files and directories untouched. Use external tables when the data is under the control of other Hadoop components, and Impala is only used to query the data files from their original locations.
- Impala might leave the data files behind unintentionally, if there is no HDFS location available to hold the HDFS trashcan for the impala user.

Make sure that you are in the correct database before dropping a table, either by issuing a USE statement first or by using a fully qualified name *db_name.table_name*.

If you intend to issue a DROP DATABASE statement, first issue DROP TABLE statements to remove all the tables in that database.

Examples:

```
create database temporary;
use temporary;
create table unimportant (x int);
create table trivial (s string);
-- Drop a table in the current database.
drop table unimportant;
-- Switch to a different database.
use default;
-- To drop a table in a different database...
drop table trivial;
ERROR: AnalysisException: Table does not exist: default.trivial
-- ...use a fully qualified name.
drop table temporary.trivial;
```

Amazon S3 considerations:

The DROP TABLE statement can remove data files from S3 if the associated S3 table is an internal table. In Impala 2.6 and higher, as part of improved support for writing to S3, Impala also removes the associated folder when dropping an internal table that resides on S3.

For compatibility with the S3 write support in Impala, follow these steps for querying table through Impala:

1. Use native Hadoop techniques, such as hadoop fs -cp or INSERT in Impala or Hive to create data files in S3.
2. Use the PURGE clause with DROP TABLE when dropping internal (managed) tables.

By default, when you do not include the PURGE clause in the statement, the data files are moved to the S3A trashcan. This operation is expensive. When you do include the PURGE clause, the data files are deleted immediately, skipping the expensive S3A trashcan operation.

In Impala 2.6 and higher, Impala DDL statements such as CREATE DATABASE, CREATE TABLE, DROP DAT ABASE CASCADE, DROP TABLE, and ALTER TABLE [ADD|DROP] PARTITION can create or remove folders as needed in the Amazon S3 system. Prior to Impala 2.6, you had to create folders yourself and point Impala database, tables, or partitions at them, and manually remove folders when no longer needed.

Cancellation: Cannot be cancelled.

HDFS permissions:

For an internal table, the user ID that the `impalad` daemon runs under, typically the impala user, must have write permission for all the files and directories that make up the table.

For an external table, dropping the table only involves changes to metadata in the metastore database. Because Impala does not remove any HDFS files or directories when external tables are dropped, no particular permissions are needed for the associated HDFS files or directories.

Kudu considerations:

Kudu tables can be managed or external, the same as with HDFS-based tables. For a managed table, the underlying Kudu table and its data are removed by DROP TABLE. For an external table, the underlying Kudu table and its data remain after a DROP TABLE.

Related Information

Managing disk space for Impala data

Impala with Amazon S3

DROP VIEW statement

The DROP VIEW statement removes the specified view, which was originally created by the CREATE VIEW statement. Because a view is purely a logical construct (an alias for a query) with no physical data behind it, DROP VIEW only involves changes to metadata in the metastore database, not any data files in HDFS.

Syntax:

```
DROP VIEW [IF EXISTS] [db_name.]view_name
```

Statement type: DDL

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Examples:

The following example creates a series of views and then drops them. These examples illustrate how views are associated with a particular database, and both the view definitions and the view names for CREATE VIEW and DROP VIEW can refer to a view in the current database or a fully qualified view name.

```
-- Create and drop a view in the current database.
CREATE VIEW few_rows_from_t1 AS SELECT * FROM t1 LIMIT 10;
DROP VIEW few_rows_from_t1;

-- Create and drop a view referencing a table in a different database.
CREATE VIEW table_from_other_db AS SELECT x FROM db1.foo WHERE x IS NOT N
ULL;
DROP VIEW table_from_other_db;
USE db1;
-- Create a view in a different database.
CREATE VIEW db2.v1 AS SELECT * FROM db2.foo;
-- Switch into the other database and drop the view.
USE db2;
DROP VIEW v1;

USE db1;
-- Create a view in a different database.
CREATE VIEW db2.v1 AS SELECT * FROM db2.foo;
-- Drop a view in the other database.
DROP VIEW db2.v1;
```

EXPLAIN statement

The EXPLAIN statement returns the execution plan for a statement, showing the low-level mechanisms that Impala will use to read the data, divide the work among nodes in the cluster, and transmit intermediate and final results across the network.

Use explain followed by a complete SELECT query. For example:

Syntax:

```
EXPLAIN { select_query | ctas_stmt | insert_stmt }
```

The *select_query* is a SELECT statement, optionally prefixed by a WITH clause.

The *insert_stmt* is an INSERT statement that inserts into or overwrites an existing table. It can use either the INSERT ... SELECT or INSERT ... VALUES syntax.

The *ctas_stmt* is a CREATE TABLE statement using the AS SELECT clause, typically abbreviated as a “CTAS” operation.

Usage notes:

You can interpret the output to judge whether the query is performing efficiently, and adjust the query and/or the schema if not. For example, you might change the tests in the WHERE clause, add hints to make join operations more efficient, introduce subqueries, change the order of tables in a join, add or change partitioning for a table, collect column statistics and/or table statistics in Hive, or any other performance tuning steps.

The EXPLAIN output reminds you if table or column statistics are missing from any table involved in the query. These statistics are important for optimizing queries involving large tables or multi-table joins.

Read the EXPLAIN plan from bottom to top:

- The last part of the plan shows the low-level details such as the expected amount of data that will be read, where you can judge the effectiveness of your partitioning strategy and estimate how long it will take to scan a table based on total data size and the size of the cluster.
- As you work your way up, next you see the operations that will be parallelized and performed on each Impala node.
- At the higher levels, you see how data flows when intermediate result sets are combined and transmitted from one node to another.
- The EXPLAIN_LEVEL query option lets you customize how much detail to show in the EXPLAIN plan depending on whether you are doing high-level or low-level tuning, dealing with logical or physical aspects of the query.

If you come from a traditional database background and are not familiar with data warehousing, keep in mind that Impala is optimized for full table scans across very large tables. The structure and distribution of this data is typically not suitable for the kind of indexing and single-row lookups that are common in OLTP environments. Seeing a query scan entirely through a large table is common, not necessarily an indication of an inefficient query. Of course, if you can reduce the volume of scanned data by orders of magnitude, for example by using a query that affects only certain partitions within a partitioned table, then you might be able to optimize a query so that it executes in seconds rather than minutes.

Extended EXPLAIN output:

For performance tuning of complex queries, and capacity planning (such as using the admission control and resource management features), you can enable more detailed and informative output for the EXPLAIN statement. In the `impala-shell` interpreter, issue the command SET EXPLAIN_LEVEL=*level*, where *level* is an integer from 0 to 3 or corresponding mnemonic values minimal, standard, extended, or verbose.

When extended EXPLAIN output is enabled, EXPLAIN statements print information about estimated memory requirements, minimum number of virtual cores, and so on.

Starting in Impala 3.2, if the EXPLAIN_LEVEL option is set to EXTENDED level or VERBOSE, the output contains the following additional information.

- The analyzed query, in the output header.

  The analyzed query may have been rewritten to include various optimizations and implicit casts. See the example below.
- The predicates in the plan output includes the same implicit casts and literals printed with a cast to show the type.

Examples:

This example shows how the standard EXPLAIN output moves from the lowest (physical) level to the higher (logical) levels. The query begins by scanning a certain amount of data; each node performs an aggregation operation (evaluating COUNT(*)) on some subset of data that is local to that node; the intermediate results are transmitted back to the coordinator node (labelled here as the EXCHANGE node); lastly, the intermediate results are summed to display the final result.

```
[impalad-host:21000] > explain select count(*) from customer_address;
+----------------------------------------------------------+
| Explain String                                           |
+----------------------------------------------------------+
| Estimated Per-Host Requirements: Memory=42.00MB VCores=1 |
```

```
|                                                        |
| 03:AGGREGATE [MERGE FINALIZE]                          |
| |  output: sum(count(*))                               |
| |                                                      |
| 02:EXCHANGE [PARTITION=UNPARTITIONED]                  |
| |                                                      |
| 01:AGGREGATE                                           |
| |  output: count(*)                                    |
| |                                                      |
| 00:SCAN HDFS [default.customer_address]                |
|    partitions=1/1 size=5.25MB                          |
+--------------------------------------------------------+
```

The following example shows an extended EXPLAIN output. Note that the analyzed query was rewritten to include:

- The 'constant folding' optimization, which simplified the expression in the original query, '1000 / 100' to '10'.
- The implicit casts in the WHERE clause.

```
EXPLAIN SELECT * FROM functional_kudu.alltypestiny WHERE bigint_col < 1000 /
 100;
+-----------------------------------------------------------+
| Explain String                                            |
+-----------------------------------------------------------+
| ...
| Analyzed query: SELECT * FROM mytable WHERE CAST(bigint_col AS DOUBLE) <
 CAST(10 AS DOUBLE)
| ...
| 00:SCAN KUDU [functional_kudu.alltypestiny]
| predicates: CAST(bigint_col AS DOUBLE) < CAST(10 AS DOUBLE)
...
```

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read and execute permissions for all applicable directories in all source tables for the query that is being explained. (A SELECT operation could read files from multiple different HDFS directories if the source table is partitioned.)

Kudu considerations:

The EXPLAIN statement displays equivalent plan information for queries against Kudu tables as for queries against HDFS-based tables.

To see which predicates Impala can “push down” to Kudu for efficient evaluation, without transmitting unnecessary rows back to Impala, look for the kudu predicates item in the scan phase of the query. The label kudu predicates indicates a condition that can be evaluated efficiently on the Kudu side. The label predicates in a SCAN KUDU node indicates a condition that is evaluated by Impala. For example, in a table with primary key column X and non-primary key column Y, you can see that some operators in the WHERE clause are evaluated immediately by Kudu and others are evaluated later by Impala:

```
EXPLAIN SELECT x,y from kudu_table WHERE
  x = 1 AND y NOT IN (2,3) AND z = 1
  AND a IS NOT NULL AND b > 0 AND length(s) > 5;
+----------------
| Explain String
+----------------
```

```
...
| 00:SCAN KUDU [kudu_table]
|    predicates: y NOT IN (2, 3), length(s) > 5
|    kudu predicates: a IS NOT NULL, b > 0, x = 1, z = 1
```

Only binary predicates, IS NULL and IS NOT NULL (in Impala 2.9 and higher), and IN predicates containing literal values that exactly match the types in the Kudu table, and do not require any casting, can be pushed to Kudu.

Related Information

Query options

Understanding Performance using EXPLAIN Plan

Table and column statistics

GRANT statement

The GRANT statement grants a privilege on a specified object to a user or to a group.

Syntax:

```
GRANT privilege ON object_type object_name
   TO USER user_name
GRANT privilege ON object_type object_name
   TO GROUP group_name

privilege ::= ALL | ALTER | CREATE | DROP | INSERT | REFRESH | SELECT |
 SELECT(column_name)
object_type ::= SERVER | URI | DATABASE | TABLE
```

Typically, the *object_name* is an identifier. For URIs, it is a string literal.

The ability to grant or revoke SELECT privilege on specific columns is available in Impala 2.3 and higher.

Usage notes:

You can only grant the ALL privilege to the URI object. Finer-grained privileges mentioned below on a URI are not supported.

The table below lists the minimum level of privileges and the scope required to execute SQL statements in CDH 6.1 / CDH 5.16 and higher. The following notations are used:

- ANY denotes the SELECT, INSERT, CREATE, or REFRESH privilege.
- ALL privilege denotes the SELECT, INSERT, CREATE, and REFRESH privileges.
- The owner of an object effectively has the ALL privilege on the object.
- The parent levels of the specified scope are implicitly supported. For example, if a privilege is listed with the TABLE scope, the same privilege granted on DATABASE and SERVER will allow the user to execute that specific SQL statement on TABLE.

SQL Statement	Privileges	Scope
SELECT	SELECT	TABLE
WITH SELECT	SELECT	TABLE
EXPLAIN SELECT	SELECT	TABLE
INSERT	INSERT	TABLE
EXPLAIN INSERT	INSERT	TABLE
TRUNCATE	INSERT	TABLE
LOAD	INSERT	TABLE
	ALL	URI
CREATE DATABASE	CREATE	SERVER

CREATE DATABASE LOCATION	CREATE	SERVER
	ALL	URI
CREATE TABLE	CREATE	DATABASE
CREATE TABLE LIKE	CREATE	DATABASE
	SELECT, INSERT, or REFRESH	TABLE
CREATE TABLE AS SELECT	CREATE	DATABASE
	INSERT	DATABASE
	SELECT	TABLE
EXPLAIN CREATE TABLE AS SELECT	CREATE	DATABASE
	INSERT	DATABASE
	SELECT	TABLE
CREATE TABLE LOCATION	CREATE	TABLE
	ALL	URI
CREATE VIEW	CREATE	DATABASE
	SELECT	TABLE
ALTER DATABASE SET OWNER	ALL WITH GRANT	DATABASE
ALTER TABLE	ALL	TABLE
ALTER TABLE SET LOCATION	ALL	TABLE
	ALL	URI
ALTER TABLE RENAME	CREATE	DATABASE
	ALL	TABLE
ALTER TABLE SET OWNER	ALL WITH GRANT	TABLE
ALTER VIEW	ALL	TABLE
	SELECT	TABLE
ALTER VIEW RENAME	CREATE	DATABASE
	ALL	TABLE
ALTER VIEW SET OWNER	ALL WITH GRANT	VIEW
DROP DATABASE	ALL	DATABASE
DROP TABLE	ALL	TABLE
DROP VIEW	ALL	TABLE
CREATE FUNCTION	CREATE	DATABASE
	ALL	URI
DROP FUNCTION	ALL	DATABASE
COMPUTE STATS	ALL	TABLE
DROP STATS	ALL	TABLE
INVALIDATE METADATA	REFRESH	SERVER
INVALIDATE METADATA <table>	REFRESH	TABLE
REFRESH <table>	REFRESH	TABLE
REFRESH AUTHORIZATION	REFRESH	SERVER
REFRESH FUNCTIONS	REFRESH	DATABASE

COMMENT ON DATABASE	ALL	DATABASE
COMMENT ON TABLE	ALL	TABLE
COMMENT ON VIEW	ALL	TABLE
COMMENT ON COLUMN	ALL	TABLE
DESCRIBE DATABASE	SELECT, INSERT, or REFRESH	DATABASE
DESCRIBE <table/view>	SELECT, INSERT, or REFRESH	TABLE
If the user has the SELECT privilege at the COLUMN level, only the columns the user has access will show.	SELECT	COLUMN
USE	ANY	TABLE
SHOW DATABASES	ANY	TABLE
SHOW TABLES	ANY	TABLE
SHOW FUNCTIONS	SELECT, INSERT, or REFRESH	DATABASE
SHOW PARTITIONS	SELECT, INSERT, or REFRESH	TABLE
SHOW TABLE STATS	SELECT, INSERT, or REFRESH	TABLE
SHOW COLUMN STATS	SELECT, INSERT, or REFRESH	TABLE
SHOW FILES	SELECT, INSERT, or REFRESH	TABLE
SHOW CREATE TABLE	SELECT, INSERT, or REFRESH	TABLE
SHOW CREATE VIEW	SELECT, INSERT, or REFRESH	TABLE
SHOW CREATE FUNCTION	SELECT, INSERT, or REFRESH	DATABASE
SHOW RANGE PARTITIONS (Kudu only)	SELECT, INSERT, or REFRESH	TABLE
UPDATE (Kudu only)	ALL	TABLE
EXPLAIN UPDATE (Kudu only)	ALL	TABLE
UPSERT (Kudu only)	ALL	TABLE
WITH UPSERT (Kudu only)	ALL	TABLE
EXPLAIN UPSERT (Kudu only)	ALL	TABLE
DELETE (Kudu only)	ALL	TABLE
EXPLAIN DELETE (Kudu only)	ALL	TABLE

Compatibility:

- The Impala GRANT and REVOKE statements are available in Impala 2.0 and later.
- In Impala 1.4 and later, Impala can make use of any privileges specified by the GRANT and REVOKE statements in Hive.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

INSERT statement

Impala supports inserting into tables and partitions that you create with the Impala CREATE TABLE statement, or pre-defined tables and partitions created through Hive.

Syntax:

```
[with_clause]
  INSERT [hint_clause] { INTO | OVERWRITE } [TABLE] table_name
  [(column_list)]
```

```
  [ PARTITION (partition_clause)]
{
    [hint_clause] select_statement
  | VALUES (value [, value ...]) [, (value [, value ...]) ...]
}

partition_clause ::= col_name [= constant] [, col_name [= constant] ...]

hint_clause ::=
  hint_with_dashes |
  hint_with_cstyle_delimiters |
  hint_with_brackets

hint_with_dashes ::= -- +SHUFFLE | -- +NOSHUFFLE -- +CLUSTERED

hint_with_cstyle_comments ::= /* +SHUFFLE */ | /* +NOSHUFFLE */ | /*
 +CLUSTERED */

hint_with_brackets ::= [SHUFFLE] | [NOSHUFFLE]
  (With this hint format, the square brackets are part of the syntax.)
```

Note: The square bracket style of hint is now deprecated and might be removed in a future release. For that reason, any newly added hints are not available with the square bracket syntax.

Appending or replacing (INTO and OVERWRITE clauses):

The INSERT INTO syntax appends data to a table. The existing data files are left as-is, and the inserted data is put into one or more new data files.

The INSERT OVERWRITE syntax replaces the data in a table. Currently, the overwritten data files are deleted immediately; they do not go through the HDFS trash mechanism.

Complex type considerations:

The INSERT statement currently does not support writing data files containing complex types (ARRAY, STRUCT, and MAP). To prepare Parquet data for such tables, you generate the data files outside Impala and then use LOAD DATA or CREATE EXTERNAL TABLE to associate those data files with the table. Currently, such tables must use the Parquet file format.

Kudu considerations:

Currently, the INSERT OVERWRITE syntax cannot be used with Kudu tables.

Kudu tables require a unique primary key for each row. If an INSERT statement attempts to insert a row with the same values for the primary key columns as an existing row, that row is discarded and the insert operation continues. When rows are discarded due to duplicate primary keys, the statement finishes with a warning, not an error. (This is a change from early releases of Kudu where the default was to return in error in such cases, and the syntax INSERT IGNORE was required to make the statement succeed. The IGNORE clause is no longer part of the INSERT syntax.)

For situations where you prefer to replace rows with duplicate primary key values, rather than discarding the new data, you can use the UPSERT statement instead of INSERT. UPSERT inserts rows that are entirely new, and for rows that match an existing primary key in the table, the non-primary-key columns are updated to reflect the values in the “upserted” data.

If you really want to store new rows, not replace existing ones, but cannot do so because of the primary key uniqueness constraint, consider recreating the table with additional columns included in the primary key.

Usage notes:

Impala currently supports:

- Copy data from another table using SELECT query. In Impala 1.2.1 and higher, you can combine CREATE TABLE and INSERT operations into a single step with the CREATE TABLE AS SELECT syntax, which bypasses the actual INSERT keyword.

- An optional WITH clause before the INSERT keyword, to define a subquery referenced in the SELECT portion.
- Create one or more new rows using constant expressions through VALUES clause. (The VALUES clause was added in Impala 1.0.1.)
- By default, the first column of each newly inserted row goes into the first column of the table, the second column into the second column, and so on.

  You can also specify the columns to be inserted, an arbitrarily ordered subset of the columns in the destination table, by specifying a column list immediately after the name of the destination table. This feature lets you adjust the inserted columns to match the layout of a SELECT statement, rather than the other way around. (This feature was added in Impala 1.1.)

  The number of columns mentioned in the column list (known as the “column permutation”) must match the number of columns in the SELECT list or the VALUES tuples. The order of columns in the column permutation can be different than in the underlying table, and the columns of each input row are reordered to match. If the number of columns in the column permutation is less than in the destination table, all unmentioned columns are set to NULL.
- An optional hint clause immediately either before the SELECT keyword or after the INSERT keyword, to fine-tune the behavior when doing an INSERT ... SELECT operation into partitioned Parquet tables. The hint clause cannot be specified in multiple places. The hint keywords are [SHUFFLE] and [NOSHUFFLE], including the square brackets. Inserting into partitioned Parquet tables can be a resource-intensive operation because it potentially involves many files being written to HDFS simultaneously, and separate large memory buffers being allocated to buffer the data for each partition.

Note:

- Insert commands that partition or add files result in changes to Hive metadata. Because Impala uses Hive metadata, such changes may necessitate a metadata refresh. For more information, see the REFRESH function.
- Currently, Impala can only insert data into tables that use the text and Parquet formats. For other file formats, insert the data using Hive and use Impala to query it.
- As an alternative to the INSERT statement, if you have existing data files elsewhere in HDFS, the LOAD DATA statement can move those files into a table. This statement works with tables of any file format.

Statement type: DML (but still affected by the SYNC_DDL query option)

Usage notes:

When you insert the results of an expression, particularly of a built-in function call, into a small numeric column such as INT, SMALLINT, TINYINT, or FLOAT, you might need to use a CAST() expression to coerce values into the appropriate type. Impala does not automatically convert from a larger type to a smaller one. For example, to insert cosine values into a FLOAT column, write CAST(COS(angle) AS FLOAT) in the INSERT statement to make the conversion explicit.

File format considerations:

Because Impala can read certain file formats that it cannot write, the INSERT statement does not work for all kinds of Impala tables. See *How Impala works with Hadoop file formats* for details about what file formats are supported by the INSERT statement.

Any INSERT statement for a Parquet table requires enough free space in the HDFS filesystem to write one block. Because Parquet data files use a block size of 1 GB by default, an INSERT might fail (even for a very small amount of data) if your HDFS is running low on space.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following example sets up new tables with the same definition as the TAB1 table using different file formats, and demonstrates inserting data into the tables created with the STORED AS TEXTFILE and STORED AS PARQUET clauses:

```
CREATE DATABASE IF NOT EXISTS file_formats;
USE file_formats;

DROP TABLE IF EXISTS text_table;
CREATE TABLE text_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS TEXTFILE;

DROP TABLE IF EXISTS parquet_table;
CREATE TABLE parquet_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS PARQUET;
```

With the INSERT INTO TABLE syntax, each new set of inserted rows is appended to any existing data in the table. This is how you would record small amounts of data that arrive continuously, or ingest new batches of data alongside the existing data. For example, after running 2 INSERT INTO TABLE statements with 5 rows each, the table contains 10 rows total:

```
[localhost:21000] > insert into table text_table select * from default.tab1;
Inserted 5 rows in 0.41s

[localhost:21000] > insert into table text_table select * from default.tab1;
Inserted 5 rows in 0.46s

[localhost:21000] > select count(*) from text_table;
+----------+
| count(*) |
+----------+
| 10       |
+----------+
Returned 1 row(s) in 0.26s
```

With the INSERT OVERWRITE TABLE syntax, each new set of inserted rows replaces any existing data in the table. This is how you load data to query in a data warehousing scenario where you analyze just the data for a particular day, quarter, and so on, discarding the previous data each time. You might keep the entire set of data in one raw table, and transfer and transform certain rows into a more compact and efficient form to perform intensive analysis on that subset.

For example, here we insert 5 rows into a table using the INSERT INTO clause, then replace the data by inserting 3 rows with the INSERT OVERWRITE clause. Afterward, the table only contains the 3 rows from the final INSERT statement.

```
[localhost:21000] > insert into table parquet_table select * from default.ta
b1;
Inserted 5 rows in 0.35s

[localhost:21000] > insert overwrite table parquet_table select * from def
ault.tab1 limit 3;
Inserted 3 rows in 0.43s
```

```
[localhost:21000] > select count(*) from parquet_table;
+----------+
| count(*) |
+----------+
| 3        |
+----------+
Returned 1 row(s) in 0.43s
```

The VALUES clause lets you insert one or more rows by specifying constant values for all the columns. The number, types, and order of the expressions must match the table definition.

Note: The INSERT ... VALUES technique is not suitable for loading large quantities of data into HDFS-based tables, because the insert operations cannot be parallelized, and each one produces a separate data file. Use it for setting up small dimension tables or tiny amounts of data for experimenting with SQL syntax, or with HBase tables. Do not use it for large ETL jobs or benchmark tests for load operations. Do not run scripts with thousands of INSERT ... VALUES statements that insert a single row each time. If you do run INSERT ... VALUES operations to load data into a staging table as one stage in an ETL pipeline, include multiple row values if possible within each VALUES clause, and use a separate database to make cleanup easier if the operation does produce many tiny files.

The following example shows how to insert one row or multiple rows, with expressions of different types, using literal values, expressions, and function return values:

```
create table val_test_1 (c1 int, c2 float, c3 string, c4 boolean, c5 timesta
mp);
insert into val_test_1 values (100, 99.9/10, 'abc', true, now());
create table val_test_2 (id int, token string);
insert overwrite val_test_2 values (1, 'a'), (2, 'b'), (-1,'xyzzy');
```

These examples show the type of “not implemented” error that you see when attempting to insert data into a table with a file format that Impala currently does not write to:

```
DROP TABLE IF EXISTS sequence_table;
CREATE TABLE sequence_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS SEQUENCEFILE;

DROP TABLE IF EXISTS rc_table;
CREATE TABLE rc_table
( id INT, col_1 BOOLEAN, col_2 DOUBLE, col_3 TIMESTAMP )
STORED AS RCFILE;

[localhost:21000] > insert into table rc_table select * from default.tab1;
Remote error
Backend 0:RC_FILE not implemented.

[localhost:21000] > insert into table sequence_table select * from default.t
ab1;
Remote error
Backend 0:SEQUENCE_FILE not implemented.
```

The following examples show how you can copy the data in all the columns from one table to another, copy the data from only some columns, or specify the columns in the select list in a different order than they actually appear in the table:

```
-- Start with 2 identical tables.
create table t1 (c1 int, c2 int);
create table t2 like t1;

-- If there is no () part after the destination table name,
-- all columns must be specified, either as * or by name.
```

```
insert into t2 select * from t1;
insert into t2 select c1, c2 from t1;

-- With the () notation following the destination table name,
-- you can omit columns (all values for that column are NULL
-- in the destination table), and/or reorder the values
-- selected from the source table. This is the "column permutation" feature.
insert into t2 (c1) select c1 from t1;
insert into t2 (c2, c1) select c1, c2 from t1;

-- The column names can be entirely different in the source and destination
 tables.
-- You can copy any columns, not just the corresponding ones, from the so
urce table.
-- But the number and type of selected columns must match the columns mentio
ned in the () part.
alter table t2 replace columns (x int, y int);
insert into t2 (y) select c1 from t1;
```

Sorting considerations: Although you can specify an ORDER BY clause in an INSERT ... SELECT statement, any ORDER BY clause is ignored and the results are not necessarily sorted. An INSERT ... SELECT operation potentially creates many different data files, prepared by different executor Impala daemons, and therefore the notion of the data being stored in sorted order is impractical.

Concurrency considerations: Each INSERT operation creates new data files with unique names, so you can run multiple INSERT INTO statements simultaneously without filename conflicts. While data is being inserted into an Impala table, the data is staged temporarily in a subdirectory inside the data directory; during this period, you cannot issue queries against that table in Hive. If an INSERT operation fails, the temporary data file and the subdirectory could be left behind in the data directory. If so, remove the relevant subdirectory and any data files it contains manually, by issuing an hdfs dfs -rm -r command, specifying the full path of the work subdirectory, whose name ends in _dir.

VALUES clause

The VALUES clause is a general-purpose way to specify the columns of one or more rows, typically within an INSERT statement.

Note: The INSERT ... VALUES technique is not suitable for loading large quantities of data into HDFS-based tables, because the insert operations cannot be parallelized, and each one produces a separate data file. Use it for setting up small dimension tables or tiny amounts of data for experimenting with SQL syntax, or with HBase tables. Do not use it for large ETL jobs or benchmark tests for load operations. Do not run scripts with thousands of INSERT ... VALUES statements that insert a single row each time. If you do run INSERT ... VALUES operations to load data into a staging table as one stage in an ETL pipeline, include multiple row values if possible within each VALUES clause, and use a separate database to make cleanup easier if the operation does produce many tiny files.

The following examples illustrate:

- How to insert a single row using a VALUES clause.
- How to insert multiple rows using a VALUES clause.
- How the row or rows from a VALUES clause can be appended to a table through INSERT INTO, or replace the contents of the table through INSERT OVERWRITE.
- How the entries in a VALUES clause can be literals, function results, or any other kind of expression. See Impala SQL literals on page 82 for the notation to use for literal values, especially for quoting and escaping conventions for strings. See Impala SQL operators on page 85 and Impala built-in functions on page 275 for other things you can include in expressions with the VALUES clause.

```
[localhost:21000] > describe val_example;
Query: describe val_example
```

```
Query finished, fetching results ...
+-------+---------+---------+
| name  | type    | comment |
+-------+---------+---------+
| id    | int     |         |
| col_1 | boolean |         |
| col_2 | double  |         |
+-------+---------+---------+

[localhost:21000] > insert into val_example values (1,true,100.0);
Inserted 1 rows in 0.30s
[localhost:21000] > select * from val_example;
+----+-------+-------+
| id | col_1 | col_2 |
+----+-------+-------+
| 1  | true  | 100   |
+----+-------+-------+
[localhost:21000] > insert overwrite val_example values (10,false,pow(2,5)),
 (50,true,10/3);
Inserted 2 rows in 0.16s
[localhost:21000] > select * from val_example;
+----+-------+-------------------+
| id | col_1 | col_2             |
+----+-------+-------------------+
| 10 | false | 32                |
| 50 | true  | 3.333333333333333 |
+----+-------+-------------------+
```

When used in an INSERT statement, the Impala VALUES clause can specify some or all of the columns in the destination table, and the columns can be specified in a different order than they actually appear in the table. To specify a different set or order of columns than in the table, use the syntax:

```
INSERT INTO destination
  (col_x, col_y, col_z)
  VALUES
  (val_x, val_y, val_z);
```

Any columns in the table that are not listed in the INSERT statement are set to NULL.

HDFS considerations:

Impala physically writes all inserted files under the ownership of its default user, typically impala. Therefore, this user must have HDFS write permission in the corresponding table directory.

The permission requirement is independent of the authorization performed by the Ranger framework. (If the connected user is not authorized to insert into a table, Ranger blocks that operation immediately, regardless of the privileges available to the impala user.) Files created by Impala are not owned by and do not inherit permissions from the connected user.

The number of data files produced by an INSERT statement depends on the size of the cluster, the number of data blocks that are processed, the partition key columns in a partitioned table, and the mechanism Impala uses for dividing the work in parallel. Do not assume that an INSERT statement will produce some particular number of output files. In case of performance issues with data written by Impala, check that the output files do not suffer from issues such as many tiny files or many tiny partitions. (In the Hadoop context, even files or partitions of a few tens of megabytes are considered “tiny”.)

The INSERT statement has always left behind a hidden work directory inside the data directory of the table. Formerly, this hidden work directory was named .impala_insert_staging . In Impala 2.0.1 and later, this directory name is changed to _impala_insert_staging . (While HDFS tools are expected to treat names beginning either with underscore and dot as hidden, in practice names beginning with an underscore are more widely supported.) If you have any scripts, cleanup jobs, and so on that rely on the name of this work directory, adjust them to use the new name.

HBase considerations:

You can use the INSERT statement with HBase tables as follows:

- You can insert a single row or a small set of rows into an HBase table with the INSERT ... VALUES syntax. This is a good use case for HBase tables with Impala, because HBase tables are not subject to the same kind of fragmentation from many small insert operations as HDFS tables are.
- You can insert any number of rows at once into an HBase table using the INSERT ... SELECT syntax.
- If more than one inserted row has the same value for the HBase key column, only the last inserted row with that value is visible to Impala queries. You can take advantage of this fact with INSERT ... VALUES statements to effectively update rows one at a time, by inserting new rows with the same key values as existing rows. Be aware that after an INSERT ... SELECT operation copying from an HDFS table, the HBase table might contain fewer rows than were inserted, if the key column in the source table contained duplicate values.
- You cannot INSERT OVERWRITE into an HBase table. New rows are always appended.
- When you create an Impala or Hive table that maps to an HBase table, the column order you specify with the INSERT statement might be different than the order you declare with the CREATE TABLE statement. Behind the scenes, HBase arranges the columns based on how they are divided into column families. This might cause a mismatch during insert operations, especially if you use the syntax INSERT INTO *hbase_table* SELECT * FROM *hdfs_table*. Before inserting data, verify the column order by issuing a DESCRIBE statement for the table, and adjust the order of the select list in the INSERT statement.

Amazon S3 considerations:

In Impala 2.6 and higher, the Impala DML statements (INSERT, LOAD DATA, and CREATE TABLE AS SELECT) can write data into a table or partition that resides in the Amazon Simple Storage Service (S3). The syntax of the DML statements is the same as for any other tables, because the S3 location for tables and partitions is specified by an s3a:// prefix in the LOCATION attribute of CREATE TABLE or ALTER TABLE statements. If you bring data into S3 using the normal S3 transfer mechanisms instead of Impala DML statements, issue a REFRESH statement for the table before using Impala to query the S3 data.

Because of differences between S3 and traditional filesystems, DML operations for S3 tables can take longer than for tables on HDFS. For example, both the LOAD DATA statement and the final stage of the INSERT and CREATE TABLE AS SELECT statements involve moving files from one directory to another. (In the case of INSERT and CREATE TABLE AS SELECT, the files are moved from a temporary staging directory to the final destination directory.) Because S3 does not support a “rename” operation for existing objects, in these cases Impala actually copies the data files from one location to another and then removes the original files. In Impala 2.6, the S3_SKIP_INSERT_STAGING query option provides a way to speed up INSERT statements for S3 tables and partitions, with the tradeoff that a problem during statement execution could leave data in an inconsistent state. It does not apply to INSERT OVERWRITE or LOAD DATA statements.

ADLS considerations:

In Impala 2.9 and higher, the Impala DML statements (INSERT, LOAD DATA, and CREATE TABLE AS SELECT) can write data into a table or partition that resides in the Azure Data Lake Store (ADLS). ADLS Gen2 is supported in Impala 3.1 and higher.

In theCREATE TABLE or ALTER TABLE statements, specify the ADLS location for tables and partitions with the adl:// prefix for ADLS Gen1 and abfs:// or abfss:// for ADLS Gen2 in the LOCATION attribute.

If you bring data into ADLS using the normal ADLS transfer mechanisms instead of Impala DML statements, issue a REFRESH statement for the table before using Impala to query the ADLS data.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts.

Cancellation: Can be cancelled. To cancel this statement, use Ctrl-C from the `impala-shell` interpreter, the Cancel button from the Watch page in Hue, Actions > Cancel from the Queries list in Cloudera Manager, or Cancel from the list of in-flight queries (for a particular node) on the Queries tab in the Impala web UI (port 25000).

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read permission for the files in the source directory of an INSERT ... SELECT operation, and write permission for all affected directories in the destination table. (An INSERT operation could write files to multiple different HDFS directories if the destination table is partitioned.) This user must also have write permission to create a temporary work directory in the top-level HDFS directory of the destination table. An INSERT OVERWRITE operation does not require write permission on the original data files in the table, only on the table directories themselves.

Restrictions:

For INSERT operations into CHAR or VARCHAR columns, you must cast all STRING literals or expressions returning STRING to to a CHAR or VARCHAR type with the appropriate length.

Related startup options:

By default, if an INSERT statement creates any new subdirectories underneath a partitioned table, those subdirectories are assigned default HDFS permissions for the impala user. To make each subdirectory have the same permissions as its parent directory in HDFS, specify the ##insert_inherit_permissions startup option for the `impalad` daemon.

Inserting into partitioned tables with PARTITION clause

For a partitioned table, the optional PARTITION clause identifies which partition or partitions the values are inserted into.

All examples in this section will use the table declared as below:

```
CREATE TABLE t1 (w INT) PARTITIONED BY (x INT, y STRING);
```

Static partition inserts

In a static partition insert where a partition key column is given a constant value, such as PART ITION (year=2012, month=2), the rows are inserted with the same values specified for those partition key columns.

The number of columns in the SELECT list must equal the number of columns in the column permutation.

The PARTITION clause must be used for static partitioning inserts.

Example:

The following statement will insert the some_other_table.c1 values for the w column, and all the rows inserted will have the same x value of 10, and the same y value of ‘a’.

```
INSERT INTO t1 PARTITION (x=10, y='a')
            SELECT c1 FROM some_other_table;
```

Dynamic partition inserts

In a dynamic partition insert where a partition key column is in the INSERT statement but not assigned a value, such as in PARTITION (year, region)(both columns unassigned) or PARTITIO N(year, region='CA') (year column unassigned), the unassigned columns are filled in with the final columns of the SELECT or VALUES clause. In this case, the number of columns in the SELECT list must equal the number of columns in the column permutation plus the number of partition key columns not assigned a constant value.

The following rules apply to dynamic partition inserts.

- The columns are bound in the order they appear in the INSERT statement.

  The table below shows the values inserted with the INSERT statements of different column orders.

	Column w Value	Column x Value	Column y Value
INSERT INTO t1 (w, x, y) VALUES (1, 2, 'c');	1	2	‘c’
INSERT INTO t1 (x,w) PARTITION (y) VALUES (1, 2, 'c');	2	1	‘c’

- When a partition clause is specified but the non-partition columns are not specified in the INSERT statement, as in the first example below, the non-partition columns are treated as though they had been specified before the PARTITION clause in the SQL.

  Example: These three statements are equivalent, inserting 1 to w, 2 to x, and ‘c’ to y columns.

  ```
  INSERT INTO t1 PARTITION (x,y) VALUES (1, 2, ‘c’);
  INSERT INTO t1 (w) PARTITION (x, y) VALUES (1, 2, ‘c’);
  INSERT INTO t1 PARTITION (x, y='c') VALUES (1, 2);
  ```

- The PARTITION clause is not required for dynamic partition, but all the partition columns must be explicitly present in the INSERT statement in the column list or in the PARTITION clause. The partition columns cannot be defaulted to NULL.

  Example:

  The following statements are valid because the partition columns, x and y, are present in the INSERT statements, either in the PARTITION clause or in the column list.

  ```
  INSERT INTO t1 PARTITION (x,y) VALUES (1, 2, ‘c’);
  INSERT INTO t1 (w, x) PARTITION (y) VALUES (1, 2, ‘c’);
  ```

  The following statement is not valid for the partitioned table as defined above because the partition columns, x and y, are not present in the INSERT statement.

  ```
  INSERT INTO t1 VALUES (1, 2, 'c');
  ```

- If partition columns do not exist in the source table, you can specify a specific value for that column in the PARTITION clause.

  Example: The source table only contains the column w and y. The value, 20, specified in the PARTITION clause, is inserted into the x column.

  ```
  INSERT INTO t1 PARTITION (x=20, y) SELECT * FROM source;
  ```

Related Information

Reading and writing ADLS data with Impala

Examples and performance characteristics of static and dynamic partitioned inserts

INVALIDATE METADATA statement

The INVALIDATE METADATA statement marks the metadata for one or all tables as stale. The next time the Impala service performs a query against a table whose metadata is invalidated, Impala reloads the associated metadata before the query proceeds. As this is a very expensive operation compared to the incremental metadata update done by the REFRESH statement, when possible, prefer REFRESH rather than INVALIDATE METADATA.

INVALIDATE METADATA is required when the following changes are made outside of Impala, in Hive and other Hive client, such as SparkSQL:

- Metadata of existing tables changes.
- New tables are added, and Impala will use the tables.
- The SERVER or DATABASE level privileges are changed from outside of Impala.
- Block metadata changes, but the files remain the same (HDFS rebalance).

- UDF jars change.
- Some tables are no longer queried, and you want to remove their metadata from the catalog and coordinator caches to reduce memory requirements.

No INVALIDATE METADATA is needed when the changes are made by impalad.

Once issued, the INVALIDATE METADATA statement cannot be cancelled.

Syntax:

```
INVALIDATE METADATA [[db_name.]table_name]
```

If there is no table specified, the cached metadata for all tables is flushed and synced with Hive Metastore (HMS). If tables were dropped from the HMS, they will be removed from the catalog, and if new tables were added, they will show up in the catalog.

If you specify a table name, only the metadata for that one table is flushed and synced with the HMS.

Usage notes:

To return accurate query results, Impala need to keep the metadata current for the databases and tables queried. Therefore, if some other entity modifies information used by Impala in the metastore, the information cached by Impala must be updated via INVALIDATE METADATA or REFRESH.

INVALIDATE METADATA and REFRESH are counterparts:

- INVALIDATE METADATA is an asynchronous operations that simply discards the loaded metadata from the catalog and coordinator caches. After that operation, the catalog and all the Impala coordinators only know about the existence of databases and tables and nothing more. Metadata loading for tables is triggered by any subsequent queries.
- REFRESH reloads the metadata synchronously. REFRESH is more lightweight than doing a full metadata load after a table has been invalidated. REFRESH cannot detect changes in block locations triggered by operations like HDFS balancer, hence causing remote reads during query execution with negative performance implications.

Use REFRESH after invalidating a specific table to separate the metadata load from the first query that's run against that table.

Examples:

This example illustrates creating a new database and new table in Hive, then doing an INVALIDATE METADATA statement in Impala using the fully qualified table name, after which both the new table and the new database are visible to Impala.

Before the INVALIDATE METADATA statement was issued, Impala would give a “not found” error if you tried to refer to those database or table names.

```
$ hive
hive> CREATE DATABASE new_db_from_hive;
hive> CREATE TABLE new_db_from_hive.new_table_from_hive (x INT);
hive> quit;
$ impala-shell
> REFRESH new_db_from_hive.new_table_from_hive;
ERROR: AnalysisException: Database does not exist: new_db_from_hive
> INVALIDATE METADATA new_db_from_hive.new_table_from_hive;

> SHOW DATABASES LIKE 'new*';
+--------------------+
| new_db_from_hive   |
+--------------------+

> SHOW TABLES IN new_db_from_hive;
+---------------------+
| new_table_from_hive |
+---------------------+
```

Use the REFRESH statement for incremental metadata update.

```
> REFRESH new_table_from_hive;
```

HDFS considerations:

By default, the INVALIDATE METADATA command checks HDFS permissions of the underlying data files and directories, caching this information so that a statement can be cancelled immediately if for example the impala user does not have permission to write to the data directory for the table. (This checking does not apply when the `catalogd` configuration option --load_catalog_in_background is set to false, which it is by default.) Impala reports any lack of write permissions as an INFO message in the log file.

If you change HDFS permissions to make data readable or writeable by the Impala user, issue another INVALIDATE METADATA to make Impala aware of the change.

Kudu considerations:

By default, much of the metadata for Kudu tables is handled by the underlying storage layer. Kudu tables have less reliance on the Metastore database, and require less metadata caching on the Impala side. For example, information about partitions in Kudu tables is managed by Kudu, and Impala does not cache any block locality metadata for Kudu tables. If the Kudu service is not integrated with the Hive Metastore, Impala will manage Kudu table metadata in the Hive Metastore.

The REFRESH and INVALIDATE METADATA statements are needed less frequently for Kudu tables than for HDFS-backed tables. Neither statement is needed when data is added to, removed, or updated in a Kudu table, even if the changes are made directly to Kudu through a client program using the Kudu API. Run REFRESH *table_name* or INVALIDATE METADATA *table_name* for a Kudu table only after making a change to the Kudu table schema, such as adding or dropping a column.

Related Information

On-demand Metadata

LOAD DATA statement

The LOAD DATA statement streamlines the ETL process for an internal Impala table by moving a data file or all the data files in a directory from an HDFS location into the Impala data directory for that table.

Syntax:

```
LOAD DATA INPATH 'hdfs_file_or_directory_path' [OVERWRITE] INTO T
ABLE tablename
  [PARTITION (partcol1=val1, partcol2=val2 ...)]
```

When the LOAD DATA statement operates on a partitioned table, it always operates on one partition at a time. Specify the PARTITION clauses and list all the partition key columns, with a constant value specified for each.

Statement type: DML (but still affected by the SYNC_DDL query option)

Usage notes:

- The loaded data files are moved, not copied, into the Impala data directory.
- You can specify the HDFS path of a single file to be moved, or the HDFS path of a directory to move all the files inside that directory. You cannot specify any sort of wildcard to take only some of the files from a directory. When loading a directory full of data files, keep all the data files at the top level, with no nested directories underneath.
- Currently, the Impala LOAD DATA statement only imports files from HDFS, not from the local filesystem. It does not support the LOCAL keyword of the Hive LOAD DATA statement. You must specify a path, not an hdfs :// URI.
- In the interest of speed, only limited error checking is done. If the loaded files have the wrong file format, different columns than the destination table, or other kind of mismatch, Impala does not raise any error for the LOAD DATA statement. Querying the table afterward could produce a runtime error or unexpected results.

Currently, the only checking the LOAD DATA statement does is to avoid mixing together uncompressed and LZO-compressed text files in the same table.

- When you specify an HDFS directory name as the LOAD DATA argument, any hidden files in that directory (files whose names start with a .) are not moved to the Impala data directory.
- The operation fails if the source directory contains any non-hidden directories. Prior to Impala 2.5 if the source directory contained any subdirectory, even a hidden one such as _impala_insert_staging, the LOAD DATA statement would fail. In Impala 2.5 and higher, LOAD DATA ignores hidden subdirectories in the source directory, and only fails if any of the subdirectories are non-hidden.
- The loaded data files retain their original names in the new location, unless a name conflicts with an existing data file, in which case the name of the new file is modified slightly to be unique. (The name-mangling is a slight difference from the Hive LOAD DATA statement, which replaces identically named files.)
- By providing an easy way to transport files from known locations in HDFS into the Impala data directory structure, the LOAD DATA statement lets you avoid memorizing the locations and layout of HDFS directory tree containing the Impala databases and tables. (For a quick way to check the location of the data files for an Impala table, issue the statement DESCRIBE FORMATTED *table_name*.)
- The PARTITION clause is especially convenient for ingesting new data for a partitioned table. As you receive new data for a time period, geographic region, or other division that corresponds to one or more partitioning columns, you can load that data straight into the appropriate Impala data directory, which might be nested several levels down if the table is partitioned by multiple columns. When the table is partitioned, you must specify constant values for all the partitioning columns.

Complex type considerations:

Because Impala currently cannot create Parquet data files containing complex types (ARRAY, STRUCT, and MAP), the LOAD DATA statement is especially important when working with tables containing complex type columns. You create the Parquet data files outside Impala, then use either LOAD DATA, an external table, or HDFS-level file operations followed by REFRESH to associate the data files with the corresponding table.

If you connect to different Impala nodes within an `impala-shell` session for load-balancing purposes, you can enable the SYNC_DDL query option to make each DDL statement wait before returning, until the new or changed metadata has been received by all the Impala nodes.

Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

First, we use a trivial Python script to write different numbers of strings (one per line) into files stored in the doc_demo HDFS user account. (Substitute the path for your own HDFS user account when doing `hdfs dfs` operations like these.)

```
random_strings.py 1000 | hdfs dfs -put - /user/doc_demo/thousand_strings.txt
random_strings.py 100 | hdfs dfs -put - /user/doc_demo/hundred_strings.txt
random_strings.py 10 | hdfs dfs -put - /user/doc_demo/ten_strings.txt
```

Next, we create a table and load an initial set of data into it. Remember, unless you specify a STORED AS clause, Impala tables default to TEXTFILE format with Ctrl-A (hex 01) as the field delimiter. This example uses a single-column table, so the delimiter is not significant. For large-scale ETL jobs, you would typically use binary format data files such as Parquet or Avro, and load them into Impala tables that use the corresponding file format.

```
[localhost:21000] > create table t1 (s string);
[localhost:21000] > load data inpath '/user/doc_demo/thousand_strings.txt' i
nto table t1;
Query finished, fetching results ...
+-----------------------------------------------------------+
| summary                                                   |
+-----------------------------------------------------------+
```

```
| Loaded 1 file(s). Total files in destination location: 1 |
+----------------------------------------------------------+
Returned 1 row(s) in 0.61s
[kilo2-202-961.cs1cloud.internal:21000] > select count(*) from t1;
Query finished, fetching results ...
+------+
| _c0  |
+------+
| 1000 |
+------+
Returned 1 row(s) in 0.67s
[localhost:21000] > load data inpath '/user/doc_demo/thousand_strings.txt'
 into table t1;
ERROR: AnalysisException: INPATH location '/user/doc_demo/thousand_strings.
txt' does not exist.
```

As indicated by the message at the end of the previous example, the data file was moved from its original location. The following example illustrates how the data file was moved into the Impala data directory for the destination table, keeping its original filename:

```
$ hdfs dfs -ls /user/hive/warehouse/load_data_testing.db/t1
Found 1 items
-rw-r--r--   1 doc_demo doc_demo      13926 2013-06-26 15:40 /user/hive/wa
rehouse/load_data_testing.db/t1/thousand_strings.txt
```

The following example demonstrates the difference between the INTO TABLE and OVERWRITE TABLE clauses. The table already contains 1000 rows. After issuing the LOAD DATA statement with the INTO TABLE clause, the table contains 100 more rows, for a total of 1100. After issuing the LOAD DATA statement with the OVERWRITE INTO TABLE clause, the former contents are gone, and now the table only contains the 10 rows from the just-loaded data file.

```
[localhost:21000] > load data inpath '/user/doc_demo/hundred_strings.txt' in
to table t1;
Query finished, fetching results ...
+----------------------------------------------------------+
| summary                                                  |
+----------------------------------------------------------+
| Loaded 1 file(s). Total files in destination location: 2 |
+----------------------------------------------------------+
Returned 1 row(s) in 0.24s
[localhost:21000] > select count(*) from t1;
Query finished, fetching results ...
+------+
| _c0  |
+------+
| 1100 |
+------+
Returned 1 row(s) in 0.55s
[localhost:21000] > load data inpath '/user/doc_demo/ten_strings.txt' overwr
ite into table t1;
Query finished, fetching results ...
+----------------------------------------------------------+
| summary                                                  |
+----------------------------------------------------------+
| Loaded 1 file(s). Total files in destination location: 1 |
+----------------------------------------------------------+
Returned 1 row(s) in 0.26s
[localhost:21000] > select count(*) from t1;
Query finished, fetching results ...
+-----+
| _c0 |
+-----+
```

```
| 10  |
+-----+
Returned 1 row(s) in 0.62s
```

Amazon S3 considerations:

In Impala 2.6 and higher, the Impala DML statements (INSERT, LOAD DATA, and CREATE TABLE AS SELECT) can write data into a table or partition that resides in the Amazon Simple Storage Service (S3). The syntax of the DML statements is the same as for any other tables, because the S3 location for tables and partitions is specified by an s3a:// prefix in the LOCATION attribute of CREATE TABLE or ALTER TABLE statements. If you bring data into S3 using the normal S3 transfer mechanisms instead of Impala DML statements, issue a REFRESH statement for the table before using Impala to query the S3 data.

Because of differences between S3 and traditional filesystems, DML operations for S3 tables can take longer than for tables on HDFS. For example, both the LOAD DATA statement and the final stage of the INSERT and CREATE TABLE AS SELECT statements involve moving files from one directory to another. (In the case of INSERT and CREATE TABLE AS SELECT, the files are moved from a temporary staging directory to the final destination directory.) Because S3 does not support a “rename” operation for existing objects, in these cases Impala actually copies the data files from one location to another and then removes the original files. In Impala 2.6, the S3_SKIP_INSERT_STAGING query option provides a way to speed up INSERT statements for S3 tables and partitions, with the tradeoff that a problem during statement execution could leave data in an inconsistent state. It does not apply to INSERT OVERWRITE or LOAD DATA statements.

ADLS considerations:

In Impala 2.9 and higher, the Impala DML statements (INSERT, LOAD DATA, and CREATE TABLE AS SELECT) can write data into a table or partition that resides in the Azure Data Lake Store (ADLS). ADLS Gen2 is supported in Impala 3.1 and higher.

In theCREATE TABLE or ALTER TABLE statements, specify the ADLS location for tables and partitions with the adl:// prefix for ADLS Gen1 and abfs:// or abfss:// for ADLS Gen2 in the LOCATION attribute.

If you bring data into ADLS using the normal ADLS transfer mechanisms instead of Impala DML statements, issue a REFRESH statement for the table before using Impala to query the ADLS data.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read and write permissions for the files in the source directory, and write permission for the destination directory.

Kudu considerations:

The LOAD DATA statement cannot be used with Kudu tables.

HBase considerations:

The LOAD DATA statement cannot be used with HBase tables.

Related information:

The LOAD DATA statement is an alternative to the INSERT statement. Use LOAD DATA when you have the data files in HDFS but outside of any Impala table.

The LOAD DATA statement is also an alternative to the CREATE EXTERNAL TABLE statement. Use LOAD DATA when it is appropriate to move the data files under Impala control rather than querying them from their original location.

Related Information

Reading and writing ADLS data with Impala

REFRESH statement

The REFRESH statement reloads the metadata for the table from the metastore database and does an incremental reload of the file and block metadata from the HDFS NameNode. REFRESH is used to avoid inconsistencies between Impala and external metadata sources, namely Hive Metastore (HMS) and NameNodes.

The REFRESH statement is only required if you load data from outside of Impala. Updated metadata, as a result of running REFRESH, is broadcast to all Impala coordinators.

Syntax:

```
REFRESH [db_name.]table_name [PARTITION (key_col1=val1 [, key_col2=val2...]
)]
```

Usage notes:

The table name is a required parameter, and the table must already exist and be known to Impala.

Only the metadata for the specified table is reloaded.

Use the REFRESH statement to load the latest metastore metadata for a particular table after one of the following scenarios happens outside of Impala:

- Deleting, adding, or modifying files.

  For example, after loading new data files into the HDFS data directory for the table, appending to an existing HDFS file, inserting data from Hive via INSERT or LOAD DATA.
- Deleting, adding, or modifying partitions.

  For example, after issuing ALTER TABLE or other table-modifying SQL statement in Hive

Note:

In Impala 2.3 and higher, the ALTER TABLE *table_name* RECOVER PARTITIONS statement is a faster alternative to REFRESH when you are only adding new partition directories through Hive or manual HDFS operations.

INVALIDATE METADATA and REFRESH are counterparts:

- INVALIDATE METADATA is an asynchronous operations that simply discards the loaded metadata from the catalog and coordinator caches. After that operation, the catalog and all the Impala coordinators only know about the existence of databases and tables and nothing more. Metadata loading for tables is triggered by any subsequent queries.
- REFRESH reloads the metadata synchronously. REFRESH is more lightweight than doing a full metadata load after a table has been invalidated. REFRESH cannot detect changes in block locations triggered by operations like HDFS balancer, hence causing remote reads during query execution with negative performance implications.

Refreshing a single partition:

In Impala 2.7 and higher, the REFRESH statement can apply to a single partition at a time, rather than the whole table. Include the optional PARTITION (*partition_spec*) clause and specify values for each of the partition key columns.

The following rules apply:

- The PARTITION clause of the REFRESH statement must include all the partition key columns.
- The order of the partition key columns does not have to match the column order in the table.
- Specifying a nonexistent partition does not cause an error.
- The partition can be one that Impala created and is already aware of, or a new partition created through Hive.

The following examples demonstrates the above rules.

```

-- Partition doesn't exist.
REFRESH p2 PARTITION (y=0, z=3);
```

```
REFRESH p2 PARTITION (y=0, z=-1)

-- Key columns specified in a different order than the table definition.
REFRESH p2 PARTITION (z=1, y=0)

-- Incomplete partition spec causes an error.
REFRESH p2 PARTITION (y=0)
ERROR: AnalysisException: Items in partition spec must exactly match the
partition columns in the table definition: default.p2 (1 vs 2)
```

Related impala-shell options:

Due to the expense of reloading the metadata for all tables, the `impala-shell` -r option is not recommended.

HDFS considerations:

All HDFS and Ranger permissions and privilege requirements are the same whether you refresh the entire table or a single partition.

The REFRESH statement checks HDFS permissions of the underlying data files and directories, caching this information so that a statement can be cancelled immediately if for example the impala user does not have permission to write to the data directory for the table. Impala reports any lack of write permissions as an INFO message in the log file.

If you change HDFS permissions to make data readable or writeable by the Impala user, issue another REFRESH to make Impala aware of the change.

Kudu considerations:

By default, much of the metadata for Kudu tables is handled by the underlying storage layer. Kudu tables have less reliance on the Metastore database, and require less metadata caching on the Impala side. For example, information about partitions in Kudu tables is managed by Kudu, and Impala does not cache any block locality metadata for Kudu tables. If the Kudu service is not integrated with the Hive Metastore, Impala will manage Kudu table metadata in the Hive Metastore.

The REFRESH and INVALIDATE METADATA statements are needed less frequently for Kudu tables than for HDFS-backed tables. Neither statement is needed when data is added to, removed, or updated in a Kudu table, even if the changes are made directly to Kudu through a client program using the Kudu API. Run REFRESH *table_name* or INVALIDATE METADATA *table_name* for a Kudu table only after making a change to the Kudu table schema, such as adding or dropping a column.

REFRESH AUTHORIZATION statement

The REFRESH AUTHORIZATION statement explicitly refreshes authorization data, including privileges and principals. When there is an external update to authorization metadata, use this statement to force Impala to refresh its authorization data without having to wait for the Ranger polling or run INVALIDATE METADATA.

Once issued, the REFRESH AUTHORIZATION statement cannot be cancelled.

Syntax:

```
REFRESH AUTHORIZATION
```

Usage notes: If authorization is not enabled, Impala returns an error.

Added in: Impala 3.2

Related Information

Enabling and using authorization in Impala

REFRESH FUNCTIONS statement

In Impala 2.9 and higher, you can run the REFRESH FUNCTIONS statement to refresh the user-defined functions (UDFs) created outside of Impala. For example, you can add Java-based UDFs to the metastore database through the Hive CREATE FUNCTION statements and make those UDFs visible to Impala at the database level by subsequently running REFRESH FUNCTIONS.

The REFRESH FUNCTIONS statement is only required if you create or modify UDFs from outside of Impala. Updated metadata, as a result of running REFRESH FUNCTIONS, is broadcast to all Impala coordinators.

Once issued, the REFRESH FUNCTIONS statement cannot be cancelled.

Syntax:

REFRESH FUNCTIONS *db_name*

REVOKE statement

The REVOKE statement revokes privileges on a specified object from groups or users.

Syntax:

```
REVOKE privilege ON object_type object_name
  FROM USER user_name

REVOKE privilege ON object_type object_name
  FROM GROUP group_name


  privilege ::= ALL | ALTER | CREATE | DROP | INSERT | REFRESH | SELECT |
 SELECT(column_name)

object_type ::= SERVER | URI | DATABASE | TABLE
```

See the GRANT statement for the required privileges and the scope for SQL operations.

The ALL privilege is a distinct privilege and not a union of all other privileges.

The following syntax is supported when Impala is using Ranger to manage authorization.

```
REVOKE privilege ON object_type object_name
  FROM USER user_name

REVOKE privilege ON object_type object_name
  FROM GROUP group_name


  privilege ::= ALL | ALTER | CREATE | DROP | INSERT | REFRESH | SELECT |
 SELECT(column_name)

object_type ::= SERVER | URI | DATABASE | TABLE
```

You cannot revoke a privilege granted with the WITH GRANT OPTION. If a privilege is granted with the WITH GRANT OPTION, first revoke the grant option, and then revoke the privilege.

Typically, the object name is an identifier. For URIs, it is a string literal.

The ability to grant or revoke SELECT privilege on specific columns is available in Impala 2.3 and higher.

Required privileges:

Only administrative users those with ALL privileges on the server can use this statement.

Compatibility:

- The Impala REVOKE statement is available in Impala 2.0 and higher.

- In Impala 1.4 and higher, Impala makes use of any privileges specified by the GRANT and REVOKE statements in Hive, when your system is configured to use the Ranger service.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related Information

GRANT statement

SELECT statement

The SELECT statement performs queries, retrieving data from one or more tables and producing result sets consisting of rows and columns.

The Impala INSERT statement also typically ends with a SELECT statement, to define data to copy from one table to another.

Syntax:

```
[WITH name AS (select_expression) [, ...] ]
SELECT
  [ALL | DISTINCT]
  [STRAIGHT_JOIN]
  expression [, expression ...]
FROM table_reference [, table_reference ...]
[[FULL | [LEFT | RIGHT] INNER | [LEFT | RIGHT] OUTER | [LEFT | RIGHT] SEMI |
 [LEFT | RIGHT] ANTI | CROSS]
  JOIN table_reference
  [ON join_equality_clauses | USING (col1[, col2 ...]] ...
WHERE conditions
GROUP BY { column | expression [, ...] }
HAVING conditions
ORDER BY { column | expression [ASC | DESC] [NULLS FIRST | NULLS LAST] [,
...] }
LIMIT expression [OFFSET expression]
[UNION [ALL] select_statement] ...]

table_reference := { table_name | (subquery) }
  [ TABLESAMPLE SYSTEM(percentage) [REPEATABLE(seed)] ]
```

Impala SELECT queries support:

- SQL scalar data types: BOOLEAN, TINYINT, SMALLINT, INT, BIGINT, DECIMAL FLOAT, DOUBLE, TIMESTAMP, STRING, VARCHAR, CHAR.
- The complex data types ARRAY, STRUCT, and MAP, are available in Impala 2.3 and higher. Queries involving these types typically involve special qualified names using dot notation for referring to the complex column fields, and join clauses for bringing the complex columns into the result set.
- An optional WITH clause before the SELECT keyword, to define a subquery whose name or column names can be referenced from later in the main query. This clause lets you abstract repeated clauses, such as aggregation functions, that are referenced multiple times in the same query.
- Subqueries in a FROM clause. In Impala 2.0 and higher, subqueries can also go in the WHERE clause, for example with the IN(), EXISTS, and NOT EXISTS operators.
- WHERE, GROUP BY, HAVING clauses.
- ORDER BY. Prior to Impala 1.4.0, Impala required that queries using an ORDER BY clause also include a LIMIT clause. In Impala 1.4.0 and higher, this restriction is lifted; sort operations that would exceed the Impala memory limit automatically use a temporary disk work area to perform the sort.

- You can refer to SELECT-list items by their ordinal position. Impala supports ordinals in the GROUP BY, HAVING, and ORDER BY clauses. From Impala 3.0, ordinals can only be used at the top level. For example, the following statements are allowed:

```
  SELECT int_col / 2, sum(x)
  FROM t
  GROUP BY 1;

  SELECT int_col / 2
  FROM t
  ORDER BY 1;

  SELECT NOT bool_col
  FROM t
  GROUP BY 1
  HAVING 1;
```

  Numbers in subexpressions are not interpreted as ordinals:

```
  SELECT int_col / 2, sum(x)
  FROM t
  GROUP BY 1 * 2;
The above parses OK, however GROUP BY 1 * 2 has no effect.
  SELECT int_col / 2
  FROM t
  ORDER BY 1 + 2;
The above parses OK, however ORDER BY 1 + 2 has no effect.
  SELECT NOT bool_col
  FROM t
  GROUP BY 1
  HAVING not 1;
The above raises an error at parse-time.
```

- Impala supports a wide variety of JOIN clauses. Left, right, semi, full, and outer joins are supported in all Impala versions. The CROSS JOIN operator is available in Impala 1.2.2 and higher. During performance tuning, you can override the reordering of join clauses that Impala does internally by including the keyword STRAIGHT_JOIN immediately after the SELECT and any DISTINCT or ALL keywords.
- UNION ALL.
- LIMIT.
- External tables.
- Relational operators such as greater than, less than, or equal to.
- Arithmetic operators such as addition or subtraction.
- Logical/Boolean operators AND, OR, and NOT. Impala does not support the corresponding symbols &&, ||, and !.
- Common SQL built-in functions such as COUNT, SUM, CAST, LIKE, IN, BETWEEN, and COALESCE.
- On optional TABLESAMPLE clause immediately after a table reference, to specify that the query only processes a specified percentage of the table data.

Impala queries ignore files with extensions commonly used for temporary work files by Hadoop tools. Any files with extensions .tmp or .copying are not considered part of the Impala table. The suffix matching is case-insensitive, so for example Impala ignores both .copying and .COPYING suffixes.

Security considerations:

If these statements in your environment contain sensitive literal values such as credit card numbers or tax identifiers, Impala can redact this sensitive information when displaying the statements in log files and other administrative contexts.

Amazon S3 considerations:

In Impala 2.6 and higher, Impala queries are optimized for files stored in Amazon S3. For Impala tables that use the file formats Parquet, ORC, RCFile, SequenceFile, Avro, and uncompressed text, the setting fs.s3a.block.size in the core-site.xml configuration file determines how Impala divides the I/O work of reading the data files. This configuration setting is specified in bytes. By default, this value is 33554432 (32 MB), meaning that Impala parallelizes S3 read operations on the files as if they were made up of 32 MB blocks. For example, if your S3 queries primarily access Parquet files written by MapReduce or Hive, increase fs.s3a.block.size to 134217728 (128 MB) to match the row group size of those files. If most S3 queries involve Parquet files written by Impala, increase fs.s3a.block.size to 268435456 (256 MB) to match the row group size produced by Impala.

Cancellation: Can be cancelled. To cancel this statement, use Ctrl-C from the `impala-shell` interpreter, the Cancel button from the Watch page in Hue, Actions > Cancel from the Queries list in Cloudera Manager, or Cancel from the list of in-flight queries (for a particular node) on the Queries tab in the Impala web UI (port 25000).

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read permissions for the files in all applicable directories in all source tables, and read and execute permissions for the relevant data directories. (A SELECT operation could read files from multiple different HDFS directories if the source table is partitioned.) If a query attempts to read a data file and is unable to because of an HDFS permission error, the query halts and does not return any further results.

Related information:

The SELECT syntax is so extensive that it forms its own category of statements: queries. The other major classifications of SQL statements are data definition language and data manipulation language.

Because the focus of Impala is on fast queries with interactive response times over huge data sets, query performance and scalability are important considerations.

Joins in Impala SELECT statements

A join query is a SELECT statement that combines data from two or more tables, and returns a result set containing items from some or all of those tables. It is a way to cross-reference and correlate related data that is organized into multiple tables, typically using identifiers that are repeated in each of the joined tables.

Syntax:

Impala supports a wide variety of JOIN clauses. Left, right, semi, full, and outer joins are supported in all Impala versions. The CROSS JOIN operator is available in Impala 1.2.2 and higher. During performance tuning, you can override the reordering of join clauses that Impala does internally by including the keyword STRAIGHT_JOIN immediately after the SELECT and any DISTINCT or ALL keywords.

```
SELECT select_list FROM
  table_or_subquery1 [INNER] JOIN table_or_subquery2 |
  table_or_subquery1 {LEFT [OUTER] | RIGHT [OUTER] | FULL [OUTER]} J
OIN table_or_subquery2 |
  table_or_subquery1 {LEFT | RIGHT} SEMI JOIN table_or_subquery2 |
  table_or_subquery1 {LEFT | RIGHT} ANTI JOIN table_or_subquery2 |
    [ ON col1 = col2 [AND col3 = col4 ...] |
      USING (col1 [, col2 ...]) ]
  [other_join_clause ...]
[ WHERE where_clauses ]

SELECT select_list FROM
  table_or_subquery1, table_or_subquery2 [, table_or_subquery3 ...]
  [other_join_clause ...]
WHERE
    col1 = col2 [AND col3 = col4 ...]

SELECT select_list FROM
  table_or_subquery1 CROSS JOIN table_or_subquery2
  [other_join_clause ...]
[ WHERE where_clauses ]
```

SQL-92 and SQL-89 Joins:

Queries with the explicit JOIN keywords are known as SQL-92 style joins, referring to the level of the SQL standard where they were introduced. The corresponding ON or USING clauses clearly show which columns are used as the join keys in each case:

```
SELECT t1.c1, t2.c2 FROM t1 JOIN t2
  ON t1.id = t2.id and t1.type_flag = t2.type_flag
  WHERE t1.c1 > 100;

SELECT t1.c1, t2.c2 FROM t1 JOIN t2
  USING (id, type_flag)
  WHERE t1.c1 > 100;
```

The ON clause is a general way to compare columns across the two tables, even if the column names are different. The USING clause is a shorthand notation for specifying the join columns, when the column names are the same in both tables. You can code equivalent WHERE clauses that compare the columns, instead of ON or USING clauses, but that practice is not recommended because mixing the join comparisons with other filtering clauses is typically less readable and harder to maintain.

Queries with a comma-separated list of tables and subqueries are known as SQL-89 style joins. In these queries, the equality comparisons between columns of the joined tables go in the WHERE clause alongside other kinds of comparisons. This syntax is easy to learn, but it is also easy to accidentally remove a WHERE clause needed for the join to work correctly.

```
SELECT t1.c1, t2.c2 FROM t1, t2
  WHERE
  t1.id = t2.id AND t1.type_flag = t2.type_flag
  AND t1.c1 > 100;
```

Self-joins:

Impala can do self-joins, for example to join on two different columns in the same table to represent parent-child relationships or other tree-structured data. There is no explicit syntax for this; just use the same table name for both the left-hand and right-hand table, and assign different table aliases to use when referring to the fully qualified column names:

```
-- Combine fields from both parent and child rows.
SELECT lhs.id, rhs.parent, lhs.c1, rhs.c2 FROM tree_data lhs, tree_data rhs
 WHERE lhs.id = rhs.parent;
```

Inner and outer joins:

An inner join is the most common and familiar type: rows in the result set contain the requested columns from the appropriate tables, for all combinations of rows where the join columns of the tables have identical values. If a column with the same name occurs in both tables, use a fully qualified name or a column alias to refer to the column in the select list or other clauses. Impala performs inner joins by default for both SQL-89 and SQL-92 join syntax:

```
-- The following 3 forms are all equivalent.
SELECT t1.id, c1, c2 FROM t1, t2 WHERE t1.id = t2.id;
SELECT t1.id, c1, c2 FROM t1 JOIN t2 ON t1.id = t2.id;
SELECT t1.id, c1, c2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

An outer join retrieves all rows from the left-hand table, or the right-hand table, or both; wherever there is no matching data in the table on the other side of the join, the corresponding columns in the result set are set to NULL. To perform an outer join, include the OUTER keyword in the join operator, along with either LEFT, RIGHT, or FULL:

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.id = t2.id;
SELECT * FROM t1 RIGHT OUTER JOIN t2 ON t1.id = t2.id;
```

```
SELECT * FROM t1 FULL OUTER JOIN t2 ON t1.id = t2.id;
```

For outer joins, Impala requires SQL-92 syntax; that is, the JOIN keyword instead of comma-separated table names. Impala does not support vendor extensions such as (+) or *= notation for doing outer joins with SQL-89 query syntax.

Equijoins and Non-Equijoins:

By default, Impala requires an equality comparison between the left-hand and right-hand tables, either through ON, USING, or WHERE clauses. These types of queries are classified broadly as equijoins. Inner, outer, full, and semi joins can all be equijoins based on the presence of equality tests between columns in the left-hand and right-hand tables.

In Impala 1.2.2 and higher, non-equijoin queries are also possible, with comparisons such as != or < between the join columns. These kinds of queries require care to avoid producing huge result sets that could exceed resource limits. Once you have planned a non-equijoin query that produces a result set of acceptable size, you can code the query using the CROSS JOIN operator, and add the extra comparisons in the WHERE clause:

```
SELECT * FROM t1 CROSS JOIN t2 WHERE t1.total > t2.maximum_price;
```

In Impala 2.3 and higher, additional non-equijoin queries are possible due to the addition of nested loop joins. These queries typically involve SEMI JOIN, ANTI JOIN, or FULL OUTER JOIN clauses. Impala sometimes also uses nested loop joins internally when evaluating OUTER JOIN queries involving complex type columns. Query phases involving nested loop joins do not use the spill-to-disk mechanism if they exceed the memory limit. Impala decides internally when to use each join mechanism; you cannot specify any query hint to choose between the nested loop join or the original hash join algorithm.

```
SELECT * FROM t1 LEFT OUTER JOIN t2 ON t1.int_col < t2.int_col;
```

Semi-joins:

Semi-joins are a relatively rarely used variation. With the left semi-join, only data from the left-hand table is returned, for rows where there is matching data in the right-hand table, based on comparisons between join columns in ON or WHERE clauses. Only one instance of each row from the left-hand table is returned, regardless of how many matching rows exist in the right-hand table. A right semi-join (available in Impala 2.0 and higher) reverses the comparison and returns data from the right-hand table.

```
SELECT t1.c1, t1.c2, t1.c2 FROM t1 LEFT SEMI JOIN t2 ON t1.id = t2.id;
```

Natural joins (not supported):

Impala does not support the NATURAL JOIN operator, again to avoid inconsistent or huge result sets. Natural joins do away with the ON and USING clauses, and instead automatically join on all columns with the same names in the left-hand and right-hand tables. This kind of query is not recommended for rapidly evolving data structures such as are typically used in Hadoop. Thus, Impala does not support the NATURAL JOIN syntax, which can produce different query results as columns are added to or removed from tables.

If you do have any queries that use NATURAL JOIN, make sure to rewrite them with explicit USING clauses, because Impala could interpret the NATURAL keyword as a table alias:

```
-- 'NATURAL' is interpreted as an alias for 't1' and Impala attempts an inne
r join,
-- resulting in an error because inner joins require explicit comparisons be
tween columns.
SELECT t1.c1, t2.c2 FROM t1 NATURAL JOIN t2;
ERROR: NotImplementedException: Join with 't2' requires at least one conjunc
tive equality predicate.
  To perform a Cartesian product between two tables, use a CROSS JOIN.
-- If you expect the tables to have identically named columns with matching
 values,
-- list the corresponding column names in a USING clause.
```

```
SELECT t1.c1, t2.c2 FROM t1 JOIN t2 USING (id, type_flag, name, address);
```

Anti-joins (Impala 2.0 and higher only):

Impala supports the LEFT ANTI JOIN and RIGHT ANTI JOIN clauses in Impala 2.0 and higher. The LEFT or RIGHT keyword is required for this kind of join. For LEFT ANTI JOIN, this clause returns those values from the left-hand table that have no matching value in the right-hand table. RIGHT ANTI JOIN reverses the comparison and returns values from the right-hand table. You can express this negative relationship either through the ANTI JOIN clause or through a NOT EXISTS operator with a subquery.

Complex type considerations:

When referring to a column with a complex type (STRUCT, ARRAY, or MAP) in a query, you use join notation to “unpack” the scalar fields of the struct, the elements of the array, or the key-value pairs of the map. (The join notation is not required for aggregation operations, such as COUNT() or SUM() for array elements.) Because Impala recognizes which complex type elements are associated with which row of the result set, you use the same syntax as for a cross or cartesian join, without an explicit join condition.

Usage notes:

You typically use join queries in situations like these:

- When related data arrives from different sources, with each data set physically residing in a separate table. For example, you might have address data from business records that you cross-check against phone listings or census data.

  

  Note: Impala can join tables of different file formats, including Impala-managed tables and HBase tables. For example, you might keep small dimension tables in HBase, for convenience of single-row lookups and updates, and for the larger fact tables use Parquet or other binary file format optimized for scan operations. Then, you can issue a join query to cross-reference the fact tables with the dimension tables.
- When data is normalized, a technique for reducing data duplication by dividing it across multiple tables. This kind of organization is often found in data that comes from traditional relational database systems. For example, instead of repeating some long string such as a customer name in multiple tables, each table might contain a numeric customer ID. Queries that need to display the customer name could “join” the table that specifies which customer ID corresponds to which name.
- When certain columns are rarely needed for queries, so they are moved into separate tables to reduce overhead for common queries. For example, a biography field might be rarely needed in queries on employee data. Putting that field in a separate table reduces the amount of I/O for common queries on employee addresses or phone numbers. Queries that do need the biography column can retrieve it by performing a join with that separate table.
- In Impala 2.3 or higher, when referring to complex type columns in queries.

When comparing columns with the same names in ON or WHERE clauses, use the fully qualified names such as *db_name*.*table_name*, or assign table aliases, column aliases, or both to make the code more compact and understandable:

```
select t1.c1 as first_id, t2.c2 as second_id from
  t1 join t2 on first_id = second_id;
select fact.custno, dimension.custno from
  customer_data as fact join customer_address as dimension
  using (custno)
```

Note:

Performance for join queries is a crucial aspect for Impala, because complex join queries are resource-intensive operations. An efficient join query produces much less network traffic and CPU overhead than an inefficient one. For best results:

- Make sure that both table and column statistics are available for all the tables involved in a join query, and especially for the columns referenced in any join conditions. Impala uses the statistics to automatically deduce an efficient join order. Use SHOW TABLE STATS *table_name* and SHOW COLUMN ST ATS *table_name* to check if statistics are already present. Issue the COMPUTE STATS *table_name* for a nonpartitioned table, or (in Impala 2.1.0 and higher) COMPUTE INCREMENTAL STATS *table_name* for a partitioned table, to collect the initial statistics at both the table and column levels, and to keep the statistics up to date after any substantial INSERT or LOAD DATA operations.
- If table or column statistics are not available, join the largest table first. You can check the existence of statistics with the SHOW TABLE STATS *table_name* and SHOW COLUMN STATS *table_name* statements.
- If table or column statistics are not available, join subsequent tables according to which table has the most selective filter, based on overall size and WHERE clauses. Joining the table with the most selective filter results in the fewest number of rows being returned.

To control the result set from a join query, include the names of corresponding column names in both tables in an ON or USING clause, or by coding equality comparisons for those columns in the WHERE clause.

```
[localhost:21000] > select c_last_name, ca_city from customer join customer_
address where c_customer_sk = ca_address_sk;
+-------------+-----------------+
| c_last_name | ca_city         |
+-------------+-----------------+
| Lewis       | Fairfield       |
| Moses       | Fairview        |
| Hamilton    | Pleasant Valley |
| White       | Oak Ridge       |
| Moran       | Glendale        |
...
| Richards    | Lakewood         |
| Day         | Lebanon          |
| Painter     | Oak Hill         |
| Bentley     | Greenfield       |
| Jones       | Stringtown       |
+-------------+------------------+
Returned 50000 row(s) in 9.82s
```

One potential downside of joins is the possibility of excess resource usage in poorly constructed queries. Impala imposes restrictions on join queries to guard against such issues. To minimize the chance of runaway queries on large data sets, Impala requires every join query to contain at least one equality predicate between the columns of the various tables. For example, if T1 contains 1000 rows and T2 contains 1,000,000 rows, a query SELECT *columns* FROM t1 JOIN t2 could return up to 1 billion rows (1000 * 1,000,000); Impala requires that the query include a clause such as ON t1.c1 = t2.c2 or WHERE t1.c1 = t2.c2.

Because even with equality clauses, the result set can still be large, as we saw in the previous example, you might use a LIMIT clause to return a subset of the results:

```
[localhost:21000] > select c_last_name, ca_city from customer, customer_addr
ess where c_customer_sk = ca_address_sk limit 10;
+-------------+-----------------+
| c_last_name | ca_city         |
+-------------+-----------------+
| Lewis       | Fairfield       |
| Moses       | Fairview        |
| Hamilton    | Pleasant Valley |
| White       | Oak Ridge       |
```

```
| Moran       | Glendale        |
| Sharp       | Lakeview        |
| Wiles       | Farmington      |
| Shipman     | Union           |
| Gilbert     | New Hope        |
| Brunson     | Martinsville    |
+-------------+-----------------+
Returned 10 row(s) in 0.63s
```

Or you might use additional comparison operators or aggregation functions to condense a large result set into a smaller set of values:

```
[localhost:21000] > -- Find the names of customers who live in one particula
r town.
[localhost:21000] > select distinct c_last_name from customer, customer_addr
ess where
  c_customer_sk = ca_address_sk
  and ca_city = "Green Acres";
+---------------+
| c_last_name   |
+---------------+
| Hensley       |
| Pearson       |
| Mayer         |
| Montgomery    |
| Ricks         |
...
| Barrett       |
| Price         |
| Hill          |
| Hansen        |
| Meeks         |
+---------------+
Returned 332 row(s) in 0.97s

[localhost:21000] > -- See how many different customers in this town have
 names starting with "A".
[localhost:21000] > select count(distinct c_last_name) from customer, cust
omer_address where
  c_customer_sk = ca_address_sk
  and ca_city = "Green Acres"
  and substr(c_last_name,1,1) = "A";
+-----------------------------+
| count(distinct c_last_name) |
+-----------------------------+
| 12                          |
+-----------------------------+
Returned 1 row(s) in 1.00s
```

Because a join query can involve reading large amounts of data from disk, sending large amounts of data across the network, and loading large amounts of data into memory to do the comparisons and filtering, you might do benchmarking, performance analysis, and query tuning to find the most efficient join queries for your data set, hardware capacity, network configuration, and cluster workload.

The two categories of joins in Impala are known as partitioned joins and broadcast joins. If inaccurate table or column statistics, or some quirk of the data distribution, causes Impala to choose the wrong mechanism for a particular join, consider using query hints as a temporary workaround.

Handling NULLs in join columns:

By default, join key columns do not match if either one contains a NULL value. To treat such columns as equal if both contain NULL, you can use an expression such as A = B OR (A IS NULL AND B IS NULL). In Impala 2.5 and higher, the <=> operator (shorthand for IS NOT DISTINCT FROM) performs the same comparison in a concise

and efficient form. The <=> operator is more efficient in for comparing join keys in a NULL-safe manner, because the operator can use a hash join while the OR expression cannot.

Examples:

The following examples refer to these simple tables containing small sets of integers:

```
[localhost:21000] > create table t1 (x int);
[localhost:21000] > insert into t1 values (1), (2), (3), (4), (5), (6);

[localhost:21000] > create table t2 (y int);
[localhost:21000] > insert into t2 values (2), (4), (6);
[localhost:21000] > create table t3 (z int);
[localhost:21000] > insert into t3 values (1), (3), (5);
```

The following example demonstrates an anti-join, returning the values from T1 that do not exist in T2 (in this case, the odd numbers 1, 3, and 5):

```
[localhost:21000] > select x from t1 left anti join t2 on (t1.x = t2.y);
+---+
| x |
+---+
| 1 |
| 3 |
| 5 |
+---+
```

Related Information

Optimizer hints in Impala

Performance considerations for join queries

SHOW statement

ORDER BY clause

The ORDER BY clause of a SELECT statement sorts the result set based on the values from one or more columns.

First, data is sorted locally by each impalad daemon, then streamed to the coordinator daemon, which merges the sorted result sets. For distributed queries, this is a relatively expensive operation and can require more memory capacity than a query without ORDER BY. Even if the query takes approximately the same time to finish with or without the ORDER BY clause, subjectively it can appear slower because no results are available until all processing is finished, rather than results coming back gradually as rows matching the WHERE clause are found. Therefore, if you only need the first N results from the sorted result set, also include the LIMIT clause, which reduces network overhead and the memory requirement on the coordinator node.

Syntax:

The full syntax for the ORDER BY clause is:

```
ORDER BY col_ref [, col_ref ...] [ASC | DESC] [NULLS FIRST | NULLS LAST]

col_ref ::= column_name | integer_literal
```

Although the most common usage is ORDER BY *column_name*, you can also specify ORDER BY 1 to sort by the first column of the result set, ORDER BY 2 to sort by the second column, and so on. The number must be a numeric literal, not some other kind of constant expression. (If the argument is some other expression, even a STRING value, the query succeeds but the order of results is undefined.)

ORDER BY *column_number* can only be used when the query explicitly lists the columns in the SELECT list, not with SELECT * queries.

Ascending and descending sorts:

The default sort order (the same as using the ASC keyword) puts the smallest values at the start of the result set, and the largest values at the end. Specifying the DESC keyword reverses that order.

Sort order for NULL values:

See the NULL on page 84 section for details about how NULL values are positioned in the sorted result set, and how to use the NULLS FIRST and NULLS LAST clauses. (The sort position for NULL values in ORDER BY ... DESC queries is changed in Impala 1.2.1 and higher to be more standards-compliant, and the NULLS FIRST and NULLS LAST keywords are new in Impala 1.2.1.)

Prior to Impala 1.4.0, Impala required any query including an ORDER_BY clause to also use a LIMIT clause. In Impala 1.4.0 and higher, the LIMIT clause is optional for ORDER BY queries. In cases where sorting a huge result set requires enough memory to exceed the Impala memory limit for a particular executor Impala daemon, Impala automatically uses a temporary disk work area to perform the sort operation.

Complex type considerations:

In Impala 2.3 and higher, the complex data types STRUCT, ARRAY, and MAP are available. These columns cannot be referenced directly in the ORDER BY clause. When you query a complex type column, you use join notation to “unpack” the elements of the complex type, and within the join query you can include an ORDER BY clause to control the order in the result set of the scalar elements from the complex type.

The following query shows how a complex type column cannot be directly used in an ORDER BY clause:

```
CREATE TABLE games (id BIGINT, score ARRAY <BIGINT>) STORED AS PARQUET;
...use LOAD DATA to load externally created Parquet files into the table...
SELECT id FROM games ORDER BY score DESC;
ERROR: AnalysisException: ORDER BY expression 'score' with complex type 'A
RRAY<BIGINT>' is not supported.
```

Examples:

The following query retrieves the user ID and score, only for scores greater than one million, with the highest scores for each user listed first. Because the individual array elements are now represented as separate rows in the result set, they can be used in the ORDER BY clause, referenced using the ITEM pseudo-column that represents each array element.

```
SELECT id, item FROM games, games.score
  WHERE item > 1000000
ORDER BY id, item desc;
```

The following queries use similar ORDER BY techniques with variations of the GAMES table, where the complex type is an ARRAY containing STRUCT or MAP elements to represent additional details about each game that was played. For an array of structures, the fields of the structure are referenced as ITEM.*field_name*. For an array of maps, the keys and values within each array element are referenced as ITEM.KEY and ITEM.VALUE.

```
CREATE TABLE games2 (id BIGINT, play array < struct <game_name: string, scor
e: BIGINT, high_score: boolean> >) STORED AS PARQUET
...use LOAD DATA to load externally created Parquet files into the table...
SELECT id, item.game_name, item.score FROM games2, games2.play
  WHERE item.score > 1000000
ORDER BY id, item.score DESC;

CREATE TABLE games3 (id BIGINT, play ARRAY < MAP <STRING, BIGINT> >) STORED
AS PARQUET;
...use LOAD DATA to load externally created Parquet files into the table...
SELECT id, info.key AS k, info.value AS v from games3, games3.play AS plays,
 games3.play.item AS info
  WHERE info.KEY = 'score' AND info.VALUE > 1000000
ORDER BY id, info.value desc;
```

Usage notes:

Although the LIMIT clause is now optional on ORDER BY queries, if your query only needs some number of rows that you can predict in advance, use the LIMIT clause to reduce unnecessary processing. For example, if the query has a clause LIMIT 10, each executor Impala daemon sorts its portion of the relevant result set and only returns 10 rows to the coordinator node. The coordinator node picks the 10 highest or lowest row values out of this small intermediate result set.

If an ORDER BY clause is applied to an early phase of query processing, such as a subquery or a view definition, Impala ignores the ORDER BY clause. To get ordered results from a subquery or view, apply an ORDER BY clause to the outermost or final SELECT level.

ORDER BY is often used in combination with LIMIT to perform “top-N” queries:

```
SELECT user_id AS "Top 10 Visitors", SUM(page_views) FROM web_stats
  GROUP BY page_views, user_id
  ORDER BY SUM(page_views) DESC LIMIT 10;
```

ORDER BY is sometimes used in combination with OFFSET and LIMIT to paginate query results, although it is relatively inefficient to issue multiple queries like this against the large tables typically used with Impala:

```
SELECT page_title AS "Page 1 of search results", page_url FROM search_conten
t
  WHERE LOWER(page_title) LIKE '%game%')
  ORDER BY page_title LIMIT 10 OFFSET 0;
SELECT page_title AS "Page 2 of search results", page_url FROM search_conten
t
  WHERE LOWER(page_title) LIKE '%game%')
  ORDER BY page_title LIMIT 10 OFFSET 10;
SELECT page_title AS "Page 3 of search results", page_url FROM search_conte
nt
  WHERE LOWER(page_title) LIKE '%game%')
  ORDER BY page_title LIMIT 10 OFFSET 20;
```

Internal details:

Impala sorts the intermediate results of an ORDER BY clause in memory whenever practical. In a cluster of N executor Impala daemons, each daemon sorts roughly 1/Nth of the result set, the exact proportion varying depending on how the data matching the query is distributed in HDFS.

If the size of the sorted intermediate result set on any executor Impala daemon would cause the query to exceed the Impala memory limit, Impala sorts as much as practical in memory, then writes partially sorted data to disk. (This technique is known in industry terminology as “external sorting” and “spilling to disk”.) As each 8 MB batch of data is written to disk, Impala frees the corresponding memory to sort a new 8 MB batch of data. When all the data has been processed, a final merge sort operation is performed to correctly order the in-memory and on-disk results as the result set is transmitted back to the coordinator node. When external sorting becomes necessary, Impala requires approximately 60 MB of RAM at a minimum for the buffers needed to read, write, and sort the intermediate results. If more RAM is available on the Impala daemon, Impala will use the additional RAM to minimize the amount of disk I/O for sorting.

This external sort technique is used as appropriate on each Impala daemon (possibly including the coordinator node) to sort the portion of the result set that is processed on that node. When the sorted intermediate results are sent back to the coordinator node to produce the final result set, the coordinator node uses a merge sort technique to produce a final sorted result set without using any extra resources on the coordinator node.

Sorting considerations:

Although you can specify an ORDER BY clause in an INSERT ... SELECT statement, any ORDER BY clause is ignored, and the results are not necessarily sorted.

An ORDER BY clause without an additional LIMIT clause is ignored in any view definition. If you need to sort the entire result set from a view, use an ORDER BY clause in the SELECT statement that queries the view. You can still make a simple “top 10” report by combining the ORDER BY and LIMIT clauses in the same view definition:

```
[localhost:21000] > create table unsorted (x bigint);
[localhost:21000] > insert into unsorted values (1), (9), (3), (7), (5), (
8), (4), (6), (2);
[localhost:21000] > create view sorted_view as select x from unsorted order
 by x;
[localhost:21000] > select x from sorted_view; -- ORDER BY clause in view h
as no effect.
+---+
| x |
+---+
| 1 |
| 9 |
| 3 |
| 7 |
| 5 |
| 8 |
| 4 |
| 6 |
| 2 |
+---+
[localhost:21000] > select x from sorted_view order by x; -- View query requ
ires ORDER BY at outermost level.
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
+---+
[localhost:21000] > create view top_3_view as select x from unsorted order b
y x limit 3;
[localhost:21000] > select x from top_3_view; -- ORDER BY and LIMIT together
 in view definition are preserved.
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

With the lifting of the requirement to include a LIMIT clause in every ORDER BY query (in Impala 1.4 and higher):

- Now the use of scratch disk space raises the possibility of an “out of disk space” error on a particular Impala daemon, as opposed to the previous possibility of an “out of memory” error. Make sure to keep at least 1 GB free on the filesystem used for temporary sorting work.

In Impala 1.2.1 and higher, all NULL values come at the end of the result set for ORDER BY ... ASC queries, and at the beginning of the result set for ORDER BY ... DESC queries. In effect, NULL is considered greater than all other values for sorting purposes. The original Impala behavior always put NULL values at the end, even for ORDER BY ... DESC queries. The new behavior in Impala 1.2.1 makes Impala more compatible with other popular database

systems. In Impala 1.2.1 and higher, you can override or specify the sorting behavior for NULL by adding the clause NULLS FIRST or NULLS LAST at the end of the ORDER BY clause.

```
[localhost:21000] > create table numbers (x int);
[localhost:21000] > insert into numbers values (1), (null), (2), (null), (
3);
[localhost:21000] > select x from numbers order by x nulls first;
+------+
| x    |
+------+
| NULL |
| NULL |
| 1    |
| 2    |
| 3    |
+------+
[localhost:21000] > select x from numbers order by x desc nulls first;
+------+
| x    |
+------+
| NULL |
| NULL |
| 3    |
| 2    |
| 1    |
+------+
[localhost:21000] > select x from numbers order by x nulls last;
+------+
| x    |
+------+
| 1    |
| 2    |
| 3    |
| NULL |
| NULL |
+------+
[localhost:21000] > select x from numbers order by x desc nulls last;
+------+
| x    |
+------+
| 3    |
| 2    |
| 1    |
| NULL |
| NULL |
+------+
```

Related information:

See the SELECT statement for further examples of queries with the ORDER BY clause.

Analytic functions use the ORDER BY clause in a different context to define the sequence in which rows are analyzed. See Impala analytic functions on page 384 for details.

GROUP BY clause

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.

Complex type considerations:

In Impala 2.3 and higher, the complex data types STRUCT, ARRAY, and MAP are available. These columns cannot be referenced directly in the ORDER BY clause. When you query a complex type column, you use join notation

to “unpack” the elements of the complex type, and within the join query you can include an ORDER BY clause to control the order in the result set of the scalar elements from the complex type.

Zero-length strings: For purposes of clauses such as DISTINCT and GROUP BY, Impala considers zero-length strings (""), NULL, and space to all be different values.

Examples:

For example, the following query finds the 5 items that sold the highest total quantity (using the SUM() function, and also counts the number of sales transactions for those items (using the COUNT() function). Because the column representing the item IDs is not used in any aggregation functions, we specify that column in the GROUP BY clause.

```
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
  order by sum(ss_quantity) desc
  limit 5;
+-------+-----------------+--------------------------+
| item  | times_purchased | total_quantity_purchased |
+-------+-----------------+--------------------------+
| 9325  | 372             | 19072                    |
| 4279  | 357             | 18501                    |
| 7507  | 371             | 18475                    |
| 5953  | 369             | 18451                    |
| 16753 | 375             | 18446                    |
+-------+-----------------+--------------------------+
```

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:

```
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;
+-------+-----------------+--------------------------+
| item  | times_purchased | total_quantity_purchased |
+-------+-----------------+--------------------------+
| 13943 | 105             | 4087                     |
| 2992  | 101             | 4176                     |
| 4773  | 107             | 4204                     |
| 14350 | 103             | 4260                     |
| 11956 | 102             | 4275                     |
+-------+-----------------+--------------------------+
```

When performing calculations involving scientific or financial data, remember that columns with type FLOAT or DOUBLE are stored as true floating-point numbers, which cannot precisely represent every possible fractional value. Thus, if you include a FLOAT or DOUBLE column in a GROUP BY clause, the results might not precisely match literal values in your query or from an original Text data file. Use rounding operations, the BETWEEN operator, or another arithmetic technique to match floating-point values that are “near” literal values you expect. For example, this

query on the ss_wholesale_cost column returns cost values that are close but not identical to the original figures that were entered as decimal fractions.

```
select ss_wholesale_cost, avg(ss_quantity * ss_sales_price) as avg_revenue_p
er_sale
  from sales
  group by ss_wholesale_cost
  order by avg_revenue_per_sale desc
  limit 5;
+-------------------+----------------------+
| ss_wholesale_cost | avg_revenue_per_sale |
+-------------------+----------------------+
| 96.94000244140625 | 4454.351539300434    |
| 95.93000030517578 | 4423.119941283189    |
| 98.37999725341797 | 4332.516490316291    |
| 97.97000122070312 | 4330.480601655014    |
| 98.52999877929688 | 4291.316953108634    |
+-------------------+----------------------+
```

Notice how wholesale cost values originally entered as decimal fractions such as 96.94 and 98.38 are slightly larger or smaller in the result set, due to precision limitations in the hardware floating-point types. The imprecise representation of FLOAT and DOUBLE values is why financial data processing systems often store currency using data types that are less space-efficient but avoid these types of rounding errors.

HAVING clause

Performs a filter operation on a SELECT query, by examining the results of aggregation functions rather than testing each individual table row. Therefore, 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.

Restrictions:

The filter expression in the HAVING clause cannot include a scalar subquery.

LIMIT clause

The LIMIT clause in a SELECT query sets a maximum number of rows for the result set. Pre-selecting the maximum size of the result set helps Impala to optimize memory usage while processing a distributed query.

Syntax:

```
LIMIT constant_integer_expression
```

The argument to the LIMIT clause must evaluate to a constant value. It can be a numeric literal, or another kind of numeric expression involving operators, casts, and function return values. You cannot refer to a column or use a subquery.

Usage notes:

This clause is useful in contexts such as:

- To return exactly N items from a top-N query, such as the 10 highest-rated items in a shopping category or the 50 hostnames that refer the most traffic to a web site.
- To demonstrate some sample values from a table or a particular query. (To display some arbitrary items, use a query with no ORDER BY clause. An ORDER BY clause causes additional memory and/or disk usage during the query.)
- To keep queries from returning huge result sets by accident if a table is larger than expected, or a WHERE clause matches more rows than expected.

Originally, the value for the LIMIT clause had to be a numeric literal. In Impala 1.2.1 and higher, it can be a numeric expression.

Prior to Impala 1.4.0, Impala required any query including an ORDER_BY clause to also use a LIMIT clause. In Impala 1.4.0 and higher, the LIMIT clause is optional for ORDER BY queries. In cases where sorting a huge result

set requires enough memory to exceed the Impala memory limit for a particular executor Impala daemon, Impala automatically uses a temporary disk work area to perform the sort operation.

In Impala 1.2.1 and higher, you can combine a LIMIT clause with an OFFSET clause to produce a small result set that is different from a top-N query, for example, to return items 11 through 20. This technique can be used to simulate “paged” results. Because Impala queries typically involve substantial amounts of I/O, use this technique only for compatibility in cases where you cannot rewrite the application logic. For best performance and scalability, wherever practical, query as many items as you expect to need, cache them on the application side, and display small groups of results to users using application logic.

Restrictions:

Correlated subqueries used in EXISTS and IN operators cannot include a LIMIT clause.

Examples:

The following example shows how the LIMIT clause caps the size of the result set, with the limit being applied after any other clauses such as WHERE.

```
[localhost:21000] > create database limits;
[localhost:21000] > use limits;
[localhost:21000] > create table numbers (x int);
[localhost:21000] > insert into numbers values (1), (3), (4), (5), (2);
Inserted 5 rows in 1.34s
[localhost:21000] > select x from numbers limit 100;
+---+
| x |
+---+
| 1 |
| 3 |
| 4 |
| 5 |
| 2 |
+---+
Returned 5 row(s) in 0.26s
[localhost:21000] > select x from numbers limit 3;
+---+
| x |
+---+
| 1 |
| 3 |
| 4 |
+---+
Returned 3 row(s) in 0.27s
[localhost:21000] > select x from numbers where x > 2 limit 2;
+---+
| x |
+---+
| 3 |
| 4 |
+---+
Returned 2 row(s) in 0.27s
```

For top-N and bottom-N queries, you use the ORDER BY and LIMIT clauses together:

```
[localhost:21000] > select x as "Top 3" from numbers order by x desc limit 3
;
+-------+
| top 3 |
+-------+
| 5     |
| 4     |
| 3     |
```

```
+-------+
[localhost:21000] > select x as "Bottom 3" from numbers order by x limit 3;
+----------+
| bottom 3 |
+----------+
| 1        |
| 2        |
| 3        |
+----------+
```

You can use constant values besides integer literals as the LIMIT argument:

```
-- Other expressions that yield constant integer values work too.
SELECT x FROM t1 LIMIT 1e6;                        -- Limit is one million.
SELECT x FROM t1 LIMIT length('hello world');      -- Limit is 11.
SELECT x FROM t1 LIMIT 2+2;                        -- Limit is 4.
SELECT x FROM t1 LIMIT cast(truncate(9.9) AS INT); -- Limit is 9.
```

OFFSET clause

The OFFSET clause in a SELECT query causes the result set to start some number of rows after the logical first item. The result set is numbered starting from zero, so OFFSET 0 produces the same result as leaving out the OFFSET clause. Always use this clause in combination with ORDER BY (so that it is clear which item should be first, second, and so on) and LIMIT (so that the result set covers a bounded range, such as items 0-9, 100-199, and so on).

In Impala 1.2.1 and higher, you can combine a LIMIT clause with an OFFSET clause to produce a small result set that is different from a top-N query, for example, to return items 11 through 20. This technique can be used to simulate “paged” results. Because Impala queries typically involve substantial amounts of I/O, use this technique only for compatibility in cases where you cannot rewrite the application logic. For best performance and scalability, wherever practical, query as many items as you expect to need, cache them on the application side, and display small groups of results to users using application logic.

Examples:

The following example shows how you could run a “paging” query originally written for a traditional database application. Because typical Impala queries process megabytes or gigabytes of data and read large data files from disk each time, it is inefficient to run a separate query to retrieve each small group of items. Use this technique only for compatibility while porting older applications, then rewrite the application code to use a single query with a large result set, and display pages of results from the cached result set.

```
[localhost:21000] > create table numbers (x int);
[localhost:21000] > insert into numbers select x from very_long_sequence;
Inserted 1000000 rows in 1.34s
[localhost:21000] > select x from numbers order by x limit 5 offset 0;
+----+
| x  |
+----+
| 1  |
| 2  |
| 3  |
| 4  |
| 5  |
+----+
[localhost:21000] > select x from numbers order by x limit 5 offset 5;
+----+
| x  |
+----+
| 6  |
| 7  |
| 8  |
| 9  |
| 10 |
```

```
+----+
```

UNION clause

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.

Syntax:

```
query_1 UNION [DISTINCT | ALL] query_2
```

Usage notes:

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.)

When an ORDER BY clause applies to a UNION ALL or UNION query, in Impala 1.4 and higher, the LIMIT clause is no longer required. To make the ORDER BY and LIMIT clauses apply to the entire result set, turn the UNION query into a subquery, SELECT from the subquery, and put the ORDER BY clause at the end, outside the subquery.

Examples:

First, set up some sample data, including duplicate 1 values:

```
[localhost:21000] > create table few_ints (x int);
[localhost:21000] > insert into few_ints values (1), (1), (2), (3);
[localhost:21000] > set default_order_by_limit=1000;
```

This example shows how UNION ALL returns all rows from both queries, without any additional filtering to eliminate duplicates. For the large result sets common with Impala queries, this is the most memory-efficient technique.

```
[localhost:21000] > select x from few_ints order by x;
+---+
| x |
+---+
| 1 |
| 1 |
| 2 |
| 3 |
+---+
Returned 4 row(s) in 0.41s
[localhost:21000] > select x from few_ints union all select x from few_ints;
+---+
| x |
+---+
| 1 |
| 1 |
| 2 |
| 3 |
| 1 |
| 1 |
| 2 |
| 3 |
+---+
Returned 8 row(s) in 0.42s
[localhost:21000] > select * from (select x from few_ints union all select x
 from few_ints) as t1 order by x;
+---+
| x |
+---+
| 1 |
```

```
| 1 |
| 1 |
| 1 |
| 2 |
| 2 |
| 3 |
| 3 |
+---+
Returned 8 row(s) in 0.53s
[localhost:21000] > select x from few_ints union all select 10;
+----+
| x  |
+----+
| 10 |
| 1  |
| 1  |
| 2  |
| 3  |
+----+
Returned 5 row(s) in 0.38s
```

This example shows how the UNION clause without the ALL keyword condenses the result set to eliminate all duplicate values, making the query take more time and potentially more memory. The extra processing typically makes this technique not recommended for queries that return result sets with millions or billions of values.

```
[localhost:21000] > select x from few_ints union select x+1 from few_ints;
+---+
| x |
+---+
| 3 |
| 4 |
| 1 |
| 2 |
+---+
Returned 4 row(s) in 0.51s
[localhost:21000] > select x from few_ints union select 10;
+----+
| x  |
+----+
| 2  |
| 10 |
| 1  |
| 3  |
+----+
Returned 4 row(s) in 0.49s
[localhost:21000] > select * from (select x from few_ints union select x fr
om few_ints) as t1 order by x;
+---+
| x |
+---+
| 1 |
| 2 |
| 3 |
+---+
Returned 3 row(s) in 0.53s
```

Subqueries in Impala SELECT statements

A *subquery* is a query that is nested within another query. Subqueries let queries on one table dynamically adapt based on the contents of another table. This technique provides great flexibility and expressive power for SQL queries.

A subquery can return a result set for use in the FROM or WITH clauses, or with operators such as IN or EXISTS.

A *scalar subquery* produces a result set with a single row containing a single column, typically produced by an aggregation function such as MAX() or SUM(). This single result value can be substituted in scalar contexts such as arguments to comparison operators. If the result set is empty, the value of the scalar subquery is NULL. For example, the following query finds the maximum value of T2.Y and then substitutes that value into the WHERE clause of the outer block that queries T1:

```
SELECT x FROM t1 WHERE x > (SELECT MAX(y) FROM t2);
```

Uncorrelated subqueries do not refer to any tables from the outer block of the query. The same value or set of values produced by the subquery is used when evaluating each row from the outer query block. In this example, the subquery returns an arbitrary number of values from T2.Y, and each value of T1.X is tested for membership in that same set of values:

```
SELECT x FROM t1 WHERE x IN (SELECT y FROM t2);
```

Correlated subqueries compare one or more values from the outer query block to values referenced in the WHERE clause of the subquery. Each row evaluated by the outer WHERE clause can be evaluated using a different set of values. These kinds of subqueries are restricted in the kinds of comparisons they can do between columns of the inner and outer tables. (See the following Restrictions item.)

For example, the following query finds all the employees with salaries that are higher than average for their department. The subquery potentially computes a different AVG() value for each employee.

```
SELECT employee_name, employee_id FROM employees one WHERE
  salary > (SELECT avg(salary) FROM employees two WHERE one.dept_id = two
.dept_id);
```

Syntax:

Subquery in the FROM clause:

```
SELECT select_list FROM table_ref [, table_ref ...]

table_ref ::= table_name | (select_statement)
```

Subqueries in WHERE clause:

```
WHERE value comparison_operator (scalar_select_statement)
WHERE value [NOT] IN (select_statement)
WHERE [NOT] EXISTS (correlated_select_statement)
WHERE NOT EXISTS (correlated_select_statement)
```

comparison_operator is a numeric comparison such as =, <=, !=, and so on, or a string comparison operator such as LIKE or REGEXP.

Although you can use non-equality comparison operators such as < or >=, the subquery must include at least one equality comparison between the columns of the inner and outer query blocks.

All syntax is available for both correlated and uncorrelated queries, except that the NOT EXISTS clause cannot be used with an uncorrelated subquery.

Impala subqueries can be nested arbitrarily deep.

Standards compliance: Introduced in SQL:1999.

Examples:

This example illustrates how subqueries can be used in the FROM clause to organize the table names, column names, and column values by producing intermediate result sets, especially for join queries.

```
SELECT avg(t1.x), max(t2.y) FROM
```

```
  (SELECT id, cast(a AS DECIMAL(10,5)) AS x FROM raw_data WHERE a BETWEEN 0
 AND 100) AS t1
  JOIN
  (SELECT id, length(s) AS y FROM raw_data WHERE s LIKE 'A%') AS t2;
  USING (id);
```

These examples show how a query can test for the existence of values in a separate table using the EXISTS() operator with a subquery.

The following examples show how a value can be compared against a set of values returned by a subquery.

```
SELECT count(x) FROM t1 WHERE EXISTS(SELECT 1 FROM t2 WHERE t1.x = t2.y * 10
);

SELECT x FROM t1 WHERE x IN (SELECT y FROM t2 WHERE state = 'CA');
```

The following examples demonstrate scalar subqueries. When a subquery is known to return a single value, you can substitute it where you would normally put a constant value.

```
SELECT x FROM t1 WHERE y = (SELECT max(z) FROM t2);
SELECT x FROM t1 WHERE y > (SELECT count(z) FROM t2);
```

Usage notes:

If the same table is referenced in both the outer and inner query blocks, construct a table alias in the outer query block and use a fully qualified name to distinguish the inner and outer table references:

```
SELECT * FROM t1 one WHERE id IN (SELECT parent FROM t1 two WHERE t1.parent
= t2.id);
```

The STRAIGHT_JOIN hint affects the join order of table references in the query block containing the hint. It does not affect the join order of nested queries, such as views, inline views, or WHERE-clause subqueries. To use this hint for performance tuning of complex queries, apply the hint to all query blocks that need a fixed join order.

Internal details:

Internally, subqueries involving IN, NOT IN, EXISTS, or NOT EXISTS clauses are rewritten into join queries. Depending on the syntax, the subquery might be rewritten to an outer join, semi join, cross join, or anti join.

A query is processed differently depending on whether the subquery calls any aggregation functions. There are correlated and uncorrelated forms, with and without calls to aggregation functions. Each of these four categories is rewritten differently.

Column statistics considerations:

Because queries that include correlated and uncorrelated subqueries in the WHERE clause are written into join queries, to achieve best performance, follow the same guidelines for running the COMPUTE STATS statement as you do for tables involved in regular join queries. Run the COMPUTE STATS statement for each associated tables after loading or substantially changing the data in that table.

Restrictions:

The initial Impala support for nested subqueries addresses the most common use cases. Some restrictions remain:

- Although you can use subqueries in a query involving UNION or UNION ALL in Impala 2.1.0 and higher, currently you cannot construct a union of two subqueries (for example, in the argument of an IN or EXISTS operator).
- Subqueries returning scalar values cannot be used with the operators ANY or ALL. (Impala does not currently have a SOME operator, but if it did, the same restriction would apply.)
- For the EXISTS and NOT EXISTS clauses, any subquery comparing values from the outer query block to another table must use at least one equality comparison, not exclusively other kinds of comparisons such as less than, greater than, BETWEEN, or !=.

- Currently, a scalar subquery cannot be used as the first or second argument to the BETWEEN operator.
- A subquery cannot be used inside an OR conjunction. Expressions inside a subquery, for example in the WHERE clause, can use OR conjunctions; the restriction only applies to parts of the query “above” the subquery.
- Scalar subqueries are only supported in numeric contexts. You cannot use a scalar subquery as an argument to the LIKE, REGEXP, or RLIKE operators, or compare it to a value of a non-numeric type such as TIMESTAMP or BOOLEAN.
- You cannot use subqueries with the CASE function to generate the comparison value, the values to be compared against, or the return value.
- A subquery is not allowed in the filter condition for the HAVING clause. (Strictly speaking, a subquery cannot appear anywhere outside the WITH, FROM, and WHERE clauses.)
- You must use a fully qualified name (*table_name.column_name* or *database_name.table_name.column_name*) when referring to any column from the outer query block within a subquery.
- The TABLESAMPLE clause of the SELECT statement does not apply to a table reference derived from a view, a subquery, or anything other than a real base table. This clause only works for tables backed by HDFS or HDFS-like data files, therefore it does not apply to Kudu or HBase tables.

Complex type considerations:

For the complex types (ARRAY, STRUCT, and MAP) available in Impala 2.3 and higher, the join queries that “unpack” complex type columns often use correlated subqueries in the FROM clause. For example, if the first table in the join clause is CUSTOMER, the second join clause might have a subquery that selects from the column CUST OMER.C_ORDERS, which is an ARRAY. The subquery re-evaluates the ARRAY elements corresponding to each row from the CUSTOMER table.

Related Information

Using subqueries with complex types

Table and column statistics

TABLESAMPLE clause

Specify the TABLESAMPLE clause when you need to explore the data distribution within the table, the table is very large, and it is impractical or unnecessary to process all the data from the table or selected partitions.

The clause makes the query process a randomized set of data files from the table, so that the total volume of data is greater than or equal to the specified percentage of data bytes within that table. (Or the data bytes within the set of partitions that remain after partition pruning is performed.)

Syntax:

```
TABLESAMPLE SYSTEM(percentage) [REPEATABLE(seed)]
```

The TABLESAMPLE clause comes immediately after a table name or table alias.

The SYSTEM keyword represents the sampling method. Currently, Impala only supports a single sampling method named SYSTEM.

The *percentage* argument is an integer literal from 0 to 100. A percentage of 0 produces an empty result set for a particular table reference, while a percentage of 100 uses the entire contents. Because the sampling works by selecting a random set of data files, the proportion of sampled data from the table may be greater than the specified percentage, based on the number and sizes of the underlying data files. See the usage notes for details.

The optional REPEATABLE keyword lets you specify an arbitrary positive integer seed value that ensures that when the query is run again, the sampling selects the same set of data files each time. REPEATABLE does not have a default value. If you omit the REPEATABLE keyword, the random seed is derived from the current time.

See the COMPUTE STATS statement for the TABLESAMPLE clause used in the COMPUTE STATS statement.

Usage notes:

You might use this clause with aggregation queries, such as finding the approximate average, minimum, or maximum where exact precision is not required. You can use these findings to plan the most effective strategy for constructing queries against the full table or designing a partitioning strategy for the data.

Some other database systems have a TABLESAMPLE clause. The Impala syntax for this clause is modeled on the syntax for popular relational databases, not the Hive TABLESAMPLE clause. For example, there is no BUCKETS keyword as in Hive SQL.

The precision of the *percentage* threshold depends on the number and sizes of the underlying data files. Impala brings in additional data files, one at a time, until the number of bytes exceeds the specified percentage based on the total number of bytes for the entire set of table data. The precision of the percentage threshold is higher when the table contains many data files with consistent sizes. See the code listings later in this section for examples.

When you estimate characteristics of the data distribution based on sampling a percentage of the table data, be aware that the data might be unevenly distributed between different files. Do not assume that the percentage figure reflects the percentage of rows in the table. For example, one file might contain all blank values for a STRING column, while another file contains long strings in that column; therefore, one file could contain many more rows than another. Likewise, a table created with the SORT BY clause might contain narrow ranges of values for the sort columns, making it impractical to extrapolate the number of distinct values for those columns based on sampling only some of the data files.

Because a sample of the table data might not contain all values for a particular column, if the TABLESAMPLE is used in a join query, the key relationships between the tables might produce incomplete result sets compared to joins using all the table data. For example, if you join 50% of table A with 50% of table B, some values in the join columns might not match between the two tables, even though overall there is a 1:1 relationship between the tables.

The REPEATABLE keyword makes identical queries use a consistent set of data files when the query is repeated. You specify an arbitrary integer key that acts as a seed value when Impala randomly selects the set of data files to use in the query. This technique lets you verify correctness, examine performance, and so on for queries using the TABL ESAMPLE clause without the sampled data being different each time. The repeatable aspect is reset (that is, the set of selected data files may change) any time the contents of the table change. The statements or operations that can make sampling results non-repeatable are:

- INSERT.
- TRUNCATE TABLE.
- LOAD DATA.
- REFRESH or INVALIDATE METADATA after files are added or removed by a non-Impala mechanism.

This clause is similar in some ways to the LIMIT clause, because both serve to limit the size of the intermediate data and final result set. LIMIT 0 is more efficient than TABLESAMPLE SYSTEM(0) for verifying that a query can execute without producing any results. TABLESAMPLE SYSTEM(*n*) often makes query processing more efficient than using a LIMIT clause by itself, because all phases of query execution use less data overall. If the intent is to retrieve some representative values from the table in an efficient way, you might combine TABLESAMPLE, ORDE R BY, and LIMIT clauses within a single query.

Partitioning:

When you query a partitioned table, any partition pruning happens before Impala selects the data files to sample. For example, in a table partitioned by year, a query with WHERE year = 2017 and a TABLESAMPLE SYSTEM(10) clause would sample data files representing at least 10% of the bytes present in the 2017 partition.

Amazon S3 considerations:

This clause applies to S3 tables the same way as tables with data files stored on HDFS.

ADLS considerations:

This clause applies to ADLS tables the same way as tables with data files stored on HDFS.

Kudu considerations:

This clause does not apply to Kudu tables.

HBase considerations:

This clause does not apply to HBase tables.

Performance considerations:

From a performance perspective, the TABLESAMPLE clause is especially valuable for exploratory queries on text, Avro, or other file formats other than Parquet. Text-based or row-oriented file formats must process substantial amounts of redundant data for queries that derive aggregate results such as MAX(), MIN(), or AVG() for a single column. Therefore, you might use TABLESAMPLE early in the ETL pipeline, when data is still in raw text format and has not been converted to Parquet or moved into a partitioned table.

Restrictions:

This clause applies only to tables that use a storage layer with underlying raw data files, such as HDFS, Amazon S3, or Microsoft ADLS.

This clause does not apply to table references that represent views. A query that applies the TABLESAMPLE clause to a view or a subquery fails with a semantic error.

Because the sampling works at the level of entire data files, it is by nature coarse-grained. It is possible to specify a small sample percentage but still process a substantial portion of the table data if the table contains relatively few data files, if each data file is very large, or if the data files vary substantially in size. Be sure that you understand the data distribution and physical file layout so that you can verify if the results are suitable for extrapolation. For example, if the table contains only a single data file, the “sample” will consist of all the table data regardless of the percentage you specify. If the table contains data files of 1 GiB, 1 GiB, and 1 KiB, when you specify a sampling percentage of 50 you would either process slightly more than 50% of the table (1 GiB + 1 KiB) or almost the entire table (1 GiB + 1 GiB), depending on which data files were selected for sampling.

If data files are added by a non-Impala mechanism, and the table metadata is not updated by a REFRESH or INVA LIDATE METADATA statement, the TABLESAMPLE clause does not consider those new files when computing the number of bytes in the table or selecting which files to sample.

If data files are removed by a non-Impala mechanism, and the table metadata is not updated by a REFRESH or INVA LIDATE METADATA statement, the query fails if the TABLESAMPLE clause attempts to reference any of the missing files.

Examples:

The following examples demonstrate the TABLESAMPLE clause. These examples intentionally use very small data sets to illustrate how the number of files, size of each file, and overall size of data in the table interact with the percentage specified in the clause.

These examples use an unpartitioned table, containing several files of roughly the same size:

```
create table sample_demo (x int, s string);

insert into sample_demo values (1, 'one');
insert into sample_demo values (2, 'two');
insert into sample_demo values (3, 'three');
insert into sample_demo values (4, 'four');
insert into sample_demo values (5, 'five');

show files in sample_demo;
+---------------------+------+-----------+
| Path                | Size | Partition |
+---------------------+------+-----------+
| 991213608_data.0.   | 7B   |           |
| 982196806_data.0.   | 6B   |           |
| _2122096884_data.0. | 8B   |           |
| _586325431_data.0.  | 6B   |           |
| 1894746258_data.0.  | 7B   |           |
+---------------------+------+-----------+
show table stats sample_demo;
+-------+--------+------+--------+-------------------------+
```

```
| #Rows | #Files | Size | Format | Location                |
+-------+--------+------+--------+-------------------------+
| -1    | 5      | 34B  | TEXT   | /tsample.db/sample_demo |
+-------+--------+------+--------+-------------------------+
```

A query that samples 50% of the table must process at least 17 bytes of data. Based on the sizes of the data files, we can predict that each such query uses 3 arbitrary files. Any 1 or 2 files are not enough to reach 50% of the total data in the table (34 bytes), so the query adds more files until it passes the 50% threshold:

```
select distinct x from sample_demo tablesample system(50);
+---+
| x |
+---+
| 4 |
| 1 |
| 5 |
+---+

select distinct x from sample_demo tablesample system(50);
+---+
| x |
+---+
| 5 |
| 4 |
| 2 |
+---+

select distinct x from sample_demo tablesample system(50);
+---+
| x |
+---+
| 5 |
| 3 |
| 2 |
+---+
```

To help run reproducible experiments, the REPEATABLE clause causes Impala to choose the same set of files for each query. Although the data set being considered is deterministic, the order of results varies (in the absence of an ORDER BY clause) because of the way distributed queries are processed:

```
select distinct x from sample_demo
  tablesample system(50) repeatable (12345);
+---+
| x |
+---+
| 3 |
| 2 |
| 1 |
+---+

select distinct x from sample_demo
  tablesample system(50) repeatable (12345);
+---+
| x |
+---+
| 2 |
| 1 |
| 3 |
+---+
```

The following examples show how uneven data distribution affects which data is sampled. Adding another data file containing a long string value changes the threshold for 50% of the total data in the table:

```
insert into sample_demo values
(1000, 'Boyhood is the longest time in life for a boy. The last term of the
 school-year is made of decades, not of weeks, and living through them is
like waiting for the millennium. Booth Tarkington');

show files in sample_demo;
+---------------------+------+-----------+
| Path                | Size | Partition |
+---------------------+------+-----------+
| 991213608_data.0.   | 7B   |           |
| 982196806_data.0.   | 6B   |           |
| _253317650_data.0.  | 196B |           |
| _2122096884_data.0. | 8B   |           |
| _586325431_data.0.  | 6B   |           |
| 1894746258_data.0.  | 7B   |           |
+---------------------+------+-----------+

show table stats sample_demo;
+-------+--------+------+--------+-------------------------+
| #Rows | #Files | Size | Format | Location                |
+-------+--------+------+--------+-------------------------+
| -1    | 6      | 230B | TEXT   | /tsample.db/sample_demo |
+-------+--------+------+--------+-------------------------+
```

Even though the queries do not refer to the S column containing the long value, all the sampling queries include the data file containing the column value X=1000, because the query cannot reach the 50% threshold (115 bytes) without including that file. The large file might be considered first, in which case it is the only file processed by the query. Or an arbitrary set of other files might be considered first.

```
select distinct x from sample_demo tablesample system(50);
+------+
| x    |
+------+
| 1000 |
| 3    |
| 1    |
+------+

select distinct x from sample_demo tablesample system(50);
+------+
| x    |
+------+
| 1000 |
+------+

select distinct x from sample_demo tablesample system(50);
+------+
| x    |
+------+
| 1000 |
| 4    |
| 2    |
| 1    |
+------+
```

The following examples demonstrate how the TABLESAMPLE clause interacts with other table aspects, such as partitioning and file format:

```
create table sample_demo_partitions (x int, s string) partitioned by (n int)
 stored as parquet;
insert into sample_demo_partitions partition (n = 1) select * from sample_d
emo;
insert into sample_demo_partitions partition (n = 2) select * from sample_d
emo;
insert into sample_demo_partitions partition (n = 3) select * from sample_d
emo;

show files in sample_demo_partitions;
+--------------------------------+--------+-----------+
| Path                           | Size   | Partition |
+--------------------------------+--------+-----------+
| 000000_364262785_data.0.parq   | 1.24KB | n=1       |
| 000001_973526736_data.0.parq   | 566B   | n=1       |
| 0000000_1300598134_data.0.parq | 1.24KB | n=2       |
| 0000001_689099063_data.0.parq  | 568B   | n=2       |
| 0000000_1861371709_data.0.parq | 1.24KB | n=3       |
| 0000001_1065507912_data.0.parq | 566B   | n=3       |
+--------------------------------+--------+-----------+
show table stats tablesample_demo_partitioned;
+-------+-------+--------+--------+---------+------------------------------
---------------+
| n     | #Rows | #Files | Size   | Format  | Location
               |
+-------+-------+--------+--------+---------+----------------------------
-----------------+
| 1     | -1    | 2      | 1.79KB | PARQUET | /tsample.db/tablesample_dem
o_partitioned/n=1 |
| 2     | -1    | 2      | 1.80KB | PARQUET | /tsample.db/tablesample_demo_p
artitioned/n=2 |
| 3     | -1    | 2      | 1.79KB | PARQUET | /tsample.db/tablesample_demo_
partitioned/n=3 |
| Total | -1    | 6      | 5.39KB |         |
               |
+-------+-------+--------+--------+---------+----------------------------
------------------+
```

If the query does not involve any partition pruning, the sampling applies to the data volume of the entire table:

```
-- 18 rows total.
select count(*) from sample_demo_partitions;
+----------+
| count(*) |
+----------+
| 18       |
+----------+
-- The number of rows per data file is not
-- perfectly balanced, therefore the count
-- is different depending on which set of files
-- is considered.
select count(*) from sample_demo_partitions
  tablesample system(75);
+----------+
| count(*) |
+----------+
| 14       |
+----------+
select count(*) from sample_demo_partitions
  tablesample system(75);
```

```
+----------+
| count(*) |
+----------+
| 16       |
+----------+
```

If the query only processes certain partitions, the query computes the sampling threshold based on the data size and set of files only from the relevant partitions:

```
select count(*) from sample_demo_partitions
  tablesample system(50) where n = 1;
+----------+
| count(*) |
+----------+
| 6        |
+----------+

select count(*) from sample_demo_partitions
  tablesample system(50) where n = 1;
+----------+
| count(*) |
+----------+
| 2        |
+----------+
```

Related Information

COMPUTE STATS statement

WITH clause

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”.

You can rewrite a query using subqueries to work the same as with the WITH clause. The purposes of the WITH clause are:

- Convenience and ease of maintenance from less repetition with the body of the query. Typically used with queries involving UNION, joins, or aggregation functions where the similar complicated expressions are referenced multiple times.
- SQL code that is easier to read and understand by abstracting the most complex part of the query into a separate block.
- Improved compatibility with SQL from other database systems that support the same clause (primarily Oracle Database).

Note:

The Impala WITH clause does not support recursive queries in the WITH, which is supported in some other database systems.

Standards compliance: Introduced in SQL:1999.

Examples:

```
-- Define 2 subqueries that can be referenced from the body of a longer quer
y.
with t1 as (select 1), t2 as (select 2) insert into tab select * from t1
union all select * from t2;
-- Define one subquery at the outer level, and another at the inner level as
 part of the
-- initial stage of the UNION ALL query.
```

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

DISTINCT operator

The DISTINCT operator in a SELECT statement filters the result set to remove duplicates.

```
-- Returns the unique values from one column.
-- NULL is included in the set of values if any rows have a NULL in this c
olumn.
SELECT DISTINCT c_birth_country FROM customer;

-- Returns the unique combinations of values from multiple columns.
SELECT DISTINCT c_salutation, c_last_name FROM customer;
```

You can use DISTINCT in combination with an aggregation function, typically COUNT(), to find how many different values a column contains.

```
-- Counts the unique values from one column.
-- NULL is not included as a distinct value in the count.
SELECT COUNT(DISTINCT c_birth_country) FROM customer;

-- Counts the unique combinations of values from multiple columns.
SELECT COUNT(DISTINCT c_salutation, c_last_name) FROM customer;
```

Zero-length strings: For purposes of clauses such as DISTINCT and GROUP BY, Impala considers zero-length strings (""), NULL, and space to all be different values.

Note:

In contrast with some database systems that always return DISTINCT values in sorted order, Impala does not do any ordering of DISTINCT values. Always include an ORDER BY clause if you need the values in alphabetical or numeric sorted order.

SET statement

The SET statement specifies values for query options that control the runtime behavior of other statements within the same session.

When issued in impala-shell, the SET command is interpreted as an impala-shell command that has differences from the SQL SET statement.

Syntax:

```
SET
SET ALL
SET query_option=option_value
SET query_option=""
```

SET and SET ALL with no arguments return a result set consisting of all the applicable query options and their current values.

The *query_option* and *option_value* are case-insensitive.

Unlike the impala-shell command version of SET, when used as a SQL statement, the string values for *option_value* need to be quoted, e.g. SET option="new_value".

The SET *query_option* = "" statement unsets the value of the *query_option* in the current session, reverting it to the default state. In impala-shell, use the UNSET command to set a query option to its default.

Each query option has a specific allowed notation for its arguments. See *Query options for the SET statement* for the details of each query option.

Usage notes:

The options are divided into groups: Regular Query Options, Advanced Query Options, Development Query Options, and Deprecated Query Options.

- The advanced options are intended for use in specific kinds of performance tuning and debugging scenarios.
- The development options are related to internal development of Impala or features that are not yet finalized. These options might be changed or removed without notice.
- The deprecated options are related to features that are removed or changed so that the options no longer have any purpose. These options might be removed in future versions.
- By default, only the first two groups, regular and advanced, are displayed by the SET command. Use SET ALL to see all groups of options.
- `impala-shell` options and user-specified variables are always displayed at the end of the list of query options, after all appropriate option groups.

SET has always been available as an `impala-shell` command. Promoting it to a SQL statement lets you use this feature in client applications through the JDBC and ODBC APIs.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Cloudera Manager: You can set any of the query options globally in Cloudera Manager to affect all the queries in the cluster.

1. Navigate to *Impala cluster* > Configuration > Impala Daemon Query Options Advanced Configuration Snippet (Safety Valve)
2. In the field, type a key-value pair of a query option and the value, e.g. EXPLAIN_LEVEL=2.
3. Click + to enter an additional option.
4. Clive Save Changes.

Related Information

Query options

SET command in impala-shell

SHOW statement

Use the SHOW statement to get information about different types of Impala objects.

Syntax:

```
SHOW DATABASES [[LIKE] 'pattern']
SHOW SCHEMAS [[LIKE] 'pattern'] - an alias for SHOW DATABASES
SHOW TABLES [IN database_name] [[LIKE] 'pattern']
SHOW [AGGREGATE | ANALYTIC] FUNCTIONS [IN database_name] [[LIKE] 'pattern']
SHOW CREATE TABLE [database_name].table_name
SHOW CREATE VIEW [database_name].view_name
SHOW TABLE STATS [database_name.]table_name
SHOW COLUMN STATS [database_name.]table_name
SHOW [RANGE] PARTITIONS [database_name.]table_name
SHOW FILES IN [database_name.]table_name [PARTITION (key_col_expression
 [, key_col_expression]]

SHOW GRANT USER user_name
SHOW GRANT USER user_name ON SERVER
SHOW GRANT USER user_name ON DATABASE database_name
SHOW GRANT USER user_name ON TABLE table_name
SHOW GRANT USER user_name ON URI uri
```

Issue a SHOW *object_type* statement to see the appropriate objects in the current database, or SHOW *object_type* IN *database_name* to see objects in a specific database.

The optional *pattern* argument is a quoted string literal, using Unix-style * wildcards and allowing | for alternation. The preceding LIKE keyword is also optional. All object names are stored in lowercase, so use all lowercase letters in the pattern string. For example:

```
show databases 'a*';
show databases like 'a*';
show tables in some_db like '*fact*';
use some_db;
show tables '*dim*|*fact*';
```

Cancellation: Cannot be cancelled.

SHOW FILES statement

The SHOW FILES statement displays the files that constitute a specified table, or a partition within a partitioned table. This syntax is available in Impala 2.2 and higher only. The output includes the names of the files, the size of each file, and the applicable partition for a partitioned table. The size includes a suffix of B for bytes, MB for megabytes, and GB for gigabytes.

In Impala 2.8 and higher, you can use general expressions with operators such as <, IN, LIKE, and BETWEEN in the PARTITION clause, instead of only equality operators. For example:

```
show files in sample_table partition (j < 5);
show files in sample_table partition (k = 3, l between 1 and 10);
show files in sample_table partition (month like 'J%');
```

Note: This statement applies to tables and partitions stored on HDFS, or in the Amazon Simple Storage System (S3). It does not apply to views. It does not apply to tables mapped onto HBase or Kudu, because those data management systems do not use the same file-based storage layout.

Usage notes:

You can use this statement to verify the results of your ETL process: that is, that the expected files are present, with the expected sizes. You can examine the file information to detect conditions such as empty files, missing files, or inefficient layouts due to a large number of small files. When you use INSERT statements to copy from one table to another, you can see how the file layout changes due to file format conversions, compaction of small input files into large data blocks, and multiple output files from parallel queries and partitioned inserts.

The output from this statement does not include files that Impala considers to be hidden or invisible, such as those whose names start with a dot or an underscore, or that end with the suffixes .copying or .tmp.

The information for partitioned tables complements the output of the SHOW PARTITIONS statement, which summarizes information about each partition. SHOW PARTITIONS produces some output for each partition, while SHOW FILES does not produce any output for empty partitions because they do not include any data files.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read permission for all the table files, read and execute permission for all the directories that make up the table, and execute permission for the database directory and all its parent directories.

SHOW GRANT USER statement

The SHOW GRANT USER statement shows the list of privileges for a given user. This statement is only allowed for administrative users. However, the current user can run SHOW GRANT USER for themselves.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

SHOW DATABASES

The SHOW DATABASES statement is often the first one you issue when connecting to an instance for the first time. You typically issue SHOW DATABASES to see the names you can specify in a USE *db_name* statement, then after switching to a database you issue SHOW TABLES to see the names you can specify in SELECT and INSERT statements.

In Impala 2.5 and higher, the output includes a second column showing any associated comment for each database.

The output of SHOW DATABASES includes the special _impala_builtins database, which lets you view definitions of built-in functions, as described under SHOW FUNCTIONS.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

SHOW TABLES statement

Displays the names of tables. By default, lists tables in the current database, or with the IN clause, in a specified database. By default, lists all tables, or with the LIKE clause, only those whose name match a pattern with * wildcards.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

The user ID that the `impalad` daemon runs under, typically the impala user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its LOCATION attribute.)

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

SHOW CREATE TABLE statement

As a schema changes over time, you might run a CREATE TABLE statement followed by several ALTER TABLE statements. To capture the cumulative effect of all those statements, SHOW CREATE TABLE displays a CREATE TABLE statement that would reproduce the current structure of a table. You can use this output in scripts that set up or clone a group of tables, rather than trying to reproduce the original sequence of CREATE TABLE and ALTER TABLE statements. When creating variations on the original table, or cloning the original table on a different system, you might need to edit the SHOW CREATE TABLE output to change things such as the database name, LOCATION field, and so on that might be different on the destination system.

If you specify a view name in the SHOW CREATE TABLE, it returns a CREATE VIEW statement with column names and the original SQL statement to reproduce the view. You need the VIEW_METADATA privilege on the view and SELECT privilege on all underlying views and tables to successfully run the SHOW CREATE VIEW statement for a view. The SHOW CREATE VIEW is available as an alias for SHOW CREATE TABLE.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Kudu considerations:

For Kudu tables:

- The column specifications include attributes such as NULL, NOT NULL, ENCODING, and COMPRESSION. If you do not specify those attributes in the original CREATE TABLE statement, the SHOW CREATE TABLE output displays the defaults that were used.
- The specifications of any RANGE clauses are not displayed in full. To see the definition of the range clauses for a Kudu table, use the SHOW RANGE PARTITIONS statement.
- The TBLPROPERTIES output reflects the Kudu master address and the internal Kudu name associated with the Impala table.

SHOW CREATE VIEW statement

The SHOW CREATE VIEW, it returns a CREATE VIEW statement with column names and the original SQL statement to reproduce the view. You need the VIEW_METADATA privilege on the view and SELECT privilege on all underlying views and tables to successfully run the SHOW CREATE VIEW statement for a view.

The SHOW CREATE VIEW is an alias for SHOW CREATE TABLE.

SHOW TABLE STATS statement

The SHOW TABLE STATS and SHOW COLUMN STATS variants are important for tuning performance and diagnosing performance issues, especially with the largest tables and the most complex join queries.

Any values that are not available (because the COMPUTE STATS statement has not been run yet) are displayed as -1.

SHOW TABLE STATS provides some general information about the table, such as the number of files, overall size of the data, whether some or all of the data is in the HDFS cache, and the file format, that is useful whether or not you have run the COMPUTE STATS statement. A -1 in the #Rows output column indicates that the COMPUTE ST ATS statement has never been run for this table. If the table is partitioned, SHOW TABLE STATS provides this information for each partition. (It produces the same output as the SHOW PARTITIONS statement in this case.)

The output of SHOW COLUMN STATS is primarily only useful after the COMPUTE STATS statement has been run on the table. A -1 in the #Distinct Values output column indicates that the COMPUTE STATS statement has never been run for this table. Currently, Impala always leaves the #Nulls column as -1, even after COMPUTE STATS has been run.

These SHOW statements work on actual tables only, not on views.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

Kudu considerations:

Because Kudu tables do not have characteristics derived from HDFS, such as number of files, file format, and HDFS cache status, the output of SHOW TABLE STATS reflects different characteristics that apply to Kudu tables. If the Kudu table is created with the clause PARTITIONS 20, then the result set of SHOW TABLE STATS consists of 20 rows, each representing one of the numbered partitions.

Impala does not compute the number of rows for each partition for Kudu tables. Therefore, you do not need to re-run COMPUTE STATS when you see -1 in the # Rows column of the output from SHOW TABLE STATS. That column always shows -1 for all Kudu tables.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its LOCATION attribute.) The Impala user must also have execute permission for the database directory, and any parent directories of the database directory in HDFS.

SHOW COLUMN STATS statement

The SHOW TABLE STATS and SHOW COLUMN STATS variants are important for tuning performance and diagnosing performance issues, especially with the largest tables and the most complex join queries.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

Kudu considerations:

The output for SHOW COLUMN STATS includes the relevant information for Kudu tables. The information for column statistics that originates in the underlying Kudu storage layer is also represented in the metastore database that Impala uses.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its LOCATION attribute.) The Impala user must also have execute permission for the database directory, and any parent directories of the database directory in HDFS.

SHOW PARTITIONS statement

SHOW PARTITIONS displays information about each partition for a partitioned table. (The output is the same as the SHOW TABLE STATS statement, but SHOW PARTITIONS only works on a partitioned table.) Because it displays table statistics for all partitions, the output is more informative if you have run the COMPUTE STATS statement after creating all the partitions. For example, on a CENSUS table partitioned on the YEAR column:

Because Kudu tables are all considered to be partitioned, the SHOW PARTITIONS statement works for any Kudu table. The default output is the same as for SHOW TABLE STATS, with the same Kudu-specific columns in the result set.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

Kudu considerations:

The optional RANGE clause only applies to Kudu tables. It displays only the partitions defined by the RANGE clause of CREATE TABLE or ALTER TABLE.

Although you can specify < or <= comparison operators when defining range partitions for Kudu tables, Kudu rewrites them if necessary to represent each range as *low_bound* <= VALUES < *high_bound*. This rewriting might involve incrementing one of the boundary values or appending a \0 for string values, so that the partition covers the same range as originally specified.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have read and execute permissions for all directories that are part of the table. (A table could span multiple different HDFS directories if it is partitioned. The directories could be widely scattered because a partition can reside in an arbitrary HDFS directory based on its LOCATION attribute.) The Impala user must also have execute permission for the database directory, and any parent directories of the database directory in HDFS.

SHOW FUNCTIONS statement

By default, SHOW FUNCTIONS displays user-defined functions (UDFs) and SHOW AGGREGATE FUNCTIONS displays user-defined aggregate functions (UDAFs) associated with a particular database. The output from SHOW FUNCTIONS includes the argument signature of each function. You specify this argument signature as part of the DROP FUNCTION statement. You might have several UDFs with the same name, each accepting different argument data types.

Usage notes:

In Impala 2.5 and higher, the SHOW FUNCTIONS output includes a new column, labelled is persistent. This property is true for Impala built-in functions, C++ UDFs, and Java UDFs created using the new CREATE FUNCTION syntax with no signature. It is false for Java UDFs created using the old CREATE FUNCTION syntax that includes the types for the arguments and return value. Any functions with false shown for this property must be created again by the CREATE FUNCTION statement each time the Impala catalog server is restarted. See CREATE FUNCTION for information on switching to the new syntax, so that Java UDFs are preserved across restarts. Java UDFs that are persisted this way are also easier to share across Impala and Hive.

Security considerations:

When authorization is enabled, the output of the SHOW statement only shows those objects for which you have the privilege to view. If you believe an object exists but you cannot see it in the SHOW output, check with the system administrator if you need to be granted a new privilege for that object.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

SHUTDOWN statement

The SHUTDOWN statement performs a graceful shutdown of Impala Daemon. The Impala daemon will notify other Impala daemons that it is shutting down, wait for a grace period, then shut itself down once no more queries or fragments are executing on that daemon. The --shutdown_grace_period_s flag determines the duration of the grace period in seconds.

Syntax:

```
:SHUTDOWN()
:SHUTDOWN([host_name[:port_number] )
:SHUTDOWN(deadline)
:SHUTDOWN([host_name[:port_number], deadline)
```

Usage notes:

All arguments are optional for SHUTDOWN.

Argument	Type	Default	Description
host_name	STRING	The current impalad host to whom the SHUTDOWN statement is submitted.	Address of the impalad to be shut down.

Argument	Type	Default	Description
port_number	INT	• In Impala 3.1, the current impalad's port used for the thrift based communication with other impalads (by default, 22000). • In Impala 3.2 and higher, the current impalad's port used for the KRPC based communication with other impalads (by default, 27000).	Specifies the port by which the impalad can be contacted. • In Impala 3.1, use the same impalad port used for the thrift based inter-Impala communication. • In Impala 3.2 and higher, use the same impalad port used for the KRPC based inter-Impala communication.
deadline	INT	The value of the ##shutdown_deadline_s flag, which defaults to 1 hour.	*deadline* must be a non-negative number, specified in seconds. The value, 0, for *deadline* specifies an immediate shutdown.

Take the following points into consideration when running the SHUTDOWN statement:

- A client can shut down the coordinator impalad that it is connected to via :SHUTDOWN().
- A client can remotely shut down any impalad via :SHUTDOWN('*hostname*').
- The shutdown time limit can be overridden to force a quicker or slower shutdown by specifying a deadline. The default deadline is determined by the --shutdown_deadline_s flag, which defaults to 1 hour.
- The executors can be shut down without disrupting running queries. Short-running queries will finish, and long-running queries will continue until the *deadline* is reached.
- If queries are submitted to a coordinator after shutdown of that coordinator has started, they will fail.
- Long running queries or other issues, such as stuck fragments, will slow down but not prevent eventual shutdown.

Security considerations:

The ALL privilege is required on the server.

Cancellation: Cannot be cancelled.

Examples:

```
:SHUTDOWN(); -- Shut down the current impalad  with the default deadline.
:SHUTDOWN('hostname'); --  Shut down impalad running on hostname  with the
 default deadline.
:SHUTDOWN(\"hostname:1234\"); -- Shut down impalad running on host at port
 1234  with the default deadline.
:SHUTDOWN(10); - Shut down the current impalad after 10 seconds.
:SHUTDOWN('hostname', 10); - Shut down impalad running on hostname when all
queries running on hostname finish, or after 10 seconds.
:SHUTDOWN('hostname:11', 10 * 60); -- Shut down impalad running on hostname
 at port 11 when all queries running on hostname finish, or after 600 second
s.
:SHUTDOWN(0); -- Perform an immdediate shutdown of the current impalad.
```

Added in: Impala 3.1

TRUNCATE TABLE statement

The TRUNCATE TABLE statement removes the data from an Impala table while leaving the table itself.

Syntax:

```
TRUNCATE [TABLE] [IF EXISTS] [db_name.]table_name
```

Statement type: DDL

Usage notes:

Often used to empty tables that are used during ETL cycles, after the data has been copied to another table for the next stage of processing. This statement is a low-overhead alternative to dropping and recreating the table, or using INSERT OVERWRITE to replace the data during the next ETL cycle.

This statement removes all the data and associated data files in the table. It can remove data files from internal tables, external tables, partitioned tables, and tables mapped to HBase or the Amazon Simple Storage Service (S3). The data removal applies to the entire table, including all partitions of a partitioned table.

Any statistics produced by the COMPUTE STATS statement are reset when the data is removed.

Make sure that you are in the correct database before truncating a table, either by issuing a USE statement first or by using a fully qualified name *db_name*.*table_name*.

The optional TABLE keyword does not affect the behavior of the statement.

The optional IF EXISTS clause makes the statement succeed whether or not the table exists. If the table does exist, it is truncated; if it does not exist, the statement has no effect. This capability is useful in standardized setup scripts that are might be run both before and after some of the tables exist. This clause is available in Impala 2.5 and higher.

Amazon S3 considerations:

Although Impala cannot write new data to a table stored in the Amazon S3 filesystem, the TRUNCATE TABLE statement can remove data files from S3.

Cancellation: Cannot be cancelled.

HDFS permissions:

The user ID that the `impalad` daemon runs under, typically the impala user, must have write permission for all the files and directories that make up the table.

Kudu considerations:

Currently, the TRUNCATE TABLE statement cannot be used with Kudu tables.

Examples:

The following example shows a table containing some data and with table and column statistics. After the TRUN CATE TABLE statement, the data is removed and the statistics are reset.

```
CREATE TABLE truncate_demo (x INT);
INSERT INTO truncate_demo VALUES (1), (2), (4), (8);
SELECT COUNT(*) FROM truncate_demo;
+----------+
| count(*) |
+----------+
| 4        |
+----------+
COMPUTE STATS truncate_demo;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 1 column(s). |
+-----------------------------------------+
SHOW TABLE STATS truncate_demo;
+-------+--------+------+-------------+-------------------+--------+----
---------------+
| #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Incre
mental stats |
+-------+--------+------+-------------+-------------------+--------+----
---------------+
| 4     | 1      | 8B   | NOT CACHED   | NOT CACHED        | TEXT   | false
           |
+-------+--------+------+-------------+-------------------+--------+----
---------------+
SHOW COLUMN STATS truncate_demo;
```

```
+--------+------+------------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+------+------------------+--------+----------+----------+
| x      | INT  | 4                | -1     | 4        | 4        |
+--------+------+------------------+--------+----------+----------+

-- After this statement, the data and the table/column stats will be gone.
TRUNCATE TABLE truncate_demo;
SELECT COUNT(*) FROM truncate_demo;
+----------+
| count(*) |
+----------+
| 0        |
+----------+
SHOW TABLE STATS truncate_demo;
+-------+--------+------+--------------+-------------------+--------+-----
--------------+
| #Rows | #Files | Size | Bytes Cached | Cache Replication | Format | Increm
ental stats |
+-------+--------+------+--------------+-------------------+--------+-----
--------------+
| -1    | 0      | 0B   | NOT CACHED   | NOT CACHED        | TEXT   | false
           |
+-------+--------+------+--------------+-------------------+--------+-----
--------------+
SHOW COLUMN STATS truncate_demo;
+--------+------+------------------+--------+----------+----------+
| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
+--------+------+------------------+--------+----------+----------+
| x      | INT  | -1               | -1     | 4        | 4        |
+--------+------+------------------+--------+----------+----------+
```

The following example shows how the IF EXISTS clause allows the TRUNCATE TABLE statement to be run without error whether or not the table exists:

```
CREATE TABLE staging_table1 (x INT, s STRING);
Fetched 0 row(s) in 0.33s

SHOW TABLES LIKE 'staging*';
+----------------+
| name           |
+----------------+
| staging_table1 |
+----------------+
Fetched 1 row(s) in 0.25s
-- Our ETL process involves removing all data from several staging tables
-- even though some might be already dropped, or not created yet.

TRUNCATE TABLE IF EXISTS staging_table1;
Fetched 0 row(s) in 5.04s

TRUNCATE TABLE IF EXISTS staging_table2;
Fetched 0 row(s) in 0.25s

TRUNCATE TABLE IF EXISTS staging_table3;
Fetched 0 row(s) in 0.25s
```

Related Information

Impala with Amazon S3

UPDATE statement

The UPDATE statement updates an arbitrary number of rows in a Kudu table. This statement only works for Impala tables that use the Kudu storage engine.

Syntax:

```
UPDATE [database_name.]table_name SET col = val [, col = val ... ]
  [ FROM joined_table_refs ]
  [ WHERE where_conditions ]
```

Usage notes:

None of the columns that make up the primary key can be updated by the SET clause.

The conditions in the WHERE clause are the same ones allowed for the SELECT statement.

If the WHERE clause is omitted, all rows in the table are updated.

The conditions in the WHERE clause can refer to any combination of primary key columns or other columns. Referring to primary key columns in the WHERE clause is more efficient than referring to non-primary key columns.

Because Kudu currently does not enforce strong consistency during concurrent DML operations, be aware that the results after this statement finishes might be different than you intuitively expect:

- If some rows cannot be updated because their some primary key columns are not found, due to their being deleted by a concurrent DELETE operation, the statement succeeds but returns a warning.
- An UPDATE statement might also overlap with INSERT, UPDATE, or UPSERT statements running concurrently on the same table. After the statement finishes, there might be more or fewer matching rows than expected in the table because it is undefined whether the UPDATE applies to rows that are inserted or updated while the UPDATE is in progress.

The number of affected rows is reported in an `impala-shell` message and in the query profile.

The optional FROM clause lets you restrict the updates to only the rows in the specified table that are part of the result set for a join query. The join clauses can include non-Kudu tables, but the table from which the rows are deleted must be a Kudu table.

Statement type: DML

Important: After adding or replacing data in a table used in performance-critical queries, issue a COMPUTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

The following examples show how to perform a simple update on a table, with or without a WHERE clause:

```
-- Set all rows to the same value for column c3.
-- In this case, c1 and c2 are primary key columns
-- and so cannot be updated.
UPDATE kudu_table SET c3 = 'not applicable';

-- Update only the rows that match the condition.
UPDATE kudu_table SET c3 = NULL WHERE c1 > 100 AND c3 IS NULL;

-- Does not update any rows, because the WHERE condition is always false.
UPDATE kudu_table SET c3 = 'impossible' WHERE 1 = 0;

-- Change the values of multiple columns in a single UPDATE statement.
UPDATE kudu_table SET c3 = upper(c3), c4 = FALSE, c5 = 0 WHERE c6 = TRUE;
```

The following examples show how to perform an update using the FROM keyword with a join clause:

```
-- Uppercase a column value, only for rows that have
-- an ID that matches the value from another table.
UPDATE kudu_table SET c3 = upper(c3)
  FROM kudu_table JOIN non_kudu_table
  ON kudu_table.id = non_kudu_table.id;

-- Same effect as previous statement.
-- Assign table aliases in FROM clause, then refer to
-- short names elsewhere in the statement.
UPDATE t1 SET c3 = upper(c3)
  FROM kudu_table t1 JOIN non_kudu_table t2
  ON t1.id = t2.id;

-- Same effect as previous statements, but more efficient.
-- Use WHERE clause to skip updating values that are
-- already uppercase.
UPDATE t1 SET c3 = upper(c3)
  FROM kudu_table t1 JOIN non_kudu_table t2
  ON t1.id = t2.id
  WHERE c3 != upper(c3);
```

UPSERT statement

The UPSERT statement acts as a combination of the INSERT and UPDATE statements.

For each row processed by the UPSERT statement:

- If another row already exists with the same set of primary key values, the other columns are updated to match the values from the row being “UPSERTed”.
- If there is not any row with the same set of primary key values, the row is created, the same as if the INSERT statement was used.

This statement only works for Impala tables that use the Kudu storage engine.

Syntax:

```
UPSERT [hint_clause] INTO [TABLE] [db_name.]table_name
  [(column_list)]
{
    [hint_clause] select_statement
  | VALUES (value [, value ...]) [, (value [, value ...]) ...]
}

hint_clause ::= [SHUFFLE] | [NOSHUFFLE]
  (Note: the square brackets are part of the syntax.)
```

The *select_statement* clause can use the full syntax, such as WHERE and JOIN clauses.

Statement type: DML

Usage notes:

If you specify a column list, any omitted columns in the inserted or updated rows are set to their default value (if the column has one) or NULL (if the column does not have a default value). Therefore, if a column is not nullable and has no default value, it must be included in the column list for any UPSERT statement. Because all primary key columns meet these conditions, all the primary key columns must be specified in every UPSERT statement.

Because Kudu tables can efficiently handle small incremental changes, the VALUES clause is more practical to use with Kudu tables than with HDFS-based tables.

Important: After adding or replacing data in a table used in performance-critical queries, issue a COMP UTE STATS statement to make sure all statistics are up-to-date. Consider updating statistics for a table after any INSERT, LOAD DATA, or CREATE TABLE AS SELECT statement in Impala, or after loading data through Hive and doing a REFRESH *table_name* in Impala. This technique is especially important for tables that are very large, used in join queries, or both.

Examples:

```
UPSERT INTO kudu_table (pk, c1, c2, c3) VALUES (0, 'hello', 50, true), (1,
'world', -1, false);
UPSERT INTO production_table SELECT * FROM staging_table;
UPSERT INTO production_table SELECT * FROM staging_table WHERE c1 IS NOT
NULL AND c2 > 0;
```

USE statement

The USE statement switches the current session to a specified database. The *current database* is where any CREA TE TABLE, INSERT, SELECT, or other statements act when you specify a table or other object name, without prefixing it with a database name. The new current database applies for the duration of the session or unti another USE statement is executed.

Syntax:

```
USE db_name
```

By default, when you connect to an Impala instance, you begin in a database named default.

Usage notes:

Switching the default database is convenient in the following situations:

- To avoid qualifying each reference to a table with the database name. For example, SELECT * FROM t1 JOIN t2 rather than SELECT * FROM db.t1 JOIN db.t2.
- To do a sequence of operations all within the same database, such as creating a table, inserting data, and querying the table.

To start the `impala-shell` interpreter and automatically issue a USE statement for a particular database, specify the option -d *db_name* for the `impala-shell` command. The -d option is useful to run SQL scripts, such as setup or test scripts, against multiple databases without hardcoding a USE statement into the SQL source.

Examples:

See the *CREATE DATABASE* statement for examples covering CREATE DATABASE, USE, and DROP DAT ABASE.

Cancellation: Cannot be cancelled.

HDFS permissions: This statement does not touch any HDFS files or directories, therefore no HDFS permissions are required.

Related Information

CREATE DATABASE statement

VALUES statement

The VALUES clause can be used as stand-alone statement, in the INSERT statement, and in the SELECT statement to construct a data set without creating a table.

Syntax:

```
VALUES (row)[, (row), ...];
```

```
SELECT select_list FROM (VALUES (row)[, (row), ...]) AS alias;

row ::= column [[AS alias], column [AS alias], ...]
```

- The VALUES keyword is followed by a comma separated list of one or more *row*s.
- *row* is a comma-separated list of one or more *column*s.
- Each *row* must have the same number of *column*s.
- *column* can be a constant, a variable, or an expression.
- The corresponding *column*s must have compatible data types in all *row*s. See the third query in the Examples section below.
- By default, the first row is used to name columns. But using the AS keyword, you can optionally give the column an *alias*.
- If used in the SELECT statement, the AS keyword with an *alias* is required.
- *select_list* is the columns to be selected for the result set.

Examples:

```
> SELECT * FROM (VALUES(4,5,6),(7,8,9)) AS t;
+---+---+---+
| 4 | 5 | 6 |
+---+---+---+
| 4 | 5 | 6 |
| 7 | 8 | 9 |
+---+---+---+

> SELECT * FROM (VALUES(1 AS c1, true AS c2, 'abc' AS c3),(100,false,'xyz
')) AS t;
+-----+-------+-----+
| c1  | c2    | c3  |
+-----+-------+-----+
| 1   | true  | abc |
| 100 | false | xyz |
+-----+-------+-----+

> VALUES (CAST('2019-01-01' AS TIMESTAMP)), ('2019-02-02');
+----------------------------------+
| cast('2019-01-01' as timestamp) |
+----------------------------------+
| 2019-01-01 00:00:00              |
| 2019-02-02 00:00:00              |
+----------------------------------+
```

Related Information

SELECT statement

Optimizer hints in Impala

The Impala SQL dialect supports query hints, for fine-tuning the inner workings of queries. Specify hints as a temporary workaround for expensive queries, where missing statistics or other factors cause inefficient performance.

Hints are most often used for the most resource-intensive kinds of Impala queries:

- Join queries involving large tables, where intermediate result sets are transmitted across the network to evaluate the join conditions.
- Inserting into partitioned Parquet tables, where many memory buffers could be allocated on each host to hold intermediate results for each partition.

Syntax:

In Impala 2.0 and higher, you can specify the hints inside comments that use either the /* */ or -- notation. Specify a + symbol immediately before the hint name. Recently added hints are only available using the /* */ and -- notation. For clarity, the /* */ and -- styles are used in the syntax and examples throughout this section. With the /* */ or -- notation for hints, specify a + symbol immediately before the first hint name. Multiple hints can be specified separated by commas, for example /* +clustered,shuffle */

```
SELECT STRAIGHT_JOIN select_list FROM
join_left_hand_table
  JOIN /* +BROADCAST|SHUFFLE */
join_right_hand_table
remainder_of_query;

SELECT select_list FROM
join_left_hand_table
  JOIN -- +BROADCAST|SHUFFLE
join_right_hand_table
remainder_of_query;

INSERT insert_clauses
  /* +SHUFFLE|NOSHUFFLE */
  SELECT remainder_of_query;

INSERT insert_clauses
  -- +SHUFFLE|NOSHUFFLE
  SELECT remainder_of_query;

INSERT /* +SHUFFLE|NOSHUFFLE */
  insert_clauses
  SELECT remainder_of_query;

INSERT -- +SHUFFLE|NOSHUFFLE
  insert_clauses
  SELECT remainder_of_query;

UPSERT /* +SHUFFLE|NOSHUFFLE */
  upsert_clauses
  SELECT remainder_of_query;

UPSERT -- +SHUFFLE|NOSHUFFLE
  upsert_clauses
  SELECT remainder_of_query;

SELECT select_list
 FROM
table_ref
  /* +{SCHEDULE_CACHE_LOCAL | SCHEDULE_DISK_LOCAL | SCHEDULE_REMOTE}
    [,RANDOM_REPLICA] */
remainder_of_query;

INSERT insert_clauses
  -- +CLUSTERED
  SELECT remainder_of_query;

INSERT insert_clauses
  /* +CLUSTERED */
  SELECT remainder_of_query;

INSERT -- +CLUSTERED
  insert_clauses
  SELECT remainder_of_query;

INSERT /* +CLUSTERED */
  insert_clauses
```

```
  SELECT remainder_of_query;

UPSERT -- +CLUSTERED
  upsert_clauses
  SELECT remainder_of_query;

UPSERT /* +CLUSTERED */
  upsert_clauses
  SELECT remainder_of_query;

CREATE /* +SHUFFLE|NOSHUFFLE */
  table_clauses
  AS SELECT remainder_of_query;

CREATE -- +SHUFFLE|NOSHUFFLE
  table_clauses
  AS SELECT remainder_of_query;

CREATE /* +CLUSTERED|NOCLUSTERED */
  table_clauses
  AS SELECT remainder_of_query;

CREATE -- +CLUSTERED|NOCLUSTERED
  table_clauses
  AS SELECT remainder_of_query;
```

The square bracket style hints are supported for backward compatibility, but the syntax is deprecated and will be removed in a future release. For that reason, any newly added hints are not available with the square bracket syntax.

```
SELECT STRAIGHT_JOIN select_list FROM
join_left_hand_table
  JOIN [{ /* +BROADCAST */ | /* +SHUFFLE */ }]
join_right_hand_table
remainder_of_query;

INSERT insert_clauses
  [{ /* +SHUFFLE */ | /* +NOSHUFFLE */ }]
  [/* +CLUSTERED */]
  SELECT remainder_of_query;

UPSERT [{ /* +SHUFFLE */ | /* +NOSHUFFLE */ }]
  [/* +CLUSTERED */]
  upsert_clauses
  SELECT remainder_of_query;
```

Usage notes:

With both forms of hint syntax, include the STRAIGHT_JOIN keyword immediately after the SELECT and any DISTINCT or ALL keywords to prevent Impala from reordering the tables in a way that makes the join-related hints ineffective.

The STRAIGHT_JOIN hint affects the join order of table references in the query block containing the hint. It does not affect the join order of nested queries, such as views, inline views, or WHERE-clause subqueries. To use this hint for performance tuning of complex queries, apply the hint to all query blocks that need a fixed join order.

To reduce the need to use hints, run the COMPUTE STATS statement against all tables involved in joins, or used as the source tables for INSERT ... SELECT operations where the destination is a partitioned Parquet table. Do this operation after loading data or making substantial changes to the data within each table. Having up-to-date statistics helps Impala choose more efficient query plans without the need for hinting.

To see which join strategy is used for a particular query, examine the EXPLAIN output for that query.

Hints for join queries:

The /* +BROADCAST */ and /* +SHUFFLE */ hints control the execution strategy for join queries. Specify one of the following constructs immediately after the JOIN keyword in a query:

- /* +SHUFFLE */ makes that join operation use the “partitioned” technique, which divides up corresponding rows from both tables using a hashing algorithm, sending subsets of the rows to other nodes for processing. (The keyword SHUFFLE is used to indicate a “partitioned join”, because that type of join is not related to “partitioned tables”.) Since the alternative “broadcast” join mechanism is the default when table and index statistics are unavailable, you might use this hint for queries where broadcast joins are unsuitable; typically, partitioned joins are more efficient for joins between large tables of similar size.
- /* +BROADCAST */ makes that join operation use the “broadcast” technique that sends the entire contents of the right-hand table to all nodes involved in processing the join. This is the default mode of operation when table and index statistics are unavailable, so you would typically only need it if stale metadata caused Impala to mistakenly choose a partitioned join operation. Typically, broadcast joins are more efficient in cases where one table is much smaller than the other. (Put the smaller table on the right side of the JOIN operator.)

Hints for INSERT ... SELECT and CREATE TABLE AS SELECT (CTAS):

When inserting into partitioned tables, such as using the Parquet file format, you can include a hint in the INSERT or CREATE TABLE AS SELECT(CTAS) statements to fine-tune the overall performance of the operation and its resource usage.

You would only use hints if an INSERT or CTAS into a partitioned table was failing due to capacity limits, or if such an operation was succeeding but with less-than-optimal performance.

- /* +SHUFFLE */ and /* +NOSHUFFLE */ Hints
  - /* +SHUFFLE */ adds an exchange node, before writing the data, which re-partitions the result of the SELECT based on the partitioning columns of the target table. With this hint, only one node writes to a partition at a time, minimizing the global number of simultaneous writes and the number of memory buffers holding data for individual partitions. This also reduces fragmentation, resulting in fewer files. Thus it reduces overall resource usage of the INSERT or CTAS operation and allows some operations to succeed that otherwise would fail. It does involve some data transfer between the nodes so that the data files for a particular partition are all written on the same node.

    Use /* +SHUFFLE */ in cases where an INSERT or CTAS statement fails or runs inefficiently due to all nodes attempting to write data for all partitions.

    If the table is unpartitioned or every partitioning expression is constant, then /* +SHUFFLE */ will cause every write to happen on the coordinator node.
  - /* +NOSHUFFLE */ does not add exchange node before inserting to partitioned tables and disables re-partitioning. So the selected execution plan might be faster overall, but might also produce a larger number of small data files or exceed capacity limits, causing the INSERT or CTAS operation to fail.

    Impala automatically uses the /* +SHUFFLE */ method if any partition key column in the source table, mentioned in the SELECT clause, does not have column statistics. In this case, use the /* +NOSHUFFLE */ hint if you want to override this default behavior.
  - If column statistics are available for all partition key columns in the source table mentioned in the INSERT ... SELECT or CTAS query, Impala chooses whether to use the /* +SHUFFLE */ or /* +NOSHUFFLE */ technique based on the estimated number of distinct values in those columns and the number of nodes involved in the operation. In this case, you might need the /* +SHUFFLE */ or the /* +NOSHUFFLE */ hint to override the execution plan selected by Impala.
- /* +CLUSTERED */ and /* +NOCLUSTERED */ Hints
  - /* +CLUSTERED */ sorts data by the partition columns before inserting to ensure that only one partition is written at a time per node. Use this hint to reduce the number of files kept open and the number of buffers kept in memory simultaneously. This technique is primarily useful for inserts into Parquet tables, where the large block size requires substantial memory to buffer data for multiple output files at once. This hint is available in Impala 2.8 or higher.

    Starting in Impala 3.0, /* +CLUSTERED */ is the default behavior for HDFS tables.

- /* +NOCLUSTERED */ does not sort by primary key before insert. This hint is available in Impala 2.8 or higher.

  Use this hint when inserting to Kudu tables.

  In the versions lower than Impala 3.0, /* +NOCLUSTERED */ is the default in HDFS tables.

Kudu consideration:

Starting from Impala 2.9, the INSERT or UPSERT operations into Kudu tables automatically add an exchange and a sort node to the plan that partitions and sorts the rows according to the partitioning/primary key scheme of the target table (unless the number of rows to be inserted is small enough to trigger single node execution). Since Kudu partitions and sorts rows on write, pre-partitioning and sorting takes some of the load off of Kudu and helps large INSERT operations to complete without timing out. However, this default behavior may slow down the end-to-end performance of the INSERT or UPSERT operations. Starting from Impala 2.10, you can use the /* +NOCLUSTERED */ and /* +NOSHUFFLE */ hints together to disable partitioning and sorting before the rows are sent to Kudu. Additionally, since sorting may consume a large amount of memory, consider setting the MEM_LIMIT query option for those queries.

Hints for scheduling of scan ranges (HDFS data blocks or Kudu tablets):

The hints /* +SCHEDULE_CACHE_LOCAL */, /* +SCHEDULE_DISK_LOCAL */, and /* +SCHEDULE_REMOTE */ have the same effect as specifying the REPLICA_PREFERENCE query option with the respective option settings of CACHE_LOCAL, DISK_LOCAL, or REMOTE.

Specifying the replica preference as a query hint always overrides the query option setting.

The hint /* +RANDOM_REPLICA */ is the same as enabling the SCHEDULE_RANDOM_REPLICA query option.

You can use these hints in combination by separating them with commas, for example, /* +SCHEDULE_CACHE_LOCAL,RANDOM_REPLICA */.

Specifying either the SCHEDULE_RANDOM_REPLICA query option or the corresponding RANDOM_REPLICA query hint enables the random tie-breaking behavior when processing data blocks during the query.

Suggestions versus directives:

In early Impala releases, hints were always obeyed and so acted more like directives. Once Impala gained join order optimizations, sometimes join queries were automatically reordered in a way that made a hint irrelevant. Therefore, the hints act more like suggestions in Impala 1.2.2 and higher.

To force Impala to follow the hinted execution mechanism for a join query, include the STRAIGHT_JOIN keyword in the SELECT statement. When you use this technique, Impala does not reorder the joined tables at all, so you must be careful to arrange the join order to put the largest table (or subquery result set) first, then the smallest, second smallest, third smallest, and so on. This ordering lets Impala do the most I/O-intensive parts of the query using local reads on the DataNodes, and then reduce the size of the intermediate result set as much as possible as each subsequent table or subquery result set is joined.

Restrictions:

Queries that include subqueries in the WHERE clause can be rewritten internally as join queries. Currently, you cannot apply hints to the joins produced by these types of queries.

Because hints can prevent queries from taking advantage of new metadata or improvements in query planning, use them only when required to work around performance issues, and be prepared to remove them when they are no longer required, such as after a new Impala release or bug fix.

In particular, the /* +BROADCAST */ and /* +SHUFFLE */ hints are expected to be needed much less frequently in Impala 1.2.2 and higher, because the join order optimization feature in combination with the COMPUTE STATS statement now automatically choose join order and join mechanism without the need to rewrite the query and add hints.

Compatibility:

The hints embedded within -- comments are compatible with Hive queries. The hints embedded within /* */ comments or [ ] square brackets are not recognized by or not compatible with Hive. For example, Hive raises an error for Impala hints within /* */ comments because it does not recognize the Impala hint names.

Considerations for views:

If you use a hint in the query that defines a view, the hint is preserved when you query the view. Impala internally rewrites all hints in views to use the -- comment notation, so that Hive can query such views without errors due to unrecognized hint names.

Examples:

For example, this query joins a large customer table with a small lookup table of less than 100 rows. The right-hand table can be broadcast efficiently to all nodes involved in the join. Thus, you would use the /* +broadcast */ hint to force a broadcast join strategy:

```
select straight_join customer.address, state_lookup.state_name
  from customer join /* +broadcast */ state_lookup
  on customer.state_id = state_lookup.state_id;
```

This query joins two large tables of unpredictable size. You might benchmark the query with both kinds of hints and find that it is more efficient to transmit portions of each table to other nodes for processing. Thus, you would use the /* +shuffle */ hint to force a partitioned join strategy:

```
select straight_join weather.wind_velocity, geospatial.altitude
  from weather join /* +shuffle */ geospatial
  on weather.lat = geospatial.lat and weather.long = geospatial.long;
```

For joins involving three or more tables, the hint applies to the tables on either side of that specific JOIN keyword. The STRAIGHT_JOIN keyword ensures that joins are processed in a predictable order from left to right. For example, this query joins t1 and t2 using a partitioned join, then joins that result set to t3 using a broadcast join:

```
select straight_join t1.name, t2.id, t3.price
  from t1 join /* +shuffle */ t2 join /* +broadcast */ t3
  on t1.id = t2.id and t2.id = t3.id;
```

Related Information

Background information about join queries

Performance considerations for join queries

Query options

Specify query options in the SET statement to apply the settings to the subsequently issued queries.

Some query options are useful in day-to-day operations for improving usability, performance, or flexibility.

Other query options control special-purpose aspects of Impala operation and are intended primarily for advanced debugging or troubleshooting.

Options with Boolean parameters can be set to 1 or true to enable, or 0 or false to turn off.

You can set query options directly through the JDBC and ODBC interfaces by using the SET statement. Formerly, SET was only available as a command within the `impala-shell` interpreter.

You can set query options for an `impala-shell` session by specifying one or more command-line arguments of the form --query_option=*option=value*.

Impala supports the following query options:

- ABORT_ON_ERROR
- ALLOW_ERASURE_CODED_FILES
- APPX_COUNT_DISTINCT
- BATCH_SIZE

- BUFFER_POOL_LIMIT
- COMPRESSION_CODEC
- COMPUTE_STATS_MIN_SAMPLE_SIZE
- DEBUG_ACTION
- DECIMAL_V2
- DEFAULT_FILE_FORMAT
- DEFAULT_HINTS_INSERT_STATEMENT
- DEFAULT_JOIN_DISTRIBUTION_MODE
- DEFAULT_SPILLABLE_BUFFER_SIZE
- DEFAULT_TRANSACTIONAL_TYPE
- DISABLE_CODEGEN
- DISABLE_CODEGEN_ROWS_THRESHOLD
- DISABLE_HBASE_NUM_ROWS_ESTIMATE
- DISABLE_ROW_RUNTIME_FILTERING
- DISABLE_STREAMING_PREAGGREGATIONS
- DISABLE_UNSAFE_SPILLS
- ENABLE_EXPR_REWRITES
- EXEC_SINGLE_NODE_ROWS_THRESHOLD
- EXEC_TIME_LIMIT_S
- EXPLAIN_LEVEL
- FETCH_ROWS_TIMEOUT_MS
- HBASE_CACHE_BLOCKS
- HBASE_CACHING
- IDLE_SESSION_TIMEOUT
- KUDU_READ_MODE
- LIVE_PROGRESS
- LIVE_SUMMARY
- MAX_ERRORS
- MAX_MEM_ESTIMATE_FOR_ADMISSION
- MAX_NUM_RUNTIME_FILTERS
- MAX_RESULT_SPOOLING_MEM
- MAX_ROW_SIZE
- MAX_SCAN_RANGE_LENGTH
- MAX_SPILLED_RESULT_SPOOLING_MEM
- MEM_LIMIT
- MIN_SPILLABLE_BUFFER_SIZE
- MT_DOP
- NUM_NODES
- NUM_ROWS_PRODUCED_LIMIT
- NUM_SCANNER_THREADS
- OPTIMIZE_PARTITION_KEY_SCANS
- PARQUET_COMPRESSION_CODEC
- PARQUET_ANNOTATE_STRINGS_UTF8
- PARQUET_ARRAY_RESOLUTION
- PARQUET_DICTIONARY_FILTERING
- PARQUET_FALLBACK_SCHEMA_RESOLUTION
- PARQUET_FILE_SIZE
- PARQUET_OBJECT_STORE_SPLIT_SIZE
- PARQUET_PAGE_ROW_COUNT_LIMIT
- PARQUET_READ_PAGE_INDEX

- PARQUET_READ_STATISTICS
- PARQUET_WRITE_PAGE_INDEX
- PREFETCH_MODE
- QUERY_TIMEOUT_S
- REPLICA_PREFERENCE
- REQUEST_POOL
- RESOURCE_TRACE_RATIO
- RUNTIME_BLOOM_FILTER_SIZE
- RUNTIME_FILTER_MAX_SIZE
- RUNTIME_FILTER_MIN_SIZE
- RUNTIME_FILTER_MODE
- RUNTIME_FILTER_WAIT_TIME_MS
- S3_SKIP_INSERT_STAGING
- SCAN_BYTES_LIMIT
- SCHEDULE_RANDOM_REPLICA
- SCRATCH_LIMIT
- SHUFFLE_DISTINCT_EXPRS
- SPOOL_QUERY_RESULTS
- SUPPORT_START_OVER
- SYNC_DDL
- THREAD_RESERVATION_AGGREGATE_LIMIT
- THREAD_RESERVATION_LIMIT
- TIMEZONE
- TOPN_BYTES_LIMIT

ABORT_ON_ERROR query option

When this option is enabled, Impala cancels a query immediately when any of the nodes encounters an error, rather than continuing and possibly returning incomplete results. This option is disabled by default, to help gather maximum diagnostic information when an error occurs, for example, whether the same problem occurred on all nodes or only a single node. Currently, the errors that Impala can skip over involve data corruption, such as a column that contains a string value when expected to contain an integer value.

To control how much logging Impala does for non-fatal errors when ABORT_ON_ERROR is turned off, use the MAX_ERRORS option.

Type: BOOLEAN

Default: FALSE (0)

ALLOW_ERASURE_CODED_FILES query option

Use the ALLOW_ERASURE_CODED_FILES query option to enable or disable the support of erasure coded files in Impala. Until Impala is fully tested and certified with erasure coded files, this query option is set to FALSE by default.

When the ALLOW_ERASURE_CODED_FILES query option is set to FALSE, Impala returns an error when a query requires scanning an erasure coded file.

Type: BOOLEAN

Default: FALSE (0)

Added in: Impala 3.1

APPX_COUNT_DISTINCT query option

When the APPX_COUNT_DISTINCT query option is set to TRUE, Impala implicitly converts COUNT(DISTINCT) operations to the NDV() function calls. The resulting count is approximate

rather than precise. Enable the query option when a tolerable amount of error is acceptable in order to obtain faster query results than with a COUNT (DISTINCT) queries.

Type: BOOLEAN

Default: FALSE (0)

BATCH_SIZE query option

Number of rows evaluated at a time by SQL operators. Unspecified or a size of 0 uses a predefined default size. Using a large number improves responsiveness, especially for scan operations, at the cost of a higher memory footprint.

This option is primarily for testing during Impala development, or for use under the direction of Cloudera support.

Type: Number

Default: 0 (meaning the predefined default of 1024)

BUFFER_POOL_LIMIT query option

Defines a limit on the amount of memory that a query can allocate from the internal buffer pool. The value for this limit applies to the memory on each host, not the aggregate memory across the cluster.

Typically not changed by users, except during diagnosis of out-of-memory errors during queries.

You can set it to an absolute value, e.g. 8GB, or a relative value, e.g. 80%, based on the MEM_LIMIT setting.

Type: INT

Default: The default setting for this option is the lower of 80% of the MEM_LIMIT setting, or the MEM_LIMIT setting minus 100 MB.

Usage notes:

If queries encounter out-of-memory errors, consider decreasing the BUFFER_POOL_LIMIT setting to less than 80% of the MEM_LIMIT setting.

COMPRESSION_CODEC query option

When Impala writes Parquet data files using the INSERT statement, the underlying compression is controlled by the COMPRESSION_CODEC query option.

For the ZSTD compression, the optional compression level can be specified as shown in the syntax below.

Note: Prior to Impala 2.0, this option was named PARQUET_COMPRESSION_CODEC. In Impala 2.0 and later, the PARQUET_COMPRESSION_CODEC name is not recognized. Use the more general name COMPRESSION_CODEC for new code.

Syntax:

```
SET COMPRESSION_CODEC=codec_name; // Supported for all codecs.
SET COMPRESSION_CODEC=codec_name:compression_level; // Only su
pported for ZSTD.
```

The allowed values for this query option are SNAPPY (the default), GZIP, ZSTD, LZ4, and NONE.

ZSTD also supports setting a compression level. The lower the level, the faster the speed at the cost of compression ratio. Compression levels from 1 up to 22 are supported for ZSTD. The default compression level 3 is used, if one is not passed using the compression_codec query option.

Note: A Parquet file created with COMPRESSION_CODEC=NONE is still typically smaller than the original data, due to encoding schemes such as run-length encoding and dictionary encoding that are applied separately from compression.

The option value is not case-sensitive.

If the option is set to an unrecognized value, all kinds of queries will fail due to the invalid option setting, not just queries involving Parquet tables. (The value BZIP2 is also recognized, but is not compatible with Parquet tables.)

Type: STRING

Default: SNAPPY

COMPUTE_STATS_MIN_SAMPLE_SIZE query option

The COMPUTE_STATS_MIN_SAMPLE_SIZE query option specifies the minimum number of bytes that will be scanned in COMPUTE STATS TABLESAMPLE, regardless of the user-supplied sampling percent. This query option prevents sampling for very small tables where accurate stats can be obtained cheaply without sampling because the minimum sample size is required to get meaningful stats.

Type: INT

Default: 1GB

DEBUG_ACTION query option

Introduces artificial problem conditions within queries. For internal Cloudera debugging and troubleshooting.

Type: STRING

Default: An empty string

DECIMAL_V2 query option

A query option that changes behavior related to the DECIMAL data type. Set this option to FALSE for backward compatibility to Impala 2.x.

Type: Boolean

Default: TRUE

DEFAULT_FILE_FORMAT query option

Use the DEFAULT_FILE_FORMAT query option to set the default table file format. The following values are supported:

- TEXT (0)
- RC_FILE (1)
- SEQUENCE_FILE (2)
- AVRO (3)
- PARQUET (4)
- KUDU (5)
- ORC (6)

In impala-shell or Hue, the SET DEFAULT_FILE_FORMAT statement will not return an error when the option was set to an unsupported value. Impala validates the value and returns an error when you submitted the next query.

Type: Enum as specified above

Default: TEXT (0)

Added in: Impala 3.3

DEFAULT_HINTS_INSERT_STATEMENT query option

The DEFAULT_HINTS_INSERT_STATEMENT query option sets the default hints for the INSERT statements with no optimizer hint specified.

When there are hints specified in an INSERT statement, these default hints are ignored.

You can specify multiple hints separated by ':'.

For example:

```
SET DEFAULT_HINTS_INSERT_STATEMENT=CLUSTERED;
SET DEFAULT_HINTS_INSERT_STATEMENT=SHUFFLE;
SET DEFAULT_HINTS_INSERT_STATEMENT=NOCLUSTERED:NOSHUFFLE;
```

The default hints apply to the HDFS and Kudu table formats and are ignored for the HBase table format.

Type: STRING

Default: No default hint

Added in: Impala 3.3

DEFAULT_JOIN_DISTRIBUTION_MODE query option

Impala optimizes join queries based on the presence of table statistics, which are produced by the Impala COMPUTE STATS statement. By default, when a table involved in the join query does not have statistics, Impala uses the “broadcast” technique that transmits the entire contents of the table to all executor nodes participating in the query. If one table involved in a join has statistics and the other does not, the table without statistics is broadcast. If both tables are missing statistics, the table on the right-hand side of the join is broadcast. This behavior is appropriate when the table involved is relatively small, but can lead to excessive network, memory, and CPU overhead if the table being broadcast is large.

Because Impala queries frequently involve very large tables, and suboptimal joins for such tables could result in spilling or out-of-memory errors, the setting DEFAULT_JOIN_DISTRIBUTION_MODE=SHUFFLE lets you override the default behavior. The shuffle join mechanism divides the corresponding rows of each table involved in a join query using a hashing algorithm, and transmits subsets of the rows to other nodes for processing. Typically, this kind of join is more efficient for joins between large tables of similar size.

The setting DEFAULT_JOIN_DISTRIBUTION_MODE=SHUFFLE is recommended when setting up and deploying new clusters, because it is less likely to result in serious consequences such as spilling or out-of-memory errors if the query plan is based on incomplete information. This setting is not the default, to avoid changing the performance characteristics of join queries for clusters that are already tuned for their existing workloads.

The allowed values are BROADCAST (0) or SHUFFLE (1).

Type: INT

Examples:

The following examples demonstrate appropriate scenarios for each setting of this query option.

```
-- Create a billion-row table.
create table big_table stored as parquet
  as select * from huge_table limit 1e9;

-- For a big table with no statistics, the
-- shuffle join mechanism is appropriate.
set default_join_distribution_mode=shuffle;
```

```
...join queries involving the big table...
```

```
-- Create a hundred-row table.
create table tiny_table stored as parquet
  as select * from huge_table limit 100;

-- For a tiny table with no statistics, the
-- broadcast join mechanism is appropriate.
set default_join_distribution_mode=broadcast;

...join queries involving the tiny table...
```

```
compute stats tiny_table;
compute stats big_table;
-- Once the stats are computed, the query option has
-- no effect on join queries involving these tables.
-- Impala can determine the absolute and relative sizes
-- of each side of the join query by examining the
-- row size, cardinality, and so on of each table.

...join queries involving both of these tables...
```

DEFAULT_SPILLABLE_BUFFER_SIZE query option

Specifies the default size for a memory buffer used when the spill-to-disk mechanism is activated, for example for queries against a large table with no statistics, or large join operations.

Type: INT

Default: 2097152 (2 MB)

Unit: A numeric argument represents a size in bytes; you can also use a suffix of m or mb for megabytes, or g or gb for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Usage notes:

This query option sets an upper bound on the size of the internal buffer size that can be used during spill-to-disk operations. The actual size of the buffer is chosen by the query planner.

If overall query performance is limited by the time needed for spilling, consider increasing the DEFAULT_SPILLABLE_BUFFER_SIZE setting. Larger buffer sizes result in Impala issuing larger I/O requests to storage devices, which might result in higher throughput, particularly on rotational disks.

The tradeoff with a large value for this setting is increased memory usage during spill-to-disk operations. Reducing this value may reduce memory consumption.

To determine if the value for this setting is having an effect by capping the spillable buffer size, you can see the buffer size chosen by the query planner for a particular query. EXPLAIN the query while the setting EXPLAIN_LEVEL=2 is in effect.

DEFAULT_TRANSACTIONAL_TYPE query option

Use the DEFAULT_TRANSACTION_TYPE query option to create insert-only transactional tables by default.

Note: The DEFAULT_TRANSACTION_TYPE query option applies only when integrated with Hive 3.

The supported values are:

- NONE: The table will not be created as transactional.
- INSERT_ONLY: The table will be created as transactional.

If either table properties, transactional or transactional_properties, are set, this query option is ignored.

Note: The DEFAULT_TRANSACTION_TYPE query option does not affect external, Kudu, or HBase tables as these cannot be transactional.

Type: Enum as specified above

Default: NONE

Added in: Impala 3.3

DISABLE_CODEGEN query option

The DISABLE_CODEGEN is a debug option, and it's used to work around any issues with Impala's runtime code generation. If a query fails with an “illegal instruction” or other hardware-specific message, try setting DISABLE_CODEGEN=true and running the query again. If the query succeeds only when the DISABLE_CODEGEN option is turned on, submit the problem to Cloudera support and include that detail in the problem report.

Most queries will run significantly slower with DISABLE_CODEGEN=true.

In Impala 2.10 and higher, the DISABLE_CODEGEN_ROWS_THRESHOLD optimisation automatically disables codegen for small queries because short-running queries may run faster without the overhead of codegen.

The following values are supported:

- TRUE or 1: Disables codegen.
- FALSE or 0: Enables codegen.

Type: BOOLEAN

Default: FALSE (0)

DISABLE_CODEGEN_ROWS_THRESHOLD query option

Specify the number of rows processed per Impala daemon which is the cutoff point below which Impala disables native code generation for the whole query. Native code generation is beneficial for queries that process many rows because it reduces the time taken to process of each row. However, generating the native code adds latency to query startup. Therefore, automatically disabling codegen for queries that process relatively small amounts of data can improve query response time.

This setting applies to queries where the number of rows processed can be accurately determined, either through table and column statistics, or by the presence of a LIMIT clause. If Impala cannot accurately estimate the number of rows, then this setting does not apply.

If a query uses the complex data types STRUCT, ARRAY, or MAP, then codegen is never automatically disabled regardless of the DISABLE_CODEGEN_ROWS_THRESHOLD setting.

Type: Number

Default: 50000

Usage notes: Typically, you increase the default value to make this optimization apply to more queries. If incorrect or corrupted table and column statistics cause Impala to apply this optimization incorrectly to queries that actually involve substantial work, you might see the queries being slower as a result of codegen being disabled. In that case, recompute statistics with the COMPUTE STATS or COMPUTE INCREMENTAL STATS statement. If there is a problem collecting accurate statistics, you can turn this feature off by setting the value to 0.

DISABLE_HBASE_NUM_ROWS_ESTIMATE

Use the DISABLE_HBASE_NUM_ROWS_ESTIMATE query option to disable key sampling of HBase tables in row count and row size estimation.

While generating a plan for an HBase query, the planner samples the underlying HBase tables to estimate their row count and row size, and the sampling can negatively impact the planning time. When the HBase table stats do not change much in a short time, disable the sampling by setting the DISABLE_HBASE_NUM_ROWS_ESTIMATE query option to TRUE. And Impala planner will fall back to using Hive Metastore (HMS) table stats instead.

When DISABLE_HBASE_NUM_ROWS_ESTIMATE query option is set to TRUE, you need to update the HMS table stats by running COMPUTE STATS. Alternatively, you can manually set table statistics by running ALTER TABLE.

The following values are supported:

- TRUE or 1: Disables the normal key sampling of HBase tables and uses HMS table stats for estimation.
- FALSE or 0: Enables the normal sampling of HBase tables.

Type: BOOLEAN

Default: FALSE

DISABLE_ROW_RUNTIME_FILTERING query

The DISABLE_ROW_RUNTIME_FILTERING query option reduces the scope of the runtime filtering feature. Queries still dynamically prune partitions, but do not apply the filtering logic to individual rows within partitions.

Only applies to queries against HDFS-based tables using the Parquet file format. For other file formats, Impala only prunes at the level of partitions, not individual rows.

This query option only applies to queries against

Type: BOOLEAN

Default: FALSE (0)

Usage notes:

Impala automatically evaluates whether the per-row filters are being effective at reducing the amount of intermediate data. Therefore, this option is typically only needed for the rare case where Impala cannot accurately determine how effective the per-row filtering is for a query.

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Kudu consideration:

When applied to a query involving a Kudu table, this option turns off all runtime filtering for the Kudu table.

DISABLE_STREAMING_PREAGGREGATIONS query option

Turns off the “streaming preaggregation” optimization that is available in Impala 2.5 and higher. This optimization reduces unnecessary work performed by queries that perform aggregation operations on columns with few or no duplicate values, for example DISTINCT *id_column* or GROUP BY *unique_column*. If the optimization causes regressions in existing queries that use aggregation functions, you can turn it off as needed by setting this query option.

Type: BOOLEAN

Default: FALSE (0)

Usage notes:

Typically, queries that would require enabling this option involve very large numbers of aggregated values, such as a billion or more distinct keys being processed on each worker node.

DISABLE_UNSAFE_SPILLS query option

Enable this option if you prefer to have queries fail when they exceed the Impala memory limit, rather than write temporary data to disk.

Queries that "spill" to disk typically complete successfully, when in earlier Impala releases they would have failed. However, queries with exorbitant memory requirements due to missing statistics or inefficient join clauses could become so slow as a result that you would rather have them cancelled automatically and reduce the memory usage through standard Impala tuning techniques.

This option prevents only "unsafe" spill operations, meaning that one or more tables are missing statistics or the query does not include a hint to set the most efficient mechanism for a join or INSE RT ... SELECT into a partitioned table. These are the tables most likely to result in suboptimal execution plans that could cause unnecessary spilling. Therefore, leaving this option enabled is a good way to find tables on which to run the COMPUTE STATS statement.

Type: BOOLEAN

Default: FALSE (0)

ENABLE_EXPR_REWRITES query option

The ENABLE_EXPR_REWRITES query option controls whether to enable or disable the query compile time optimizations. These optimizations rewrite the expression trees to a more compact and optimized form that helps avoid redundant expression evaluation at run time. Performance optimizations controlled by this query option include:

- Constant folding
- Extracting common conjuncts from disjunctions
- Simplify conditionals with constant conditions

Set the option to FALSE or 0 to disable the performance optimizations.

Type: BOOLEAN

Default: TRUE (1)

EXEC_SINGLE_NODE_ROWS_THRESHOLD query option

Specify the number of rows scanned as the cutoff point below which Impala treats a query as a "small" query, turning off optimizations such as parallel execution and native code generation. The overhead for these optimizations is applicable for queries involving substantial amounts of data, but it makes sense to skip them for queries involving tiny amounts of data. Reducing the overhead for small queries allows Impala to complete them more quickly, keeping admission control slots, CPU, memory, and so on available for resource-intensive queries.

This setting applies to queries where the number of rows processed can be accurately determined, either through table and column statistics, or by the presence of a LIMIT clause. If Impala cannot accurately estimate the number of rows, then this setting does not apply.

In Impala 2.3 and higher, where Impala supports the complex data types STRUCT, ARRAY, and MAP, if a query refers to any column of those types, the small-query optimization is turned off for that query regardless of the EXEC_SINGLE_NODE_ROWS_THRESHOLD setting.

For a query that is determined to be "small", all work is performed on the coordinator node. This might result in some I/O being performed by remote reads. The savings from not distributing the query work and not generating native code are expected to outweigh any overhead from the remote reads.

Type: Number

Default: 100

Usage notes: Typically, you increase the default value to make this optimization apply to more queries. If incorrect or corrupted table and column statistics cause Impala to apply this optimization incorrectly to queries that actually involve substantial work, you might see the queries being slower as a result of remote reads. In that case, recompute statistics with the COMPUTE STATS or COMP UTE INCREMENTAL STATS statement. If there is a problem collecting accurate statistics, you can turn this feature off by setting the value to -1.

EXEC_TIME_LIMIT_S query option

The EXEC_TIME_LIMIT_S query option sets a time limit on query execution in seconds. If a query is still executing when time limit expires, it is automatically canceled. The option is intended to prevent runaway queries that execute for much longer than intended.

For example, an Impala administrator could set a default value of EXEC_TIME_LIMIT_S=3600 for a resource pool to automatically kill queries that execute for longer than one hour. Then, if a user accidentally runs a large query that executes for more than one hour, it will be automatically killed after the time limit expires to free up resources. Users can override the default value per query or per session if they do not want the default EXEC_TIME_LIMIT_S value to apply to a specific query or a session.

Note:

The time limit only starts once the query is executing. Time spent planning the query, scheduling the query, or in admission control is not counted towards the execution time limit. SELECT statements are eligible for automatic cancellation until the client has fetched all result rows. DML queries are eligible for automatic cancellation until the DML statement has finished.

Type: Number

Default: 0 (no time limit)

EXPLAIN_LEVEL query option

Controls the amount of detail provided in the output of the EXPLAIN statement. The basic output can help you identify high-level performance issues such as scanning a higher volume of data or more partitions than you expect. The higher levels of detail show how intermediate results flow between nodes and how different SQL operations such as ORDER BY, GROUP BY, joins, and WHERE clauses are implemented within a distributed query.

Type: STRING or INT

Default: 1

Arguments:

The allowed range of numeric values for this option is 0 to 3:

- 0 or MINIMAL: A barebones list, one line per operation. Primarily useful for checking the join order in very long queries where the regular EXPLAIN output is too long to read easily.
- 1 or STANDARD: The default level of detail, showing the logical way that work is split up for the distributed query.
- 2 or EXTENDED: Includes additional detail about how the query planner uses statistics in its decision-making process, to understand how a query could be tuned by gathering statistics, using query hints, adding or removing predicates, and so on. In Impala 3.2 and higher, the output also includes the analyzed query with the cast information in the output header, and the implicit cast info in the Predicate section.
- 3 or VERBOSE: The maximum level of detail, showing how work is split up within each node into “query fragments” that are connected in a pipeline. This extra detail is primarily useful for low-level performance testing and tuning within Impala itself, rather than for rewriting the SQL code at the user level.

Changing the value of this option controls the amount of detail in the output of the EXPLAIN statement. The extended information from level 2 or 3 is especially useful during performance tuning, when you need to confirm whether the work for the query is distributed the way you expect, particularly for the most resource-intensive operations such as join queries against large tables, queries against tables with large numbers of partitions, and insert operations for Parquet tables. The extended information also helps to check estimated resource usage when you use the admission control or resource management features.

Usage notes:

Read the EXPLAIN output from bottom to top. The lowest lines represent the initial work of the query (scanning data files), the lines in the middle represent calculations done on each node and how intermediate results are transmitted from one node to another, and the topmost lines represent the final results being sent back to the coordinator node.

The numbers in the left column are generated internally during the initial planning phase and do not represent the actual order of operations, so it is not significant if they appear out of order in the EXPLAIN output.

At all EXPLAIN levels, the plan contains a warning if any tables in the query are missing statistics. Use the COMPUTE STATS statement to gather statistics for each table and suppress this warning. See *Tables and column statistics* for details about how the statistics help query performance.

The PROFILE command in `impala-shell` always starts with an explain plan showing full detail, the same as with EXPLAIN_LEVEL=3. After the explain plan comes the executive summary, the same output as produced by the SUMMARY command in `impala-shell`.

FETCH_ROWS_TIMEOUT_MS query option

Use the FETCH_ROWS_TIMEOUT_MS query option to control how long Impala waits for query results when clients fetch rows.

When this query option is set to 0, fetch requests wait indefinitely.

The timeout applies both when query result spooling is enabled and disabled:

- When result spooling is disabled (SPOOL_QUERY_RESULTS = FALSE), the timeout controls how long a client waits for a single row batch to be produced by the coordinator.
- When result spooling is enabled ((SPOOL_QUERY_RESULTS = TRUE), a client can fetch multiple row batches at a time, so this timeout controls the total time a client waits for row batches to be produced.

The timeout also applies to fetch requests issued against queries in the 'RUNNING' state. A 'RUNNING' query has no rows available, so any fetch request will wait until the query transitions to the 'FINISHED' state and for it to fetch all requested rows. A query in the 'FINISHED' state means that the rows are available to be fetched.

Type: INT

Default: 10000 (10 seconds)

Added in: Impala 3.4

HBASE_CACHE_BLOCKS query option

Setting this option is equivalent to calling the setCacheBlocks method of the class org.apache.hadoop.hbase.client.Scan in an HBase Java application. Helps to control the memory pressure on the HBase RegionServer, in conjunction with the HBASE_CACHING query option.

Type: BOOLEAN

Default: FALSE (0)

HBASE_CACHING query option

Setting this option is equivalent to calling the setCaching method of the class org.apache.hadoop.hbase.client.Scan in an HBase Java application. Helps to control the memory pressure on the HBase RegionServer, in conjunction with the HBASE_CACHE_BLOCKS query option.

Type: BOOLEAN

Default: FALSE (0)

IDLE_SESSION_TIMEOUT query option

Specify the time in seconds after which an idle session is cancelled. A session is idle when no activity is occurring for any of the queries in that session, and the session has not started any new queries. Once a session is expired, you cannot issue any new query requests to it. The session remains open, but the only operation you can perform is to close it.

The IDLE_SESSION_TIMEOUT query option overrides the ##idle_session_timeout startup option. See *Setting timeout periods for daemons, queries, and sessions* for the ##idle_session_timeout startup option.

The IDLE_SESSION_TIMEOUT query option allows JDBC/ODBC connections to set the session timeout as a query option with the SET statement.

Type: Number

Default: 0

When this option is set to default:

- If ##idle_session_timeout is not set, the session never expires.
- If ##idle_session_timeout is set, use that timeout value.

KUDU_READ_MODE query option

The KUDU_READ_MODE query option allows you to set a desired consistency level for scans of Kudu tables.

The following values are supported for the query option:

- "DEFAULT": The value of the startup flag, --kudu_read_mode, is used.
- "READ_LATEST": Commonly known as the Read Committed isolation mode, in this mode, Kudu provides no consistency guarantees for this mode, except that all returned rows were committed at some point.
- "READ_AT_SNAPSHOT": Kudu will take a snapshot of the current state of the data and perform the scan over the snapshot, possibly after briefly waiting for ongoing writes to complete. This provides "Read Your Writes" consistency within a single Impala session, except in the case of a Kudu leader change. See the Kudu documentation for more details.

Type: String

Default: "DEFAULT"

Added in: Impala 3.1

LIVE_PROGRESS query option

When the LIVE_PROGRESS query option is set to TRUE, Impala displays an interactive progress bar showing roughly what percentage of processing has been completed for queries submitted through the `impala-shell` command. When the query finishes, the progress bar is erased from the `impala-shell` console output.

You can enable this query option within `impala-shell` by starting the shell with the --live_progress command-line option. You can still turn this setting off and on again within the shell through the SET command.

Starting in Impala 3.1, the summary output also includes the queuing status consisting of whether the query was queued and what was the latest queuing reason.

Type: BOOLEAN

Default: FALSE (0)

Usage notes:

The output from this query option is printed to standard error. The output is only displayed in interactive mode, that is, not when the -q or -f options are used.

Restrictions:

Because the percentage complete figure is calculated using the number of issued and completed "scan ranges", which occur while reading the table data, the progress bar might reach 100% before the query is entirely finished. For example, the query might do work to perform aggregations after all the table data has been read. If many of your queries fall into this category, consider using the LIVE_SUMMARY option instead for more granular progress reporting.

The LIVE_PROGRESS and LIVE_SUMMARY query options currently do not produce any output during COMPUTE STATS operations.

Because the LIVE_PROGRESS query option is available only within the `impala-shell` interpreter, you cannot change the query option through the SQL SET statement using the JDBC or ODBC interfaces. The SET command in `impala-shell` recognizes the name as a shell-only option.

LIVE_SUMMARY query option

When the LIVE_SUMMARY query option is set to TRUE, Impala displays the same output as the SUMMARY command for queries submitted through the `impala-shell` command, with the measurements updated in real time as the query progresses. When the query finishes, the final SUMMARY output remains visible in the `impala-shell` console output.

You can enable this query option within `impala-shell` by starting the shell with the --live_summary command-line option. You can still turn this setting off and on again within the shell through the SET command.

Starting in Impala 3.1, the summary output also includes the queuing status consisting of whether the query was queued and what was the latest queuing reason.

Type: Boolean

Default: FALSE (0)

Usage notes:

The output from this query option is printed to standard error. The output is only displayed in interactive mode, that is, not when the -q or -f options are used.

Only a single report is displayed at any one time, with each update overwriting the previous numbers.

The live summary output can be useful for evaluating long-running queries, to evaluate which phase of execution takes up the most time, or if some hosts take much longer than others for certain operations, dragging overall performance down. By making the information available in real time, this feature lets you decide what action to take even before you cancel a query that is taking much longer than normal.

For example, you might see the HDFS scan phase taking a long time, and therefore revisit performance-related aspects of your schema design such as constructing a partitioned table, switching to the Parquet file format, running the COMPUTE STATS statement for the table, and so on. Or you might see a wide variation between the average and maximum times for all hosts to perform some phase of the query, and therefore investigate if one particular host needed more memory or was experiencing a network problem.

Restrictions:

The LIVE_PROGRESS and LIVE_SUMMARY query options currently do not produce any output during COMPUTE STATS operations.

Because the LIVE_SUMMARY query option is available only within the `impala-shell` interpreter, you cannot change the query option through the SQL SET statement using the JDBC or ODBC interfaces. The SET command in `impala-shell` recognizes the name as a shell-only option.

MAX_ERRORS query option

Maximum number of non-fatal errors for any particular query that are recorded in the Impala log file. For example, if a billion-row table had a non-fatal data error in every row, you could diagnose the problem without all billion errors being logged. Unspecified or 0 indicates the built-in default value of 1000.

This option only controls how many errors are reported. To specify whether Impala continues or halts when it encounters such errors, use the ABORT_ON_ERROR option.

Type: Number

Default: 0 (meaning 1000 errors)

MAX_MEM_ESTIMATE_FOR_ADMISSION query option

Use the MAX_MEM_ESTIMATE_FOR_ADMISSION query option to set an upper limit on the memory estimates of a query as a workaround for over-estimates precluding a query from being admitted.

The query option takes effect when all of the below conditions are met:

- Memory-based admission control is enabled for the pool.
- The MEM_LIMIT query option is not set at the query, session, resource pool, or global level.

When the above conditions are met, MIN(MAX_MEM_ESTIMATE_FOR_ADMISSION, mem_estimate) is used for admission control.

Setting the MEM_LIMIT query option is usually a better option. Use the MAX_MEM_ESTIMATE _FOR_ADMISSION query option when it is not feasible to set MEM_LIMIT for each individual query.

Type: INT

Added in: Impala 3.1

MAX_NUM_RUNTIME_FILTERS query option

The MAX_NUM_RUNTIME_FILTERS query option sets an upper limit on the number of runtime filters that can be produced for each query.

Type: INT

Default: 10

Usage notes:

Each runtime filter imposes some memory overhead on the query. Depending on the setting of the RUNTIME_BLOOM_FILTER_SIZE query option, each filter might consume between 1 and 16 megabytes per plan fragment. There are typically 5 or fewer filters per plan fragment.

Impala evaluates the effectiveness of each filter, and keeps the ones that eliminate the largest number of partitions or rows. Therefore, this setting can protect against potential problems due to excessive memory overhead for filter production, while still allowing a high level of optimization for suitable queries.

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Kudu consideration:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

MAX_RESULT_SPOOLING_MEM query option

Use the MAX_RESULT_SPOOLING_MEM query option to set the maximum amount of memory used when spooling query results.

If the amount of memory exceeds this value when spooling query results, all memory will most likely be spilled to disk.

The MAX_RESULT_SPOOLING_MEM query option is applicable only when query result spooling is enabled with the SPOOL_QUERY_RESULTS query option set to TRUE.

Setting the option to 0 or -1 means the memory is unbounded.

You cannot set this query option to values below -1.

Type: INT

Default: 100 * 1024 * 1024 (100 MB)

Added in: Impala 3.4

MAX_ROW_SIZE query option

Ensures that Impala can process rows of at least the specified size. (Larger rows might be successfully processed, but that is not guaranteed.) Applies when constructing intermediate or final rows in the result set. This setting prevents out-of-control memory use when accessing columns containing huge strings.

A numeric argument represents a size in bytes; you can also use a suffix of m or mb for megabytes, or g or gb for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Type: INT

Default: 524288 (512 KB)

Usage notes:

If a query fails because it involves rows with long strings and/or many columns, causing the total row size to exceed MAX_ROW_SIZE bytes, increase the MAX_ROW_SIZE setting to accommodate the total bytes stored in the largest row. Examine the error messages for any failed queries to see the size of the row that caused the problem.

Impala attempts to handle rows that exceed the MAX_ROW_SIZE value where practical, so in many cases, queries succeed despite having rows that are larger than this setting.

Specifying a value that is substantially higher than actually needed can cause Impala to reserve more memory than is necessary to execute the query.

In a Hadoop cluster with highly concurrent workloads and queries that process high volumes of data, traditional SQL tuning advice about minimizing wasted memory is worth remembering. For example, if a table has STRING columns where a single value might be multiple megabytes, make sure that the SELECT lists in queries only refer to columns that are actually needed in the result set, instead of using the SELECT * shorthand.

MAX_SCAN_RANGE_LENGTH query option

Maximum length of the scan range. Interacts with the number of HDFS blocks in the table to determine how many CPU cores across the cluster are involved with the processing for a query. (Each core processes one scan range.)

Lowering the value can sometimes increase parallelism if you have unused CPU capacity, but a too-small value can limit query performance because each scan range involves extra overhead.

Only applicable to HDFS tables. Has no effect on Parquet tables. Unspecified or 0 indicates backend default, which is the same as the HDFS block size for each table.

Although the scan range can be arbitrarily long, Impala internally uses an 8 MB read buffer so that it can query tables with huge block sizes without allocating equivalent blocks of memory.

The value can include unit specifiers, such as 100m or 100mb.

Type: Number

Default: 0

MAX_SPILLED_RESULT_SPOOLING_MEM query option

Use the MAX_SPILLED_RESULT_SPOOLING_MEM query option to set the maximum amount of memory that can be spilled when spooling query results.

If the amount of memory exceeds this value when spooling query results, the coordinator fragment will block until the client has consumed enough rows to free up more memory.

The MAX_SPILLED_RESULT_SPOOLING_MEM query option is applicable only when query result spooling is enabled with the SPOOL_QUERY_RESULTS query option set to TRUE.

The value must be greater than or equal to the value of MAX_RESULT_SPOOLING_MEM.

Setting the option to 0 or -1 means the memory is unbounded.

Values below -1 are not allowed for this query option.

Type: INT

Default: 1024 * 1024 * 1024 (1 GB)

Added in: Impala 3.4

MEM_LIMIT query option

The MEM_LIMIT query option defines the maximum amount of memory a query can allocate on each node. The total memory that can be used by a query is the MEM_LIMIT times the number of nodes.

There are two levels of memory limit for Impala. The ##mem_limit startup option sets an overall limit for the `impalad` process (which handles multiple queries concurrently). That process memory limit can be expressed either as a percentage of RAM available to the process such as -mem_limit=70% or as a fixed amount of memory, such as ##mem_limit=100gb. The memory available to the process is based on the host's physical memory and memory limits from Linux Control Groups. For example, if an `impalad` process is running in a Docker container on a host with 100 GB of memory, the memory available is 100 GB or the Docker container's memory limit, whichever is less.

The MEM_LIMIT query option, which you set through `impala-shell` or the SET statement in a JDBC or ODBC application, applies to each individual query. The MEM_LIMIT query option is usually expressed as a fixed size such as 10gb, and must always be less than the `impalad` memory limit.

If query processing approaches the specified memory limit on any node, either the per-query limit or the `impalad` limit, then the SQL operations will start to reduce their memory consumption, for example by writing the temporary data to disk (known as spilling to disk). The result is a query that completes successfully, rather than failing with an out-of-memory error. The tradeoff is decreased performance due to the extra disk I/O to write the temporary data and read it back in. The slowdown

could potentially be significant. Thus, while this feature improves reliability, you should optimize your queries, system parameters, and hardware configuration to make this spilling a rare occurrence.

Type: Number

Units: A numeric argument represents memory size in bytes; you can also use a suffix of m or mb for megabytes, or more commonly g or gb for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Default: 0 (unlimited)

Usage notes:

The MEM_LIMIT setting is primarily useful for production workloads. Impala's Admission Controller can be configured to automatically assign memory limits to queries and limit memory consumption of resource pools. See Concurrent Queries and Admission Control. and Memory Limits and Admission Control for more information on configuring the resource usage through admission control.

Use the output of the SUMMARY command in `impala-shell` to get a report of memory used for each phase of your most heavyweight queries on each node, and then set a MEM_LIMIT somewhat higher than that. See Understanding Performance using SUMMARY Report for usage information about the SUMMARY command.

The following examples show how to set the MEM_LIMIT query option using a fixed number of bytes, or suffixes representing gigabytes or megabytes.

```
[localhost:21000] > set mem_limit=3000000000;
MEM_LIMIT set to 3000000000
[localhost:21000] > select 5;
Query: select 5
+---+
| 5 |
+---+
| 5 |
+---+

[localhost:21000] > set mem_limit=3g;
MEM_LIMIT set to 3g
[localhost:21000] > select 5;
Query: select 5
+---+
| 5 |
+---+
| 5 |
+---+

[localhost:21000] > set mem_limit=3gb;
MEM_LIMIT set to 3gb
[localhost:21000] > select 5;
+---+
| 5 |
+---+
| 5 |
+---+

[localhost:21000] > set mem_limit=3m;
MEM_LIMIT set to 3m
[localhost:21000] > select 5;
+---+
| 5 |
+---+
| 5 |
+---+
```

```
[localhost:21000] > set mem_limit=3mb;
MEM_LIMIT set to 3mb
[localhost:21000] > select 5;
+---+
| 5 |
+---+
```

The following examples show how unrecognized MEM_LIMIT values lead to errors for subsequent queries.

```
[localhost:21000] > set mem_limit=3pb;
MEM_LIMIT set to 3pb
[localhost:21000] > select 5;
ERROR: Failed to parse query memory limit from '3pb'.

[localhost:21000] > set mem_limit=xyz;
MEM_LIMIT set to xyz
localhost:21000] > select 5;
Query: select 5
ERROR: Failed to parse query memory limit from 'xyz'.
```

The following examples shows the automatic query cancellation when the MEM_LIMIT value is exceeded on any host involved in the Impala query. First it runs a successful query and checks the largest amount of memory used on any node for any stage of the query. Then it sets an artificially low MEM_LIMIT setting so that the same query cannot run.

```
[localhost:21000] > select count(*) from customer;
Query: select count(*) from customer
+----------+
| count(*) |
+----------+
| 150000   |
+----------+

[localhost:21000] > select count(distinct c_name) from customer;
Query: select count(distinct c_name) from customer
+------------------------+
| count(distinct c_name) |
+------------------------+
| 150000                 |
+------------------------+

[localhost:21000] > summary;
+--------------+--------+--------+----------+----------+---------
+------------+----------+--------------+--------------+
| Operator     | #Hosts | #Inst  | Avg Time | Max Time | #Rows
 | Est. #Rows | Peak Mem | Est. Peak Mem | Detail        |
+--------------+--------+--------+----------+----------+---------
+------------+----------+--------------+--------------+
| 06:AGGREGATE | 1      | 1      | 230.00ms | 230.00ms | 1
 | 1          | 16.00 KB | -1 B          | FINALIZE      |
| 05:EXCHANGE  | 1      | 1      | 43.44us  | 43.44us  | 1
| 1          | 0 B      | -1 B          | UNPARTITIONED |
| 02:AGGREGATE | 1      | 1      | 227.14ms | 227.14ms | 1
 | 1          | 12.00 KB | 10.00 MB      |               |
| 04:AGGREGATE | 1      | 1      | 126.27ms | 126.27ms | 150.00K
 | 150.00K    | 15.17 MB | 10.00 MB      |               |
| 03:EXCHANGE  | 1      | 1      | 44.07ms  | 44.07ms  | 150.00K
 | 150.00K    | 0 B      | 0 B           | HASH(c_name)  |
| 01:AGGREGATE | 1      | 1      | 361.94ms | 361.94ms | 150.00K
 | 150.00K    | 23.04 MB | 10.00 MB      |               |
```

```
| 00:SCAN HDFS | 1      | 1      | 43.64ms  | 43.64ms  | 150.00K
 | 150.00K    | 24.19 MB | 64.00 MB      | tpch.customer |
+--------------+--------+--------+----------+----------+---------
+------------+----------+--------------+--------------+

[localhost:21000] > set mem_limit=15mb;
MEM_LIMIT set to 15mb
[localhost:21000] > select count(distinct c_name) from customer;
Query: select count(distinct c_name) from customer
ERROR:
Rejected query from pool default-pool: minimum memory reservation
 is greater than memory available to the query
for buffer reservations. Memory reservation needed given the curr
ent plan: 38.00 MB. Adjust either the mem_limit
or the pool config (max-query-mem-limit, min-query-mem-limit) for
 the query to allow the query memory limit to be
at least 70.00 MB. Note that changing the mem_limit may also c
hange the plan. See the query profile for more
information about the per-node memory requirements.
```

MIN_SPILLABLE_BUFFER_SIZE query option

Specifies the minimum size for a memory buffer used when the spill-to-disk mechanism is activated, for example for queries against a large table with no statistics, or large join operations.

Type: INT

Default: 65536 (64 KB)

Units: A numeric argument represents memory size in bytes; you can also use a suffix of m or mb for megabytes, or more commonly g or gb for gigabytes. If you specify a value with unrecognized formats, subsequent queries fail with an error.

Usage notes:

This query option sets a lower bound on the size of the internal buffer size that can be used during spill-to-disk operations. The actual size of the buffer is chosen by the query planner.

If overall query performance is limited by the time needed for spilling, consider increasing the MIN_SPILLABLE_BUFFER_SIZE setting. Larger buffer sizes result in Impala issuing larger I/O requests to storage devices, which might result in higher throughput, particularly on rotational disks.

The tradeoff with a large value for this setting is increased memory usage during spill-to-disk operations. Reducing this value may reduce memory consumption.

To determine if the value for this setting is having an effect by capping the spillable buffer size, you can see the buffer size chosen by the query planner for a particular query. EXPLAIN the query while the setting EXPLAIN_LEVEL=2 is in effect.

MT_DOP query option

Sets the degree of parallelism used for certain operations that can benefit from multithreaded execution. You can specify values higher than zero to find the ideal balance of response time, memory usage, and CPU usage during statement processing.

Note:

The Impala execution engine is being revamped incrementally to add additional parallelism within a single host for certain statements and kinds of operations. The setting MT_DOP=0 uses the “old” code path with limited intra-node parallelism.

Currently, the operations affected by the MT_DOP query option are:

- COMPUTE [INCREMENTAL] STATS. Impala automatically sets MT_DOP=4 for COMPUTE STATS and COMPUTE INCREMENTAL STATS statements on Parquet tables.
- Queries with execution plans containing only scan and aggregation operators. Other queries produce an error if MT_DOP is set to a non-zero value. Therefore, this query option is typically only set for the duration of specific long-running, CPU-intensive queries.

Type: INT

Default: 0

Because COMPUTE STATS and COMPUTE INCREMENTAL STATS statements for Parquet tables benefit substantially from extra intra-node parallelism, Impala automatically sets MT_DOP=4 when computing stats for Parquet tables.

Range: 0 to 64

NUM_NODES query option

Limit the number of nodes that process a query, typically during debugging.

Type: Number

Allowed values: Only accepts the values 0 (meaning all nodes) or 1 (meaning all work is done on the coordinator node).

Default: 0

Usage notes:

If you are diagnosing a problem that you suspect is due to a timing issue due to distributed query processing, you can set NUM_NODES=1 to verify if the problem still occurs when all the work is done on a single node.

You might set the NUM_NODES option to 1 briefly, during INSERT or CREATE TABLE AS SELECT statements. Normally, those statements produce one or more data files per data node. If the write operation involves small amounts of data, a Parquet table, and/or a partitioned table, the default behavior could produce many small files when intuitively you might expect only a single output file. SET NUM_NODES=1 turns off the “distributed” aspect of the write operation, making it more likely to produce only one or a few data files.

Warning:

Because this option results in increased resource utilization on a single host, it could cause problems due to contention with other Impala statements or high resource usage. Symptoms could include queries running slowly, exceeding the memory limit, or appearing to stop responding. Use it only in a single-user development/test environment; do not use it in a production environment or in a cluster with a high-concurrency or high-volume or performance-critical workload.

NUM_ROWS_PRODUCED_LIMIT query option

The NUM_ROWS_PRODUCED_LIMIT query option limits the number of rows produced by a query. A query is canceled when its execution produces more rows than the specified limit set by the NUM_ROWS_PRODUCED_LIMIT option.

This limit only applies when the results are returned to a client, for example, to a SELECT query, but not to an INSERT query.

The default value of 0 specifies that there is no limit on the number of rows produced.

Type: INT

Allowed values: 0 or positive numbers

Default: 0

NUM_SCANNER_THREADS query option

Maximum number of scanner threads (on each node) used for each query. By default, Impala uses as many cores as are available (one thread per core). You might lower this value if queries are using excessive resources on a busy cluster. Impala imposes a maximum value automatically, so a high value has no practical effect.

Type: Number

Default: 0

OPTIMIZE_PARTITION_KEY_SCANS query option

Enables a fast code path for queries that apply simple aggregate functions to partition key columns: MIN(*key_column*), MAX(*key_column*), or COUNT(DISTINCT *key_column*).

Type: BOOLEAN

Default: FALSE (0)

Usage notes:

This optimization speeds up common “introspection” operations over partition key columns, for example determining the distinct values of partition keys.

This optimization does not apply to SELECT statements that reference columns that are not partition keys. It also only applies when all the partition key columns in the SELECT statement are referenced in one of the following contexts:

- Within a MAX() or MAX() aggregate function or as the argument of any aggregate function with the DISTINCT keyword applied.
- Within a WHERE, GROUP BY or HAVING clause.

This optimization is enabled by a query option because it skips some consistency checks and therefore can return slightly different partition values if partitions are in the process of being added, dropped, or loaded outside of Impala. Queries might exhibit different behavior depending on the setting of this option in the following cases:

- If files are removed from a partition using HDFS or other non-Impala operations, there is a period until the next REFRESH of the table where regular queries fail at run time because they detect the missing files. With this optimization enabled, queries that evaluate only the partition key column values (not the contents of the partition itself) succeed, and treat the partition as if it still exists.
- If a partition contains any data files, but the data files do not contain any rows, a regular query considers that the partition does not exist. With this optimization enabled, the partition is treated as if it exists.

  If the partition includes no files at all, this optimization does not change the query behavior: the partition is considered to not exist whether or not this optimization is enabled.

PARQUET_COMPRESSION_CODEC query option

Deprecated. Use COMPRESSION_CODEC in Impala 2.0 and later.

PARQUET_ANNOTATE_STRINGS_UTF8 query option

Causes Impala INSERT and CREATE TABLE AS SELECT statements to write Parquet files that use the UTF-8 annotation for STRING columns.

Type: BOOLEAN

Default: FALSE (0)

Usage notes:

By default, Impala represents a STRING column in Parquet as an unannotated binary field.

Impala always uses the UTF-8 annotation when writing CHAR and VARCHAR columns to Parquet files. An alternative to using the query option is to cast STRING values to VARCHAR.

This option is to help make Impala-written data more interoperable with other data processing engines. Impala itself currently does not support all operations on UTF-8 data. Although data processed by Impala is typically represented in ASCII, it is valid to designate the data as UTF-8 when storing on disk, because ASCII is a subset of UTF-8.

PARQUET_ARRAY_RESOLUTION query option

The PARQUET_ARRAY_RESOLUTION query option controls the behavior of the indexed-based resolution for nested arrays in Parquet.

In Parquet, you can represent an array using a 2-level or 3-level representation. The modern, standard representation is 3-level. The legacy 2-level scheme is supported for compatibility with older Parquet files. However, there is no reliable metadata within Parquet files to indicate which encoding was used. It is even possible to have mixed encodings within the same file if there are multiple arrays. The PARQUET_ARRAY_RESOLUTION option controls the process of resolution that is to match every column/field reference from a query to a column in the Parquet file.

The supported values for the query option are:

- THREE_LEVEL: Assumes arrays are encoded with the 3-level representation, and does not attempt the 2-level resolution.
- TWO_LEVEL: Assumes arrays are encoded with the 2-level representation, and does not attempt the 3-level resolution.
- TWO_LEVEL_THEN_THREE_LEVEL: First tries to resolve assuming a 2-level representation, and if unsuccessful, tries a 3-level representation.

All of the above options resolve arrays encoded with a single level.

A failure to resolve a column/field reference in a query with a given array resolution policy does not necessarily result in a warning or error returned by the query. A mismatch might be treated like a missing column (returns NULL values), and it is not possible to reliably distinguish the 'bad resolution' and 'legitimately missing column' cases.

The name-based policy generally does not have the problem of ambiguous array representations. You specify to use the name-based policy by setting the PARQUET_FALLBACK_SCHEMA_RESOLUTION query option to NAME.

Type: Enum of TWO_LEVEL, TWO_LEVEL_THEN_THREE_LEVEL, and THREE_LEVEL

Default: THREE_LEVEL

Examples:

EXAMPLE A: The following Parquet schema of a file can be interpreted as a 2-level or 3-level:

```
ParquetSchemaExampleA {
  optional group single_element_groups (LIST) {
    repeated group single_element_group {
      required int64 count;
    }
  }
```

```
}
```

The following table schema corresponds to a 2-level interpretation:

```
CREATE TABLE t (col1 array<struct<f1: bigint>>) STORED AS PARQU
ET;
```

Successful query with a 2-level interpretation:

```
SET PARQUET_ARRAY_RESOLUTION=TWO_LEVEL;
SELECT ITEM.f1 FROM t.col1;
```

The following table schema corresponds to a 3-level interpretation:

```
CREATE TABLE t (col1 array<bigint>) STORED AS PARQUET;
```

Successful query with a 3-level interpretation:

```
SET PARQUET_ARRAY_RESOLUTION=THREE_LEVEL;
SELECT ITEM FROM t.col1
```

EXAMPLE B: The following Parquet schema of a file can be only be successfully interpreted as a 2-level:

```
ParquetSchemaExampleB {
  required group list_of_ints (LIST) {
    repeated int32 list_of_ints_tuple;
  }
}
```

The following table schema corresponds to a 2-level interpretation:

```
CREATE TABLE t (col1 array<int>) STORED AS PARQUET;
```

Successful query with a 2-level interpretation:

```
SET PARQUET_ARRAY_RESOLUTION=TWO_LEVEL;
SELECT ITEM FROM t.col1
```

Unsuccessful query with a 3-level interpretation. The query returns NULLs as if the column was missing in the file:

```
SET PARQUET_ARRAY_RESOLUTION=THREE_LEVEL;
SELECT ITEM FROM t.col1
```

PARQUET_DICTIONARY_FILTERING query option

The PARQUET_DICTIONARY_FILTERING query option controls whether Impala uses dictionary filtering for Parquet files.

To efficiently process a highly selective scan query, when this option is enabled, Impala checks the values in the Parquet dictionary page and determines if the whole row group can be thrown out.

A column chunk is purely dictionary encoded and can be used by dictionary filtering if any of the following conditions are met:

1. If the encoding_stats is in the Parquet file, dictionary filtering uses it to determine if there are only dictionary encoded pages (i.e. there are no data pages with an encoding other than PLAIN_DICTIONARY).
2. If the encoding stats are not present, dictionary filtering looks at the encodings. The column is purely dictionary encoded if both of the conditions satisfy:
   - PLAIN_DICTIONARY is present.
   - Only PLAIN_DICTIONARY, RLE, or BIT_PACKED encodings are listed.
3. Dictionary filtering works for the Parquet dictionaries with less than 40000 values if the file was written by Impala 2.9.

In the query runtime profile output for each Impalad instance, the NumDictFilteredRowGroups field in the SCAN node section shows the number of row groups that were skipped based on dictionary filtering.

Note that row groups can be filtered out by Parquet statistics, and in such cases, dictionary filtering will not be considered.

The supported values for the query option are:

- TRUE (1): Use dictionary filtering.
- FALSE (0): Do not use dictionary filtering
- Any other values are treated as FALSE.

Type: BOOLEAN

Default: TRUE (1)

PARQUET_FALLBACK_SCHEMA_RESOLUTION query option

The PARQUET_FALLBACK_SCHEMA_RESOLUTION query option allows Impala to look up columns within Parquet files by column name, rather than column order, when necessary. The allowed values are:

- POSITION (0)
- NAME (1)

Type: Enum of the above allowed values

Usage notes:

By default, Impala looks up columns within a Parquet file based on the order of columns in the table. The name setting for this option enables behavior for Impala queries similar to the Hive setting parquet.column.index access=false. It also allows Impala to query Parquet files created by Hive with the parquet.column.index.access=false setting in effect.

PARQUET_FILE_SIZE query option

Specifies the maximum size of each Parquet data file produced by Impala INSERT statements in bytes, or with a trailing m or g character to indicate megabytes or gigabytes.

With tables that are small or finely partitioned, the default Parquet block size (formerly 1 GB, now 256 MB in Impala 2.0 and later) could be much larger than needed for each data file. For INSERT operations into such tables, you can increase parallelism by specifying a smaller PARQUET_FILE _SIZE value, resulting in more HDFS blocks that can be processed by different nodes.

Type: Number, with optional unit specifier

Important:

Currently, the maximum value for this setting is 1 gigabyte (1g). Setting a value higher than 1 gigabyte could result in errors during an INSERT operation.

Default: 0 (produces files with a target size of 256 MB; files might be larger for very wide tables)

PARQUET_OBJECT_STORE_SPLIT_SIZE

Use the PARQUET_OBJECT_STORE_SPLIT_SIZE query option to control the Parquet split sizes on non-block based stores, e.g. S3, ADLS, etc.

Type: INT

Allowed values: The value must be greater than or equal to 1 MB.

Default: 256 MB

Added in: Impala 3.4 / CDP 1.2

PARQUET_PAGE_ROW_COUNT_LIMIT query option

Use the PARQUET_PAGE_ROW_COUNT_LIMIT query option to set the maximum number of rows that can be written on a single Parquet data page. By default there is no row count limit.

Type: INT

Allowed values: Positive integers

Added in: Impala 3.3

PARQUET_READ_PAGE_INDEX query option

Use the PARQUET_READ_PAGE_INDEX query option to disable or enable using the Parquet page index during scans. The page index contains min/max statistics at the page-level granularity. It can be used to skip pages and rows that do not match the conditions in the WHERE clause.

This option enables the same optimization as the PARQUET_READ_STATISTICS at the finer grained page level.

Impala supports filtering based on Parquet statistics:

- Of the types: BOOLEAN, INT, DECIMAL, STRING, TIMESTAMP, DATE
- For simple predicates of the forms: <slot> <op> <constant> or <constant> <op> <slot>, where <op> is LT, LE, GE, GT, and EQ

The supported values for the query option are:

- TRUE (1): Read the page-level statistics from the Parquet page index during query processing and filter out pages based on the statistics.
- FALSE (0): Do not use the Parquet page index.
- Any other values are treated as false.

Type: BOOLEAN

Default: TRUE (1)

PARQUET_READ_STATISTICS query option

The PARQUET_READ_STATISTICS query option controls whether to read statistics from Parquet files and use them during query processing.

Parquet stores min/max stats which can be used to skip reading row groups if they don't qualify a certain predicate. When this query option is set to true, Impala reads the Parquet statistics and skips reading row groups that do not match the conditions in the WHERE clause.

Impala supports filtering based on Parquet statistics:

- Of the numerical types for the old version of the statistics: BOOLEAN, INTEGER, FLOAT
- Of the types for the new version of the statistics (starting in impala 2.8): BOOLEAN, INTE GER, FLOAT, DECIMAL, STRING, TIMESTAMP, DATE
- For simple predicates of the forms: <slot> <op> <constant> or <constant> <op> <slot>, where <op> is LT, LE, GE, GT, and EQ

The PARQUET_READ_STATISTICS option provides a workaround when dealing with files that have corrupt Parquet statistics and unknown errors.

In the query runtime profile output for each Impalad instance, the NumStatsFilteredRowGroups field in the SCAN node section shows the number of row groups that were skipped based on Parquet statistics.

The supported values for the query option are:

- TRUE (1): Read statistics from Parquet files and use them in query processing.
- FALSE (0): Do not use Parquet read statistics.
- Any other values are treated as FALSE.

Type: BOOLEAN

Default: TRUE

PARQUET_WRITE_PAGE_INDEX query option

The PARQUET_WRITE_PAGE_INDEX query option disables or enables the Parquet page index writing.

Impala writes page-level statistics into the Parquet page index of the types: INT, DECIMAL, STRI NG, TIMESTAMP, DATE

The supported values for the query option are:

- TRUE (1): Write the Parquet page index when creating Parquet files.
- FALSE (0): Do not write the Parquet page index when creating Parquet files.
- Any other values are treated as false.

Type: BOOLEAN

Default: TRUE (1)

Added in: Impala 3.3

PREFETCH_MODE query option

Determines whether the prefetching optimization is applied during join query processing.

Allowed values are:

- NONE (0)
- HT_BUCKET (1): Hash table buckets are prefetched during join query processing.

Type: Enum of the above allowed values

Default: 1 (HT_BUCKET)

QUERY_TIMEOUT_S query option

Sets the idle query timeout value for the session, in seconds. Queries that sit idle for longer than the timeout value are automatically cancelled. If the system administrator specified the --idle_query _timeout startup option, QUERY_TIMEOUT_S must be smaller than or equal to the --idle_query _timeout value.

Note:

The timeout clock for queries and sessions only starts ticking when the query or session is idle.

For queries, this means the query has results ready but is waiting for a client to fetch the data. A query can run for an arbitrary time without triggering a timeout, because the query is computing results rather than sitting idle waiting for the results to be fetched. The timeout period is intended to prevent unclosed queries from consuming resources and taking up slots in the admission count of running queries, potentially preventing other queries from starting.

For sessions, this means that no query has been submitted for some period of time.

Type: Number

Default: 0 (no timeout if --idle_query_timeout not in effect; otherwise, use --idle_query_timeout value)

REPLICA_PREFERENCE query option

The REPLICA_PREFERENCE query option lets you distribute the work more evenly if hotspots and bottlenecks persist. It causes the access cost of all replicas of a data block to be considered equal to or worse than the configured value. This allows Impala to schedule reads to suboptimal replicas (e.g. local in the presence of cached ones) in order to distribute the work across more executor nodes.

Allowed values are:

- CACHE_LOCAL (0)
- DISK_LOCAL (2)
- REMOTE (4)

Type: Enum

Default: CACHE_LOCAL (0)

Usage Notes:

By default Impala selects the best replica it can find in terms of access cost. The preferred order is cached, local, and remote. With REPLICA_PREFERENCE, the preference of all replicas are capped at the selected value. For example, when REPLICA_PREFERENCE is set to DISK_LOC AL, cached and local replicas are treated with the equal preference. When set to REMOTE, all three types of replicas, cached, local, remote, are treated with equal preference.

REQUEST_POOL query option

The pool or queue name that queries should be submitted to. Only applies when you enable the Impala admission control feature. Specifies the name of the pool used by requests from Impala to the resource manager.

Type: STRING

Default: An empty string (use the user-to-pool mapping defined by an `impalad` startup option in the Impala configuration file)

RESOURCE_TRACE_RATIO query option

The RESOURCE_TRACE_RATIO query option specifies the ratio of queries where the CPU usage info will be included in the profiles. Collecting CPU usage and sending it around adds a slight overhead during query execution. This query option lets you control whether to collect additional information to diagnose the resource usage.

For example, setting RESOURCE_TRACE_RATIO=1 adds a trace of the CPU usage to the profile of each query.

Setting RESOURCE_TRACE_RATIO=0.5 means that randomly selected half of all queries will have that information collected by the coordinator and included in the profiles.

Setting RESOURCE_TRACE_RATIO=0 means that CPU usage will not be tracked and included in the profiles.

Values from 0 to 1 are allowed.

Type: Number

Default: 0

Added in: Impala 3.2

RUNTIME_BLOOM_FILTER_SIZE query option

Size (in bytes) of Bloom filter data structure used by the runtime filtering feature.

Important:

In Impala 2.6 and higher, this query option only applies as a fallback, when statistics are not available. By default, Impala estimates the optimal size of the Bloom filter structure regardless of the setting for this option. (This is a change from the original behavior in Impala 2.5.)

In Impala 2.6 and higher, when the value of this query option is used for query planning, it is constrained by the minimum and maximum sizes specified by the RUNTIME_FILTER_MIN_SIZE and RUNTIME_FILTER_MAX_SIZE query options. The filter size is adjusted upward or downward if necessary to fit within the minimum/maximum range.

Type: INTEGER

Default: 1048576 (1 MB)

Maximum: 16 MB

Usage notes:

This setting affects optimizations for large and complex queries, such as dynamic partition pruning for partitioned tables, and join optimization for queries that join large tables. Larger filters are more effective at handling higher cardinality input sets, but consume more memory per filter.

If your query filters on high-cardinality columns (for example, millions of different values) and you do not get the expected speedup from the runtime filtering mechanism, consider doing some benchmarks with a higher value for RUNTIME_BLOOM_FILTER_SIZE. The extra memory devoted to the Bloom filter data structures can help make the filtering more accurate.

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Because the effectiveness of this setting depends so much on query characteristics and data distribution, you typically only use it for specific queries that need some extra tuning, and the ideal value depends on the query. Consider setting this query option immediately before the expensive query and unsetting it immediately afterward.

Kudu consideration:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

RUNTIME_FILTER_MAX_SIZE query option

The RUNTIME_FILTER_MAX_SIZE query option adjusts the settings for the runtime filtering feature. This option defines the maximum size for a filter, no matter what the estimates produced

by the planner are. This value also overrides any lower number specified for the RUNTIME_BLOOM_FILTER_SIZE query option. Filter sizes are rounded up to the nearest power of two.

Type: INTEGER

Default: 1048576 (1 MB)

Usage notes:

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Kudu consideration:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

RUNTIME_FILTER_MIN_SIZE query option

The RUNTIME_FILTER_MIN_SIZE query option adjusts the settings for the runtime filtering feature. This option defines the minimum size for a filter, no matter what the estimates produced by the planner are. This value also overrides any lower number specified for the RUNTIME_BLOOM_FILTER_SIZE query option. Filter sizes are rounded up to the nearest power of two.

Type: INTEGER

Default: 0 (meaning use the value from the corresponding `impalad` startup option)

Usage notes:

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables.

Kudu consideration:

This query option affects only Bloom filters, not the min/max filters that are applied to Kudu tables. Therefore, it does not affect the performance of queries against Kudu tables.

RUNTIME_FILTER_MODE query option

The RUNTIME_FILTER_MODE query option adjusts the settings for the runtime filtering feature. It turns this feature on and off, and controls how extensively the filters are transmitted between hosts.

Allowed values are:

- OFF (0)
- LOCAL (1)
- GLOBAL (2)

Type: Enum of the above allowed values

Default: GLOBAL (2)

Usage notes:

The default is GLOBAL setting is recommended for a wide variety of workloads, to provide best performance with “out of the box” settings.

The lowest setting of LOCAL does a similar level of optimization (such as partition pruning) as in earlier Impala releases. This setting was the default in Impala 2.5, to allow for a period of post-upgrade testing for existing workloads. This setting is suitable for workloads with non-performance-critical queries, or if the coordinator node is under heavy CPU or memory pressure.

You might change the setting to OFF if your workload contains many queries involving partitioned tables or joins that do not experience a performance increase from the runtime filters feature. If the

overhead of producing the runtime filters outweighs the performance benefit for queries, you can turn the feature off entirely.

RUNTIME_FILTER_WAIT_TIME_MS query option

The RUNTIME_FILTER_WAIT_TIME_MS query option adjusts the settings for the runtime filtering feature. It specifies a time in milliseconds that each scan node waits for runtime filters to be produced by other plan fragments.

Type: INTEGER

Default: 0 (meaning use the value from the corresponding `impalad` startup option)

Usage notes:

Because the runtime filtering feature applies mainly to resource-intensive and long-running queries, only adjust this query option when tuning long-running queries involving some combination of large partitioned tables and joins involving large tables

S3_SKIP_INSERT_STAGING query option

Speeds up INSERT operations on tables or partitions residing on the Amazon S3 filesystem. The tradeoff is the possibility of inconsistent data left behind if an error occurs partway through the operation.

By default, Impala write operations to S3 tables and partitions involve a two-stage process. Impala writes intermediate files to S3, then (because S3 does not provide a “rename” operation) those intermediate files are copied to their final location, making the process more expensive as on a filesystem that supports renaming or moving files. This query option makes Impala skip the intermediate files, and instead write the new data directly to the final destination.

Type: BOOLEAN

Default: TRUE (1)

Usage notes:

Important:

If a host that is participating in the INSERT operation fails partway through the query, you might be left with a table or partition that contains some but not all of the expected data files. Therefore, this option is most appropriate for a development or test environment where you have the ability to reconstruct the table if a problem during INSERT leaves the data in an inconsistent state.

The timing of file deletion during an INSERT OVERWRITE operation makes it impractical to write new files to S3 and delete the old files in a single operation. Therefore, this query option only affects regular INSERT statements that add to the existing data in a table, not INSERT OVERWRITE statements. Use TRUNCATE TABLE if you need to remove all contents from an S3 table before performing a fast INSERT with this option enabled.

Performance improvements with this option enabled can be substantial. The speed increase might be more noticeable for non-partitioned tables than for partitioned tables.

SCAN_BYTES_LIMIT query option

The SCAN_BYTES_LIMIT query option sets a limit on the bytes scanned by HDFS and HBase SCAN operations. If a query is still executing when the query’s coordinator detects that it has exceeded the limit, the query is terminated with an error. The option is intended to prevent runaway queries that scan more data than is intended.

For example, an Impala administrator could set a default value of SCAN_BYTES_LIMIT=100GB for a resource pool to automatically kill queries that scan more than 100 GB of data (see Impala Admission Control and Query Queuing for information about default query options). If a user accidentally omits a partition filter in a WHERE clause and runs a large query that scans a lot of

data, the query will be automatically terminated after it scans more data than the SCAN_BYTES_L IMIT.

You can override the default value per-query or per-session, in the same way as other query options, if you do not want the default SCAN_BYTES_LIMIT value to apply to a specific query or session.

Note:

- Only data actually read from the underlying storage layer is counted towards the limit. E.g. Impala's Parquet scanner employs several techniques to skip over data in a file that is not relevant to a specific query, so often only a fraction of the file size is counted towards SCAN_BYTES_LIMIT.
- As of Impala 3.1, bytes scanned by Kudu tablet servers are not counted towards the limit.

Because the checks are done periodically, the query may scan over the limit at times.

Type: Number

Units:

- A numeric argument represents memory size in bytes.
- Specify a suffix of m or mb for megabytes.
- Specify a suffix of g or gb for gigabytes.
- If you specify a suffix with unrecognized formats, subsequent queries fail with an error.

Default: 0 (no limit)

Added in: Impala 3.1

SCHEDULE_RANDOM_REPLICA query option

The SCHEDULE_RANDOM_REPLICA query option fine-tunes the scheduling algorithm for deciding which host processes each HDFS data block or Kudu tablet to reduce the chance of CPU hotspots.

By default, Impala estimates how much work each host has done for the query, and selects the host that has the lowest workload. This algorithm is intended to reduce CPU hotspots arising when the same host is selected to process multiple data blocks / tablets. Use the SCHEDULE_RANDOM_ REPLICA query option if hotspots still arise for some combinations of queries and data layout.

The SCHEDULE_RANDOM_REPLICA query option only applies to tables and partitions that are not enabled for the HDFS caching.

Type: BOOLEAN

Default: FALSE

SCRATCH_LIMIT query option

Specifies the maximum amount of disk storage, in bytes, that any Impala query can consume on any host using the "spill to disk" mechanism that handles queries that exceed the memory limit.

Specify the size in bytes, or with a trailing m or g character to indicate megabytes or gigabytes.

A value of zero turns off the spill to disk feature for queries in the current session, causing them to fail immediately if they exceed the memory limit.

The amount of memory used per host for a query is limited by the MEM_LIMIT query option.

The more DataNodes in the cluster, the less memory is used on each host, and therefore also less scratch space is required for queries that exceed the memory limit.

Type: Number, with optional unit specifier

Default: -1 (amount of spill space is unlimited)

SHUFFLE_DISTINCT_EXPRS query option

The SHUFFLE_DISTINCT_EXPRS query option controls the shuffling behavior when a query has both grouping and distinct expressions. Impala can optionally include the distinct expressions in the hash exchange to spread the data among more nodes. However, this plan requires one more hash exchange phase.

It is recommended that you turn off this option if the NDVs of the grouping expressions are high.

Type: BOOLEAN

Default: FALSE

SPOOL_QUERY_RESULTS query Option

Use the SPOOL_QUERY_RESULTS query option to enable query result spooling, which is disabled by default.

Query result spooling controls how rows are returned to the client.

- When query result spooling is disabled (SPOOL_QUERY_RESULTS = FALSE), Impala relies on clients to fetch results to trigger the generation of more result row batches until all the result rows have been produced. If a client issues a query without fetching all the results, the query fragments will continue to consume the resources until the query is cancelled and unregistered, potentially tying up resources and cause other queries to wait for extended period of time in admission control.
- When query result spooling is enabled (SPOOL_QUERY_RESULTS = TRUE), the result sets of queries are eagerly fetched and spooled, either in memory or on disk.

  Once all result rows have been fetched and stored in the spooling location, the resources are freed up. Incoming client fetches can get the data from the spooled results.

Type: INT

Default: FALSE

Added in: Impala 3.4

SUPPORT_START_OVER query option

Leave this setting at its default value. It is a read-only setting, tested by some client applications such as Hue.

If you accidentally change it through `impala-shell`, subsequent queries encounter errors until you undo the change by issuing UNSET support_start_over.

Type: BOOLEAN

Default: FALSE

SYNC_DDL query option

When enabled, causes any DDL operation such as CREATE TABLE or ALTER TABLE to return only when the changes have been propagated to all other Impala nodes in the cluster by the Impala catalog service. That way, if you issue a subsequent CONNECT statement in `impala-shell` to connect to a different node in the cluster, you can be sure that other node will already recognize any added or changed tables. (The catalog service automatically broadcasts the DDL changes to all nodes automatically, but without this option there could be a period of inconsistency if you quickly switched to another node, such as by issuing a subsequent query through a load-balancing proxy.)

Although INSERT is classified as a DML statement, when the SYNC_DDL option is enabled, INSERT statements also delay their completion until all the underlying data and metadata changes are propagated to all Impala nodes. Internally, Impala inserts have similarities with DDL statements in traditional database systems, because they create metadata needed to track HDFS block locations for new files and they potentially add new partitions to partitioned tables.

Note: Because this option can introduce a delay after each write operation, if you are running a sequence of CREATE DATABASE, CREATE TABLE, ALTER TABLE, INSERT, and similar statements within a setup script, to minimize the overall delay you can enable the SYNC_DDL query option only near the end, before the final DDL statement.

Type: BOOLEAN

Default: FALSE

THREAD_RESERVATION_AGGREGATE_LIMIT query option

The THREAD_RESERVATION_AGGREGATE_LIMIT query option limits the number of reserved threads for a query across all nodes on which it is executing. The option is intended to prevent execution of complex queries that can consume excessive CPU or operating system resources on a cluster. Queries that have more threads than this threshold are rejected by Impala's admission controller before they start executing.

For example, an Impala administrator could set a default value of THREAD_RESERVATION_AGGREGATE_LIMIT=2000 for a resource pool on a 100 node where they expect only relatively simple queries with less than 20 threads per node to run. This will reject queries that require more than 2000 reserved threads across all nodes, for example a query with 21 fragments running on all 100 nodes of the cluster.

You can override the default value per-query or per-session, in the same way as other query options, if you do not want the default THREAD_RESERVATION_AGGREGATE_LIMIT value to apply to a specific query or session.

Type: Number

Default: 0 (no limit)

Added in: Impala 3.1

THREAD_RESERVATION_LIMIT query option

The THREAD_RESERVATION_LIMIT query option limits the number of reserved threads for a query on each node. The option is intended to prevent execution of complex queries that can consume excessive CPU or operating system resources on a single node. Queries that have more threads per node than this threshold are rejected by Impala's admission controller before they start executing. You can see the number of reserved threads for a query in its explain plan in the "Per-Host Resource Reservation" line.

For example, an Impala administrator could set a default value of THREAD_RESERVATION_LIMIT=100 for a resource pool where they expect only relatively simple queries to run. This will reject queries that require more than 100 reserved threads on a node, for example, queries with more than 100 fragments.

You can override the default value per-query or per-session, in the same way as other query options, if you do not want the default THREAD_RESERVATION_LIMIT value to apply to a specific query or session.

Note: The number of reserved threads on a node may be lower than the maximum value in the explain plan if not all fragments of that query are scheduled on every node.

Type: Number

Default: 3000

Added in: Impala 3.1

TIMEZONE query option

The TIMEZONE query option defines the timezone used for conversions between UTC and the local time. If not set, Impala uses the system time zone where the Coordinator Impalad runs. As

query options are not sent to the Coordinator immediately, the timezones are validated only when the query runs.

Impala takes the timezone into a consideration in the following cases:

- When calling the NOW() function
- When converting between Unix time and timestamp if the use_local_tz_for_unix_timestamp_ conversions flag is TRUE
- When reading Parquet timestamps written by Hive if the convert_legacy_hive_parquet_utc_time stamps flag is TRUE

Syntax:

```
SET TIMEZONE=time zone
```

time zone can be a canonical code or a time zone name defined in IANA Time Zone Database. The value is case-sensitive.

Leading/trailing quotes (') and double quotes (") are stripped.

If *time zone* is an empty string, the time zone for the query is set to the default time zone of the Impalad Coordinator.

If *time zone* is NULL or a space character, Impala returns an error when the query is executed.

Type: STRING

Default: The system time zone where the Coordinator impalad runs

Examples:

```
SET TIMEZONE=UTC;
SET TIMEZONE="Europe/Budapest";
```

Added in: Impala 3.1

TOPN_BYTES_LIMIT query option

The TOPN_BYTES_LIMIT query option places a limit on the amount of estimated memory that Impala can process for top-N queries.

Top-N queries are the queries that include both ORDER BY and LIMIT clauses. Top-N queries don't spill to disk so have to keep all rows they process in memory, and those queries can cause out-of-memory issues when running with a large limit and an offset. If the Impala planner estimates that a top-N operator will process more bytes than the TOPN_BYTES_LIMIT value, it will replace the top-N operator with the sort operator. Switching to the sort operator allows Impala to spill to disk, thus requiring less memory than top-N, but potentially with performance penalties.

The option has no effect when set to 0 or -1.

Syntax:

```
SET TOPN_BYTES_LIMIT=limit
```

Type: Number

Default: 536870912 (512 MB)

Added in: Impala 3.1

Impala built-in functions

Impala supports several categories of built-in functions. These functions let you perform mathematical calculations, string manipulation, date calculations, and other kinds of data transformations directly in SQL statements.

The categories of built-in functions supported by Impala are:

- Impala mathematical functions on page 280
- Impala type conversion functions on page 308
- Impala date and time functions on page 311
- Impala conditional functions on page 330
- Impala string functions on page 335
- Impala aggregate functions on page 354.
- Impala analytic functions on page 384
- Impala bit functions on page 298
- Impala miscellaneous functions on page 352

The following is a complete list of built-in functions supported in Impala:

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

Impala mathematical functions

Mathematical functions, or arithmetic functions, perform numeric calculations that are typically more complex than basic addition, subtraction, multiplication, and division. For example, these functions include trigonometric, logarithmic, and base conversion operations.

Note: In Impala, exponentiation uses the POW() function rather than an exponentiation operator such as **.

Related information:

The mathematical functions operate mainly on these data types:

- INT
- BIGINT
- SMALLINT
- TINYINT
- DOUBLE

- FLOAT
- DECIMAL

For the operators that perform the standard operations such as addition, subtraction, multiplication, and division, see Arithmetic operators on page 85.

Functions that perform bitwise operations are explained in Impala bit functions on page 298.

Function reference:

Impala supports the following mathematical functions:

- ABS
- ACOS
- ASIN
- ATAN
- ATAN2
- BIN
- CEIL, CEILING, DCEIL
- CONV
- COS
- COSH
- COT
- DEGREES
- E
- EXP
- FACTORIAL
- FLOOR, DFLOOR
- FMOD
- FNV_HASH
- GREATEST
- HEX
- IS_INF
- IS_NAN
- LEAST
- LN
- LOG
- LOG10
- LOG2
- MAX_INT, MAX_TINYINT, MAX_SMALLINT, MAX_BIGINT
- MIN_INT, MIN_TINYINT, MIN_SMALLINT, MIN_BIGINT
- MOD
- MURMUR_HASH
- NEGATIVE
- PI
- PMOD
- POSITIVE
- POW, POWER, DPOW, FPOW
- PRECISION
- QUOTIENT
- RADIANS
- RAND, RANDOM
- ROUND, DROUND

ABS(numeric_type a)

Purpose: Returns the absolute value of the argument.

Return type: Same as the input value

Usage notes: Use this function to ensure all return values are positive. This is different than the positive() function, which returns its argument unchanged (even if the argument was negative).

ACOS(DOUBLE a)

Purpose: Returns the arccosine of the argument.

Return type: DOUBLE

ASIN(DOUBLE a)

Purpose: Returns the arcsine of the argument.

Return type: DOUBLE

ATAN(DOUBLE a)

Purpose: Returns the arctangent of the argument.

Return type: DOUBLE

ATAN(DOUBLE a, DOUBLE b)

Purpose: Returns the arctangent of the two arguments, with the signs of the arguments used to determine the quadrant of the result.

Return type: DOUBLE

BIN(BIGINT a)

Purpose: Returns the binary representation of an integer value, that is, a string of 0 and 1 digits.

Return type: STRING

CEIL(DOUBLE a), CEIL(DECIMAL(p,s) a), CEILING(DOUBLE a), CEILING(DECIMAL(p,s) a), DCEIL(DOUBLE a), DCEIL(DECIMAL(p,s) a)

Purpose: Returns the smallest integer that is greater than or equal to the argument.

Return type: Same as the input type

CONV(BIGINT n, INT from_base, INT to_base), CONV(STRING s, INT from_base, INT to_base)

Purpose: Returns a string representation of the first argument converted from from_base to to_base. The first argument can be specified as a number or a string. For example, conv(100, 2, 10) and conv ('100', 2, 10) both return '4'.

Return type: STRING

Usage notes:

If to_base is negative, the first argument is treated as signed, and otherwise, it is treated as unsigned. For example:

- conv(-17, 10, -2) returns '-10001', -17 in base 2.
- conv(-17, 10, 10) returns '18446744073709551599'. -17 is interpreted as an unsigned, 2^64-17, and then the value is returned in base 10.

The function returns NULL when the following illegal arguments are specified:

- Any argument is NULL.
- from_base or to_base is below -36 or above 36.
- from_base or to_base is -1, 0, or 1.
- The first argument represents a positive number and from_base is a negative number.

If the first argument represents a negative number and from_base is a negative number, the function returns 0.

If the first argument represents a number larger than the maximum bigint, the function returns:

- The string representation of -1 in to_base if to_base is negative.
- The string representation of 18446744073709551615' (2^64 - 1) in to_base if to_base is positive.

If the first argument does not represent a valid number in from_base, e.g. 3 in base 2 or '1a23' in base 10, the digits in the first argument are evaluated from left-to-right and used if a valid digit in from_base. The invalid digit and the digits to the right are ignored.

For example:

- conv(445, 5, 10) is converted to conv(44, 5, 10) and returns '24'.
- conv('1a23', 10, 16) is converted to conv('1', 10 , 16) and returns '1'.

COS(DOUBLE a)

Purpose: Returns the cosine of the argument.

Return type: DOUBLE

COSH(DOUBLE a)

Purpose: Returns the hyperbolic cosine of the argument.

Return type: DOUBLE

COT(DOUBLE a)

Purpose: Returns the cotangent of the argument.

Return type: DOUBLE

Added in: Impala 2.3.0

DEGREES(DOUBLE a)

Purpose: Converts argument value from radians to degrees.

Return type: DOUBLE

E()

Purpose: Returns the mathematical constant e.

Return type: DOUBLE

EXP(DOUBLE a), DEXP(DOUBLE a)

Purpose: Returns the mathematical constant e raised to the power of the argument.

Return type: DOUBLE

FACTORIAL(integer_type a)

Purpose: Computes the factorial of an integer value. It works with any integer type.

Added in: Impala 2.3.0

Usage notes: You can use either the factorial() function or the ! operator. The factorial of 0 is 1. Likewise, the factorial() function returns 1 for any negative value. The maximum positive value for the input argument is 20; a value of 21 or greater overflows the range for a BIGINT and causes an error.

Return type: BIGINT

Added in: Impala 2.3.0

```
select factorial(5);
+--------------+
| factorial(5) |
+--------------+
| 120          |
+--------------+

select 5!;
+-----+
| 5!  |
+-----+
| 120 |
+-----+
```

FLOOR(DOUBLE a), FLOOR(DECIMAL(p,s) a), DFLOOR(DOUBLE a), DFLOOR(DECIMAL(p,s) a)

Purpose: Returns the largest integer that is less than or equal to the argument.

Return type: Same as the input type

FMOD(DOUBLE a, DOUBLE b), FMOD(FLOAT a, FLOAT b)

Purpose: Returns the modulus of a floating-point number.

Return type: FLOAT or DOUBLE, depending on type of arguments

Added in: Impala 1.1.1

Usage notes:

Because this function operates on DOUBLE or FLOAT values, it is subject to potential rounding errors for values that cannot be represented precisely. Prefer to use whole numbers, or values that you know can be represented precisely by the DOUBLE or FLOAT types.

Examples:

The following examples show equivalent operations with the fmod() function and the % arithmetic operator, for values not subject to any rounding error.

```
select fmod(10,3);
+-------------+
| fmod(10, 3) |
+-------------+
| 1           |
+-------------+

select fmod(5.5,2);
+--------------+
| fmod(5.5, 2) |
+--------------+
| 1.5          |
+--------------+

select 10 % 3;
+--------+
| 10 % 3 |
+--------+
```

```
| 1      |
+--------+
select 5.5 % 2;
+---------+
| 5.5 % 2 |
+---------+
| 1.5     |
+---------+
```

The following examples show operations with the fmod() function for values that cannot be represented precisely by the DOUBLE or FLOAT types, and thus are subject to rounding error. fmod(9.9,3.0) returns a value slightly different than the expected 0.9 because of rounding. fmod(9.9 ,3.3) returns a value quite different from the expected value of 0 because of rounding error during intermediate calculations.

```
select fmod(9.9,3.0);
+--------------------+
| fmod(9.9, 3.0)     |
+--------------------+
| 0.8999996185302734 |
+--------------------+

select fmod(9.9,3.3);
+-------------------+
| fmod(9.9, 3.3)    |
+-------------------+
| 3.299999713897705 |
+-------------------+
```

FNV_HASH(type v)

Purpose: Returns a consistent 64-bit value derived from the input argument, for convenience of implementing hashing logic in an application.

Return type: BIGINT

Usage notes:

You might use the return value in an application where you perform load balancing, bucketing, or some other technique to divide processing or storage.

Because the result can be any 64-bit value, to restrict the value to a particular range, you can use an expression that includes the ABS() function and the % (modulo) operator. For example, to produce a hash value in the range 0-9, you could use the expression ABS(FNV_HASH(x)) % 10.

This function implements the same algorithm that Impala uses internally for hashing, on systems where the CRC32 instructions are not available.

This function implements the Fowler–Noll–Vo hash function, in particular the FNV-1a variation. This is not a perfect hash function: some combinations of values could produce the same result value. It is not suitable for cryptographic use.

Similar input values of different types could produce different hash values, for example the same numeric value represented as SMALLINT or BIGINT, FLOAT or DOUBLE, or DECIMAL(5,2) or DECIMAL(20,5).

Examples:

```
[localhost:21000] > create table h (x int, s string);
[localhost:21000] > insert into h values (0, 'hello'), (1,'wor
ld'), (1234567890,'antidisestablishmentarianism');
[localhost:21000] > select x, fnv_hash(x) from h;
+------------+---------------------+
| x          | fnv_hash(x)         |
```

```
+------------+---------------------+
| 0          | -2611523532599129963 |
| 1          | 4307505193096137732  |
| 1234567890 | 3614724209955230832  |
+------------+---------------------+
[localhost:21000] > select s, fnv_hash(s) from h;
+-----------------------------+---------------------+
| s                           | fnv_hash(s)         |
+-----------------------------+---------------------+
| hello                       | 6414202926103426347 |
| world                       | 6535280128821139475 |
| antidisestablishmentarianism | -209330013948433970 |
+-----------------------------+---------------------+
[localhost:21000] > select s, abs(fnv_hash(s)) % 10 from h;
+-----------------------------+-------------------------+
| s                           | abs(fnv_hash(s)) % 10.0 |
+-----------------------------+-------------------------+
| hello                       | 8                       |
| world                       | 6                       |
| antidisestablishmentarianism | 4                       |
+-----------------------------+-------------------------+
```

For short argument values, the high-order bits of the result have relatively low entropy:

```
[localhost:21000] > create table b (x boolean);
[localhost:21000] > insert into b values (true), (true), (false),
 (false);
[localhost:21000] > select x, fnv_hash(x) from b;
+-------+---------------------+
| x     | fnv_hash(x)         |
+-------+---------------------+
| true  | 2062020650953872396 |
| true  | 2062020650953872396 |
| false | 2062021750465500607 |
| false | 2062021750465500607 |
+-------+---------------------+
```

Added in: Impala 1.2.2

GREATEST(BIGINT a[, BIGINT b ...]), GREATEST(DOUBLE a[, DOUBLE b ...]), GREATEST(DECIMAL(p,s) a[, DECIMAL(p,s) b ...]), GREATEST(STRING a[, STRING b ...]), GREATEST(TIMESTAMP a[, TIMESTAMP b ...])

Purpose: Returns the largest value from a list of expressions.

Return type: same as the initial argument value, except that integer values are promoted to BIGI NT and floating-point values are promoted to DOUBLE; use CAST() when inserting into a smaller numeric column

HEX(BIGINT a), HEX(STRING a)

Purpose: Returns the hexadecimal representation of an integer value, or of the characters in a string.

Return type: STRING

IS_INF(DOUBLE a)

Purpose: Tests whether a value is equal to the special value “inf”, signifying infinity.

Return type: BOOLEAN

Usage notes:

Infinity and NaN can be specified in text data files as inf and nan respectively, and Impala interprets them as these special values. They can also be produced by certain arithmetic expressions; for

example, 1/0 returns Infinity and pow(-1, 0.5) returns NaN. Or you can cast the literal values, such as CAST('nan' AS DOUBLE) or CAST('inf' AS DOUBLE).

IS_NAN(DOUBLE a)

Purpose: Tests whether a value is equal to the special value “NaN”, signifying “not a number”.

Return type: BOOLEAN

Usage notes:

Infinity and NaN can be specified in text data files as inf and nan respectively, and Impala interprets them as these special values. They can also be produced by certain arithmetic expressions; for example, 1/0 returns Infinity and pow(-1, 0.5) returns NaN. Or you can cast the literal values, such as CAST('nan' AS DOUBLE) or CAST('inf' AS DOUBLE).

LEAST(BIGINT a[, BIGINT b ...]), LEAST(DOUBLE a[, DOUBLE b ...]), LEAST(DECIMAL(p,s) a[, DECIMAL(p,s) b ...]), LEAST(STRING a[, STRING b ...]), LEAST(TIMESTAMP a[, TIMESTAMP b ...])

Purpose: Returns the smallest value from a list of expressions.

Return type: same as the initial argument value, except that integer values are promoted to BIGINT and floating-point values are promoted to DOUBLE; use CAST() when inserting into a smaller numeric column

LN(DOUBLE a), DLOG1(DOUBLE a)

Purpose: Returns the natural logarithm of the argument.

Return type: DOUBLE

LOG(DOUBLE base, DOUBLE a)

Purpose: Returns the logarithm of the second argument to the specified base.

Return type: DOUBLE

LOG10(DOUBLE a), DLOG10(DOUBLE a)

Purpose: Returns the logarithm of the argument to the base 10.

Return type: DOUBLE

LOG2(DOUBLE a)

Purpose: Returns the logarithm of the argument to the base 2.

Return type: DOUBLE

MAX_INT(), MAX_TINYINT(), MAX_SMALLINT(), MAX_BIGINT()

Purpose: Returns the largest value of the associated integral type.

Return type: The same as the integral type being checked.

Usage notes: Use the corresponding min_ and max_ functions to check if all values in a column are within the allowed range, before copying data or altering column definitions. If not, switch to the next higher integral type or to a DECIMAL with sufficient precision.

MIN_INT(), MIN_TINYINT(), MIN_SMALLINT(), MIN_BIGINT()

Purpose: Returns the smallest value of the associated integral type (a negative number).

Return type: The same as the integral type being checked.

Usage notes: Use the corresponding min_ and max_ functions to check if all values in a column are within the allowed range, before copying data or altering column definitions. If not, switch to the next higher integral type or to a DECIMAL with sufficient precision.

MOD(numeric_type a, same_type b)

Purpose: Returns the modulus of a number. Equivalent to the % arithmetic operator. Works with any size integer type, any size floating-point type, and DECIMAL with any precision and scale.

Return type: Same as the input value

Added in: Impala 2.20

Usage notes:

Because this function works with DECIMAL values, prefer it over fmod() when working with fractional values. It is not subject to the rounding errors that make fmod() problematic with floating-point numbers.

Query plans shows the MOD() function as the % operator.

Examples:

The following examples show how the mod() function works for whole numbers and fractional values, and how the % operator works the same way. In the case of mod(9.9,3), the type conversion for the second argument results in the first argument being interpreted as DOUBLE, so to produce an accurate DECIMAL result requires casting the second argument or writing it as a DECIMAL literal, 3.0.

```
select mod(10,3);
+-------------+
| mod(10, 3) |
+-------------+
| 1           |
+-------------+

select mod(5.5,2);
+--------------+
| mod(5.5, 2) |
+--------------+
| 1.5          |
+--------------+

select 10 % 3;
+--------+
| 10 % 3 |
+--------+
| 1      |
+--------+
select 5.5 % 2;
+---------+
| 5.5 % 2 |
+---------+
| 1.5     |
+---------+

select mod(9.9,3.3);
+---------------+
| mod(9.9, 3.3) |
+---------------+
| 0.0           |
+---------------+

select mod(9.9,3);
+--------------------+
| mod(9.9, 3)        |
+--------------------+
| 0.8999996185302734 |
+--------------------+
select mod(9.9, cast(3 as decimal(2,1)));
```

```
+----------------------------------+
| mod(9.9, cast(3 as decimal(2,1))) |
+----------------------------------+
| 0.9                              |
+----------------------------------+

select mod(9.9,3.0);
+---------------+
| mod(9.9, 3.0) |
+---------------+
| 0.9           |
+---------------+
```

MURMUR_HASH(type v)

Purpose: Returns a consistent 64-bit value derived from the input argument, for convenience of implementing MurmurHash2 non-cryptographic hash function.

Return type: BIGINT

Usage notes:

You might use the return value in an application where you perform load balancing, bucketing, or some other technique to divide processing or storage. This function provides a good performance for all kinds of keys such as number, ascii string and UTF-8. It can be recommended as general-purpose hashing function.

Regarding comparison of murmur_hash with fnv_hash, murmur_hash is based on Murmur2 hash algorithm and fnv_hash function is based on FNV-1a hash algorithm. Murmur2 and FNV-1a can show very good randomness and performance compared with well known other hash algorithms, but Murmur2 slightly show better randomness and performance than FNV-1a.

Similar input values of different types could produce different hash values, for example the same numeric value represented as SMALLINT or BIGINT, FLOAT or DOUBLE, or DECIMAL(5,2) or DECIMAL(20,5).

Examples:

```
[localhost:21000] > create table h (x int, s string);
[localhost:21000] > insert into h values (0, 'hello'), (1,'wor
ld'), (1234567890,'antidisestablishmentarianism');
[localhost:21000] > select x, murmur_hash(x) from h;
+------------+---------------------+
| x          | murmur_hash(x)      |
+------------+---------------------+
| 0          | 6960269033020761575 |
| 1          | -780611581681153783 |
| 1234567890 | -5754914572385924334 |
+------------+---------------------+
[localhost:21000] > select s, murmur_hash(s) from h;
+------------------------------+---------------------+
| s                            | murmur_hash(s)      |
+------------------------------+---------------------+
| hello                        | 2191231550387646743 |
| world                        | 5568329560871645431 |
| antidisestablishmentarianism | -2261804666958489663 |
+------------------------------+---------------------+
```

For short argument values, the high-order bits of the result have relatively higher entropy than fnv_hash:

```
[localhost:21000] > create table b (x boolean);
[localhost:21000] > insert into b values (true), (true), (false),
 (false);
```

```
[localhost:21000] > select x, murmur_hash(x) from b;
+-------+----------------------+
| x     | murmur_hash(x)       |
+-------+----------------------++
| true  | -5720937396023583481 |
| true  | -5720937396023583481 |
| false | 6351753276682545529  |
| false | 6351753276682545529  |
+-------+--------------------+-+
```

Added in: Impala 2.12.0

NEGATIVE(numeric_type a)

Purpose: Returns the argument with the sign reversed; returns a positive value if the argument was already negative.

Return type: Same as the input value

Usage notes: Use -abs(a) instead if you need to ensure all return values are negative.

PI()

Purpose: Returns the constant pi.

Return type: double

PMOD(BIGINT a, BIGINT b), PMOD(DOUBLE a, DOUBLE b)

Purpose: Returns the positive modulus of a number. Primarily for Hive SQL compatibility.

Return type: INT or DOUBLE, depending on type of arguments

Examples:

The following examples show how the fmod() function sometimes returns a negative value depending on the sign of its arguments, and the pmod() function returns the same value as fmod(), but sometimes with the sign flipped.

```
select fmod(-5,2);
+-------------+
| fmod(-5, 2) |
+-------------+
| -1          |
+-------------+

select pmod(-5,2);
+-------------+
| pmod(-5, 2) |
+-------------+
| 1           |
+-------------+

select fmod(-5,-2);
+--------------+
| fmod(-5, -2) |
+--------------+
| -1           |
+--------------+

select pmod(-5,-2);
+--------------+
| pmod(-5, -2) |
+--------------+
| -1           |
+--------------+
```

```
select fmod(5,-2);
+-------------+
| fmod(5, -2) |
+-------------+
| 1           |
+-------------+

select pmod(5,-2);
+-------------+
| pmod(5, -2) |
+-------------+
| -1          |
+-------------+
```

POSITIVE(numeric_type a)

Purpose: Returns the original argument unchanged (even if the argument is negative).

Return type: Same as the input value

Usage notes: Use abs() instead if you need to ensure all return values are positive.

POW(DOUBLE a, double p), POWER(DOUBLE a, DOUBLE p), DPOW(DOUBLE a, DOUBLE p), FPOW(DOUBLE a, DOUBLE p)

Purpose: Returns the first argument raised to the power of the second argument.

Return type: DOUBLE

PRECISION(numeric_expression)

Purpose: Computes the precision (number of decimal digits) needed to represent the type of the argument expression as a DECIMAL value.

Usage notes:

Typically used in combination with the scale() function, to determine the appropriate DECIMAL(*precision*,*scale*) type to declare in a CREATE TABLE statement or CAST() function.

Return type: INT

Examples:

The following examples demonstrate how to check the precision and scale of numeric literals or other numeric expressions. Impala represents numeric literals in the smallest appropriate type. 5 is a TINYINT value, which ranges from -128 to 127, therefore 3 decimal digits are needed to represent the entire range, and because it is an integer value there are no fractional digits. 1.333 is interpreted as a DECIMAL value, with 4 digits total and 3 digits after the decimal point.

```
[localhost:21000] > select precision(5), scale(5);
+--------------+----------+
| precision(5) | scale(5) |
+--------------+----------+
| 3            | 0        |
+--------------+----------+
[localhost:21000] > select precision(1.333), scale(1.333);
+------------------+--------------+
| precision(1.333) | scale(1.333) |
+------------------+--------------+
| 4                | 3            |
+------------------+--------------+
[localhost:21000] > with t1 as
  ( select cast(12.34 as decimal(20,2)) x union select cast(1 as
 decimal(8,6)) x )
  select precision(x), scale(x) from t1 limit 1;
+--------------+----------+
| precision(x) | scale(x) |
```

```
+--------------+----------+
| 24           | 6        |
+--------------+----------+
```

QUOTIENT(BIGINT numerator, BIGINT denominator), QUOTIENT(DOUBLE numerator, DOUBLE denominator)

Purpose: Returns the first argument divided by the second argument, discarding any fractional part. Avoids promoting integer arguments to DOUBLE as happens with the / SQL operator. Also includes an overload that accepts DOUBLE arguments, discards the fractional part of each argument value before dividing, and again returns BIGINT. With integer arguments, this function works the same as the DIV operator.

Return type: BIGINT

RADIANS(DOUBLE a)

Purpose: Converts argument value from degrees to radians.

Return type: DOUBLE

RAND(), RAND(BIGINT seed), RANDOM(), RANDOM(BIGINT seed)

Purpose: Returns a random value between 0 and 1. After rand() is called with a seed argument, it produces a consistent random sequence based on the seed value.

Return type: DOUBLE

Usage notes: Currently, the random sequence is reset after each query, and multiple calls to rand() within the same query return the same value each time. For different number sequences that are different for each query, pass a unique seed value to each call to rand(). For example, select rand(unix_timestamp()) from ...

Examples:

The following examples show how rand() can produce sequences of varying predictability, so that you can reproduce query results involving random values or generate unique sequences of random values for each query. When rand() is called with no argument, it generates the same sequence of values each time, regardless of the ordering of the result set. When rand() is called with a constant integer, it generates a different sequence of values, but still always the same sequence for the same seed value. If you pass in a seed value that changes, such as the return value of the expression unix_timestamp(now()), each query will use a different sequence of random values, potentially more useful in probability calculations although more difficult to reproduce at a later time. Therefore, the final two examples with an unpredictable seed value also include the seed in the result set, to make it possible to reproduce the same random sequence later.

```
select x, rand() from three_rows;
+---+-----------------------+
| x | rand()                |
+---+-----------------------+
| 1 | 0.0004714746030380365 |
| 2 | 0.5895895192351144    |
| 3 | 0.4431900859080209    |
+---+-----------------------+
select x, rand() from three_rows order by x desc;
+---+-----------------------+
| x | rand()                |
+---+-----------------------+
| 3 | 0.0004714746030380365 |
| 2 | 0.5895895192351144    |
| 1 | 0.4431900859080209    |
+---+-----------------------+

select x, rand(1234) from three_rows order by x;
+---+----------------------+
```

```
| x | rand(1234)           |
+---+----------------------+
| 1 | 0.7377511392057646   |
| 2 | 0.009428468537250751 |
| 3 | 0.208117277924026    |
+---+----------------------+

select x, rand(1234) from three_rows order by x desc;
+---+----------------------+
| x | rand(1234)           |
+---+----------------------+
| 3 | 0.7377511392057646   |
| 2 | 0.009428468537250751 |
| 1 | 0.208117277924026    |
+---+----------------------+
select x, unix_timestamp(now()), rand(unix_timestamp(now()))
  from three_rows order by x;
+---+-----------------------+-----------------------------+
| x | unix_timestamp(now()) | rand(unix_timestamp(now())) |
+---+-----------------------+-----------------------------+
| 1 | 1440777752            | 0.002051228658320023        |
| 2 | 1440777752            | 0.5098743483004506          |
| 3 | 1440777752            | 0.9517714925817081          |
+---+-----------------------+-----------------------------+
select x, unix_timestamp(now()), rand(unix_timestamp(now()))
  from three_rows order by x desc;
+---+-----------------------+-----------------------------+
| x | unix_timestamp(now()) | rand(unix_timestamp(now())) |
+---+-----------------------+-----------------------------+
| 3 | 1440777761            | 0.9985985015512437          |
| 2 | 1440777761            | 0.3251255333074953          |
| 1 | 1440777761            | 0.02422675025846192         |
+---+-----------------------+-----------------------------+
```

ROUND(DOUBLE a), ROUND(DOUBLE a, INT d), ROUND(DECIMAL a, int_type d), DROUND(DOUBLE a), DROUND(DOUBLE a, INT d), DROUND(DECIMAL(p,s) a, int_type d)

Purpose: Rounds a floating-point value. By default (with a single argument), rounds to the nearest integer. Values ending in .5 are rounded up for positive numbers, down for negative numbers (that is, away from zero). The optional second argument specifies how many digits to leave after the decimal point; values greater than zero produce a floating-point return value rounded to the requested number of digits to the right of the decimal point.

Return type: Same as the input type

SCALE(numeric_expression)

Purpose: Computes the scale (number of decimal digits to the right of the decimal point) needed to represent the type of the argument expression as a DECIMAL value.

Usage notes:

Typically used in combination with the precision() function, to determine the appropriate DECI MAL(*precision*,*scale*) type to declare in a CREATE TABLE statement or CAST() function.

Return type: int

Examples:

The following examples demonstrate how to check the precision and scale of numeric literals or other numeric expressions. Impala represents numeric literals in the smallest appropriate type. 5 is a TINYINT value, which ranges from -128 to 127, therefore 3 decimal digits are needed to represent

the entire range, and because it is an integer value there are no fractional digits. 1.333 is interpreted as a DECIMAL value, with 4 digits total and 3 digits after the decimal point.

```
[localhost:21000] > select precision(5), scale(5);
+--------------+----------+
| precision(5) | scale(5) |
+--------------+----------+
| 3            | 0        |
+--------------+----------+
[localhost:21000] > select precision(1.333), scale(1.333);
+------------------+--------------+
| precision(1.333) | scale(1.333) |
+------------------+--------------+
| 4                | 3            |
+------------------+--------------+
[localhost:21000] > with t1 as
  ( select cast(12.34 as decimal(20,2)) x union select cast(1 as
 decimal(8,6)) x )
  select precision(x), scale(x) from t1 limit 1;
+--------------+----------+
| precision(x) | scale(x) |
+--------------+----------+
| 24           | 6        |
+--------------+----------+
```

SIGN(DOUBLE a)

Purpose: Returns -1, 0, or 1 to indicate the signedness of the argument value.

Return type: INT

SIN(DOUBLE a)

Purpose: Returns the sine of the argument.

Return type: DOUBLE

SINH(DOUBLE a)

Purpose: Returns the hyperbolic sine of the argument.

Return type: DOUBLE

SQRT(DOUBLE a), DSQRT(DOUBLE a)

Purpose: Returns the square root of the argument.

Return type: DOUBLE

TAN(DOUBLE a)

Purpose: Returns the tangent of the argument.

Return type: DOUBLE

TANH(DOUBLE a)

Purpose: Returns the hyperbolic tangent of the argument.

Return type: DOUBLE

TRUNCATE(DOUBLE_or_DECIMAL a[, digits_to_leave]), DTRUNC(DOUBLE_or_DECIMAL a[, digits_to_leave]), TRUNC(DOUBLE_or_DECIMAL a[, digits_to_leave])

Purpose: Removes some or all fractional digits from a numeric value.

Arguments: With a single floating-point argument, removes all fractional digits, leaving an integer value. The optional second argument specifies the number of fractional digits to include in the return value, and only applies when the argument type is DECIMAL. A second argument of 0

truncates to a whole integer value. A second argument of negative N sets N digits to 0 on the left side of the decimal

Scale argument: The scale argument applies only when truncating DECIMAL values. It is an integer specifying how many significant digits to leave to the right of the decimal point. A scale argument of 0 truncates to a whole integer value. A scale argument of negative N sets N digits to 0 on the left side of the decimal point.

TRUNCATE(), DTRUNC(), and TRUNC() are aliases for the same function.

Return type: Same as the input type

Added in: The TRUNC() alias was added in Impala 2.10.0.

Usage notes:

You can also pass a DOUBLE argument, or DECIMAL argument with optional scale, to the DTRUNC() or TRUNCATE functions. Using the TRUNC() function for numeric values is common with other industry-standard database systems, so you might find such TRUNC() calls in code that you are porting to Impala.

The TRUNC() function also has a signature that applies to TIMESTAMP values.

Examples:

The following examples demonstrate the TRUNCATE() and DTRUNC() signatures for this function:

```
select truncate(3.45);
+----------------+
| truncate(3.45) |
+----------------+
| 3              |
+----------------+

select truncate(-3.45);
+-----------------+
| truncate(-3.45) |
+-----------------+
| -3              |
+-----------------+

select truncate(3.456,1);
+--------------------+
| truncate(3.456, 1) |
+--------------------+
| 3.4                |
+--------------------+

select dtrunc(3.456,1);
+------------------+
| dtrunc(3.456, 1) |
+------------------+
| 3.4              |
+------------------+

select truncate(3.456,2);
+--------------------+
| truncate(3.456, 2) |
+--------------------+
| 3.45               |
+--------------------+

select truncate(3.456,7);
+--------------------+
```

```
| truncate(3.456, 7) |
+--------------------+
| 3.4560000          |
+--------------------+
```

The following examples demonstrate using TRUNC() with DECIMAL or DOUBLE values, and with an optional scale argument for DECIMAL values. (The behavior is the same for the TRUN CATE() and DTRUNC() aliases also.)

```
create table t1 (d decimal(20,7));
-- By default, no digits to the right of the decimal point.
insert into t1 values (1.1), (2.22), (3.333), (4.4444), (5.5555
5);
select trunc(d) from t1 order by d;
+----------+
| trunc(d) |
+----------+
| 1        |
| 2        |
| 3        |
| 4        |
| 5        |
+----------+

-- 1 digit to the right of the decimal point.
select trunc(d,1) from t1 order by d;
+-------------+
| trunc(d, 1) |
+-------------+
| 1.1         |
| 2.2         |
| 3.3         |
| 4.4         |
| 5.5         |
+-------------+

-- 2 digits to the right of the decimal point,
-- including trailing zeroes if needed.
select trunc(d,2) from t1 order by d;
+-------------+
| trunc(d, 2) |
+-------------+
| 1.10        |
| 2.22        |
| 3.33        |
| 4.44        |
| 5.55        |
+-------------+
insert into t1 values (9999.9999), (8888.8888);

-- Negative scale truncates digits to the left
-- of the decimal point.
select trunc(d,-2) from t1 where d > 100 order by d;
+--------------+
| trunc(d, -2) |
+--------------+
| 8800         |
| 9900         |
+--------------+

-- The scale of the result is adjusted to match the
-- scale argument.
```

```
select trunc(d,2),
  precision(trunc(d,2)) as p,
  scale(trunc(d,2)) as s
from t1 order by d;
+-------------+----+---+
| trunc(d, 2) | p  | s |
+-------------+----+---+
| 1.10        | 15 | 2 |
| 2.22        | 15 | 2 |
| 3.33        | 15 | 2 |
| 4.44        | 15 | 2 |
| 5.55        | 15 | 2 |
| 8888.88     | 15 | 2 |
| 9999.99     | 15 | 2 |
+-------------+----+---+
```

```
create table dbl (d double);

insert into dbl values
  (1.1), (2.22), (3.333), (4.4444), (5.55555),
  (8888.8888), (9999.9999);

-- With double values, there is no optional scale argument.
select trunc(d) from dbl order by d;
+----------+
| trunc(d) |
+----------+
| 1        |
| 2        |
| 3        |
| 4        |
| 5        |
| 8888     |
| 9999     |
+----------+
```

UNHEX(STRING a)

Purpose: Returns a string of characters with ASCII values corresponding to pairs of hexadecimal digits in the argument.

Return type: STRING

WIDTH_BUCKET(DECIMAL expr, DECIMAL min_value, DECIMAL max_value, INT num_buckets)

Purpose: Returns the bucket number in which the expr value would fall in the histogram where its range between min_value and max_value is divided into num_buckets buckets of identical sizes.

The function returns:

- NULL if any argument is NULL.
- 0 if expr < min_value.
- num_buckets + 1 if expr >= max_val.
- If none of the above, the bucket number where expr falls.

Arguments:The following rules apply to the arguments.

- min_val is the minimum value of the histogram range.
- max_val is the maximum value of the histogram range.
- num_buckets must be greater than 0.
- min_value must be less than max_value.

Usage notes:

Each bucket contains values equal to or greater than the base value of that bucket and less than the base value of the next bucket. For example, with width_bucket(8, 1, 10, 3), the bucket ranges are actually the 0th "underflow bucket" with the range (-infinity to 0.999...), (1 to 3.999...), (4, to 6.999...), (7 to 9.999...), and the "overflow bucket" with the range (10 to infinity).

Return type: BIGINT

Added in: Impala 3.1.

Examples:

The below function creates 3 buckets between the range of 1 and 20 with the bucket width of 6.333, and returns 2 for the bucket #2 where the value 8 falls in:

```
WIDTH_BUCKET(8, 1, 20, 3)
```

The below statement returns a list of accounts with the energy spending and the spending bracket each account falls in, between 0 and 11. Bucket 0 (underflow bucket) will be assigned to the accounts whose energy spendings are less than $50. Bucket 11 (overflow bucket) will be assigned to the accounts whose energy spendings are more than or equal to $1000.

```
SELECT account, invoice_amount, WIDTH_BUCKET(invoice_amount,50,1
000,10)
FROM invoices_june2018
ORDER BY 3;
```

Related Information

Mathematical constant e
Factorial
Fowler–Noll–Vo hash function
Natural logarithm
MurmurHash2
More Hash Function Tests
Hash functions: An empirical comparison

Impala bit functions

Bit manipulation functions perform bitwise operations involved in scientific processing or computer science algorithms. For example, these functions include setting, clearing, or testing bits within an integer value, or changing the positions of bits with or without wraparound.

If a function takes two integer arguments that are required to be of the same type, the smaller argument is promoted to the type of the larger one if required. For example, BITAND(1,4096) treats both arguments as SMALLINT, because 1 can be represented as a TINYINT but 4096 requires a SMALLINT.

Remember that all Impala integer values are signed. Therefore, when dealing with binary values where the most significant bit is 1, the specified or returned values might be negative when represented in base 10.

Whenever any argument is NULL, either the input value, bit position, or number of shift or rotate positions, the return value from any of these functions is also NULL

Related information:

The bit functions operate on all the integral data types:

- INT
- BIGINT
- SMALLINT
- TINYINT

Function reference:

Impala supports the following bit functions:

- BITAND
- BITNOT
- BITOR
- BITXOR
- COUNTSET
- GETBIT
- ROTATELEFT
- ROTATERIGHT
- SETBIT
- SHIFTLEFT
- SHIFTRIGHT

BITAND(integer_type a, same_type b)

Purpose: Returns an integer value representing the bits that are set to 1 in both of the arguments. If the arguments are of different sizes, the smaller is promoted to the type of the larger.

Usage notes: The BITAND() function is equivalent to the & binary operator.

Return type: Same as the input value

Examples:

The following examples show the results of ANDing integer values. 255 contains all 1 bits in its lowermost 7 bits. 32767 contains all 1 bits in its lowermost 15 bits. You can use the bin() function to check the binary representation of any integer value, although the result is always represented as a 64-bit value. If necessary, the smaller argument is promoted to the type of the larger one.

```
select bitand(255, 32767); /* 0000000011111111 & 0111111111111111
 */
+--------------------+
| bitand(255, 32767) |
+--------------------+
| 255                |
+--------------------+
select bitand(32767, 1); /* 0111111111111111 & 0000000000000001
 */
+------------------+
| bitand(32767, 1) |
+------------------+
| 1                |
+------------------+

select bitand(32, 16); /* 00010000 & 00001000 */
+----------------+
| bitand(32, 16) |
+----------------+
| 0              |
+----------------+

select bitand(12,5); /* 00001100 & 00000101 */
+---------------+
| bitand(12, 5) |
+---------------+
| 4             |
+---------------+
select bitand(-1,15); /* 11111111 & 00001111 */
+----------------+
| bitand(-1, 15) |
+----------------+
```

```
| 15             |
+----------------+
```

BITNOT(integer_type a)

Purpose: Inverts all the bits of the input argument.

Usage notes: The BITNOT() function is equivalent to the ~ unary operator.

Examples:

These examples illustrate what happens when you flip all the bits of an integer value. The sign always changes. The decimal representation is one different between the positive and negative values.

```
select bitnot(127); /* 01111111 -> 10000000 */
+-------------+
| bitnot(127) |
+-------------+
| -128        |
+-------------+

select bitnot(16); /* 00010000 -> 11101111 */
+------------+
| bitnot(16) |
+------------+
| -17        |
+------------+

select bitnot(0); /* 00000000 -> 11111111 */
+-----------+
| bitnot(0) |
+-----------+
| -1        |
+-----------+

select bitnot(-128); /* 10000000 -> 01111111 */
+--------------+
| bitnot(-128) |
+--------------+
| 127          |
+--------------+
```

BITOR(integer_type a, same_type b)

Purpose: Returns an integer value representing the bits that are set to 1 in either of the arguments. If the arguments are of different sizes, the smaller is promoted to the type of the larger.

Usage notes: The BITOR() function is equivalent to the | binary operator.

Return type: Same as the input value

Examples:

The following examples show the results of ORing integer values.

```
select bitor(1,4); /* 00000001 | 00000100 */
+-------------+
| bitor(1, 4) |
+-------------+
| 5           |
+-------------+

select bitor(16,48); /* 00001000 | 00011000 */
+---------------+
| bitor(16, 48) |
```

```
+---------------+
| 48            |
+---------------+

select bitor(0,7); /* 00000000 | 00000111 */
+-------------+
| bitor(0, 7) |
+-------------+
| 7           |
+-------------+
```

BITXOR(integer_type a, same_type b)

Purpose: Returns an integer value representing the bits that are set to 1 in one but not both of the arguments. If the arguments are of different sizes, the smaller is promoted to the type of the larger.

Usage notes: The BITXOR() function is equivalent to the ^ binary operator.

Return type: Same as the input value

Examples:

The following examples show the results of XORing integer values. XORing a non-zero value with zero returns the non-zero value. XORing two identical values returns zero, because all the 1 bits from the first argument are also 1 bits in the second argument. XORing different non-zero values turns off some bits and leaves others turned on, based on whether the same bit is set in both arguments.

```
select bitxor(0,15); /* 00000000 ^ 00001111 */
+---------------+
| bitxor(0, 15) |
+---------------+
| 15            |
+---------------+

select bitxor(7,7); /* 00000111 ^ 00000111 */
+--------------+
| bitxor(7, 7) |
+--------------+
| 0            |
+--------------+

select bitxor(8,4); /* 00001000 ^ 00000100 */
+--------------+
| bitxor(8, 4) |
+--------------+
| 12           |
+--------------+

select bitxor(3,7); /* 00000011 ^ 00000111 */
+--------------+
| bitxor(3, 7) |
+--------------+
| 4            |
+--------------+
```

COUNTSET(integer_type a [, INT zero_or_one])

Purpose: By default, returns the number of 1 bits in the specified integer value. If the optional second argument is set to zero, it returns the number of 0 bits instead.

Usage notes:

In discussions of information theory, this operation is referred to as the “population count” or “popcount”.

Return type: Same as the input value

Examples:

The following examples show how to count the number of 1 bits in an integer value.

```
select countset(1); /* 00000001 */
+-------------+
| countset(1) |
+-------------+
| 1           |
+-------------+

select countset(3); /* 00000011 */
+-------------+
| countset(3) |
+-------------+
| 2           |
+-------------+

select countset(16); /* 00010000 */
+--------------+
| countset(16) |
+--------------+
| 1            |
+--------------+

select countset(17); /* 00010001 */
+--------------+
| countset(17) |
+--------------+
| 2            |
+--------------+

select countset(7,1); /* 00000111 = 3 1 bits; the function counts
 1 bits by default */
+----------------+
| countset(7, 1) |
+----------------+
| 3              |
+----------------+

select countset(7,0); /* 00000111 = 5 0 bits; third argument can
only be 0 or 1 */
+----------------+
| countset(7, 0) |
+----------------+
| 5              |
+----------------+
```

GETBIT(integer_type a, INT position)

Purpose: Returns a 0 or 1 representing the bit at a specified position. The positions are numbered right to left, starting at zero. The position argument cannot be negative.

Usage notes:

When you use a literal input value, it is treated as an 8-bit, 16-bit, and so on value, the smallest type that is appropriate. The type of the input value limits the range of the positions. Cast the input value to the appropriate type if you need to ensure it is treated as a 64-bit, 32-bit, and so on value.

Return type: Same as the input value

Examples:

The following examples show how to test a specific bit within an integer value.

```
select getbit(1,0); /* 00000001 */
+--------------+
| getbit(1, 0) |
+--------------+
| 1            |
+--------------+
select getbit(16,1) /* 00010000 */
+---------------+
| getbit(16, 1) |
+---------------+
| 0             |
+---------------+

select getbit(16,4) /* 00010000 */
+---------------+
| getbit(16, 4) |
+---------------+
| 1             |
+---------------+
select getbit(16,5) /* 00010000 */
+---------------+
| getbit(16, 5) |
+---------------+
| 0             |
+---------------+

select getbit(-1,3); /* 11111111 */
+---------------+
| getbit(-1, 3) |
+---------------+
| 1             |
+---------------+

select getbit(-1,25); /* 11111111 */
ERROR: Invalid bit position: 25

select getbit(cast(-1 as int),25); /* 11111111111111111111111111
111111 */
+-----------------------------+
| getbit(cast(-1 as int), 25) |
+-----------------------------+
| 1                           |
+-----------------------------+
```

ROTATELEFT(integer_type a, INT positions)

Purpose: Rotates an integer value left by a specified number of bits. As the most significant bit is taken out of the original value, if it is a 1 bit, it is “rotated” back to the least significant bit. Therefore, the final value has the same number of 1 bits as the original value, just in different positions. In computer science terms, this operation is a “circular shift”.

Usage notes:

Specifying a second argument of zero leaves the original value unchanged. Rotating a -1 value by any number of positions still returns -1, because the original value has all 1 bits and all the 1 bits are preserved during rotation. Similarly, rotating a 0 value by any number of positions still returns 0. Rotating a value by the same number of bits as in the value returns the same value. Because this is a circular operation, the number of positions is not limited to the number of bits in the input value. For example, rotating an 8-bit value by 1, 9, 17, and so on positions returns an identical result in each case.

Return type: Same as the input value

Examples:

```
select rotateleft(1,4); /* 00000001 -> 00010000 */
+------------------+
| rotateleft(1, 4) |
+------------------+
| 16               |
+------------------+
select rotateleft(-1,155); /* 11111111 -> 11111111 */
+----------------------+
| rotateleft(-1, 155) |
+----------------------+
| -1                   |
+----------------------+

select rotateleft(-128,1); /* 10000000 -> 00000001 */
+----------------------+
| rotateleft(-128, 1) |
+----------------------+
| 1                    |
+----------------------+

select rotateleft(-127,3); /* 10000001 -> 00001100 */
+----------------------+
| rotateleft(-127, 3) |
+----------------------+
| 12                   |
+----------------------+
```

ROTATERIGHT(integer_type a, INT positions)

Purpose: Rotates an integer value right by a specified number of bits. As the least significant bit is taken out of the original value, if it is a 1 bit, it is “rotated” back to the most significant bit. Therefore, the final value has the same number of 1 bits as the original value, just in different positions. In computer science terms, this operation is a “circular shift”.

Usage notes:

Specifying a second argument of zero leaves the original value unchanged. Rotating a -1 value by any number of positions still returns -1, because the original value has all 1 bits and all the 1 bits are preserved during rotation. Similarly, rotating a 0 value by any number of positions still returns 0. Rotating a value by the same number of bits as in the value returns the same value. Because this is a circular operation, the number of positions is not limited to the number of bits in the input value. For example, rotating an 8-bit value by 1, 9, 17, and so on positions returns an identical result in each case.

Return type: Same as the input value

Examples:

```
select rotateright(16,4); /* 00010000 -> 00000001 */
+--------------------+
| rotateright(16, 4) |
+--------------------+
| 1                  |
+--------------------+
select rotateright(-1,155); /* 11111111 -> 11111111 */
+----------------------+
| rotateright(-1, 155) |
+----------------------+
| -1                   |
```

```
+----------------------+

select rotateright(-128,1); /* 10000000 -> 01000000 */
+----------------------+
| rotateright(-128, 1) |
+----------------------+
| 64                   |
+----------------------+

select rotateright(-127,3); /* 10000001 -> 00110000 */
+----------------------+
| rotateright(-127, 3) |
+----------------------+
| 48                   |
+----------------------+
```

SETBIT(integer_type a, INT position [, INT zero_or_one])

Purpose: By default, changes a bit at a specified position to a 1, if it is not already. If the optional third argument is set to zero, the specified bit is set to 0 instead.

Usage notes:

If the bit at the specified position was already 1 (by default) or 0 (with a third argument of zero), the return value is the same as the first argument. The positions are numbered right to left, starting at zero. (Therefore, the return value could be different from the first argument even if the position argument is zero.) The position argument cannot be negative.

When you use a literal input value, it is treated as an 8-bit, 16-bit, and so on value, the smallest type that is appropriate. The type of the input value limits the range of the positions. Cast the input value to the appropriate type if you need to ensure it is treated as a 64-bit, 32-bit, and so on value.

Return type: Same as the input value

Examples:

```
select setbit(0,0); /* 00000000 -> 00000001 */
+--------------+
| setbit(0, 0) |
+--------------+
| 1            |
+--------------+
select setbit(0,3); /* 00000000 -> 00001000 */
+--------------+
| setbit(0, 3) |
+--------------+
| 8            |
+--------------+

select setbit(7,3); /* 00000111 -> 00001111 */
+--------------+
| setbit(7, 3) |
+--------------+
| 15           |
+--------------+

select setbit(15,3); /* 00001111 -> 00001111 */
+---------------+
| setbit(15, 3) |
+---------------+
| 15            |
+---------------+
```

```
select setbit(0,32); /* By default, 0 is a TINYINT with only 8
 bits. */
ERROR: Invalid bit position: 32

select setbit(cast(0 as bigint),32); /* For BIGINT, the position
can be 0..63. */
+-------------------------------+
| setbit(cast(0 as bigint), 32) |
+-------------------------------+
| 4294967296                    |
+-------------------------------+

select setbit(7,3,1); /* 00000111 -> 00001111; setting to 1 is t
he default */
+-----------------+
| setbit(7, 3, 1) |
+-----------------+
| 15              |
+-----------------+

select setbit(7,2,0); /* 00000111 -> 00000011; third argument of
0 clears instead of sets */
+-----------------+
| setbit(7, 2, 0) |
+-----------------+
| 3               |
+-----------------+
```

SHIFTLEFT(integer_type a, INT positions)

Purpose: Shifts an integer value left by a specified number of bits. As the most significant bit is taken out of the original value, it is discarded and the least significant bit becomes 0. In computer science terms, this operation is a “logical shift”.

Usage notes:

The final value has either the same number of 1 bits as the original value, or fewer. Shifting an 8-bit value by 8 positions, a 16-bit value by 16 positions, and so on produces a result of zero.

Specifying a second argument of zero leaves the original value unchanged. Shifting any value by 0 returns the original value. Shifting any value by 1 is the same as multiplying it by 2, as long as the value is small enough; larger values eventually become negative when shifted, as the sign bit is set. Starting with the value 1 and shifting it left by N positions gives the same result as 2 to the Nth power, or pow(2,*N*).

Return type: Same as the input value

Examples:

```
select shiftleft(1,0); /* 00000001 -> 00000001 */
+-----------------+
| shiftleft(1, 0) |
+-----------------+
| 1               |
+-----------------+

select shiftleft(1,3); /* 00000001 -> 00001000 */
+-----------------+
| shiftleft(1, 3) |
+-----------------+
| 8               |
+-----------------+

select shiftleft(8,2); /* 00001000 -> 00100000 */
```

```
+-----------------+
| shiftleft(8, 2) |
+-----------------+
| 32              |
+-----------------+
select shiftleft(127,1); /* 01111111 -> 11111110 */
+-------------------+
| shiftleft(127, 1) |
+-------------------+
| -2                |
+-------------------+

select shiftleft(127,5); /* 01111111 -> 11100000 */
+-------------------+
| shiftleft(127, 5) |
+-------------------+
| -32               |
+-------------------+

select shiftleft(-1,4); /* 11111111 -> 11110000 */
+------------------+
| shiftleft(-1, 4) |
+------------------+
| -16              |
+------------------+
```

SHIFTRIGHT(integer_type a, INT positions)

Purpose: Shifts an integer value right by a specified number of bits. As the least significant bit is taken out of the original value, it is discarded and the most significant bit becomes 0. In computer science terms, this operation is a “logical shift”.

Usage notes:

Therefore, the final value has either the same number of 1 bits as the original value, or fewer. Shifting an 8-bit value by 8 positions, a 16-bit value by 16 positions, and so on produces a result of zero.

Specifying a second argument of zero leaves the original value unchanged. Shifting any value by 0 returns the original value. Shifting any positive value right by 1 is the same as dividing it by 2. Negative values become positive when shifted right.

Return type: Same as the input value

Examples:

```
select shiftright(16,0); /* 00010000 -> 00010000 */
+-------------------+
| shiftright(16, 0) |
+-------------------+
| 16                |
+-------------------+

select shiftright(16,4); /* 00010000 -> 00000001 */
+-------------------+
| shiftright(16, 4) |
+-------------------+
| 1                 |
+-------------------+

select shiftright(16,5); /* 00010000 -> 00000000 */
+-------------------+
| shiftright(16, 5) |
+-------------------+
```

```
| 0                  |
+--------------------+
select shiftright(-1,1); /* 11111111 -> 01111111 */
+--------------------+
| shiftright(-1, 1) |
+--------------------+
| 127                |
+--------------------+

select shiftright(-1,5); /* 11111111 -> 00000111 */
+--------------------+
| shiftright(-1, 5) |
+--------------------+
| 7                  |
+--------------------+
```

Related Information

Hamming weight (population count)
Circular shift
Logical shift

Impala type conversion functions

Conversion functions are usually used in combination with other functions, to explicitly pass the expected data types. Impala has strict rules regarding data types for function parameters. For example, Impala does not automatically convert a DOUBLE value to FLOAT, a BIGINT value to INT, or other conversion where precision could be lost or overflow could occur. Also, for reporting or dealing with loosely defined schemas in big data contexts, you might frequently need to convert values to or from the STRING type.

Note: Although in Impala 2.3, the SHOW FUNCTIONS output for database _IMPALA_BUILTINS contains some function signatures matching the pattern castto*, these functions are not intended for public use and are expected to be deprecated in future.

Function reference:

Impala supports the following type conversion functions:

- CAST
- TYPEOF

CAST(expression AS type)

Purpose: Returns *expression* converted to the *type* data type.

If the *expression* value is of a type that cannot be converted to the target *type*:

- Of DECIMAL, DATE, and BOOLEAN, the function returns an error.
- Of all other types, the function returns NULL.

Usage notes:

Use CAST when passing a column value or literal to a function that expects a parameter with a different type. Frequently used in SQL operations such as CREATE TABLE AS SELECT and INSERT ... VALUES to ensure that values from various sources are of the appropriate type for the destination columns.

Where practical, do a one-time CAST() operation during the ingestion process to make each column into the appropriate type, rather than using many CAST() operations in each query; doing type conversions for each row during each query can be expensive for tables with millions or billions of rows.

The way this function deals with time zones when converting to or from TIMESTAMP values is affected by the ##use_local_tz_for_unix_timestamp_conversions startup flag for the `impalad`

daemon. See TIMESTAMP data type on page 40 for details about how Impala handles time zone considerations for the TIMESTAMP data type.

CAST(expression AS type FORMAT pattern)

Purpose: Returns *expression* converted to the *type* data type based on the *pattern* format string. This signature of CAST() with the FORMAT clause is supported only for casts between STRING / CHAR / VARCHAR types and TIMESTAMP / DATE types.

Note: The patterns allowed in the FORMAT clause support ISO SQL:2016 standard patterns. Those patterns are not the same format patterns used with the other Impala conversion functions, e.g. TO_TIMESTAMP() and FROM_TIMESTAMP().

The following rules apply to *pattern*. Any exceptions to these rules are noted in the Details column of the table below.

- *pattern* is a case-insensitive STRING.
- If *pattern* is NULL, an empty string, or a number, an error returns.
- A fewer digits in *expression* than specified in the *pattern* is accepted if a separator is correctly specified in the *pattern*. For example, CAST('5-01-2017' AS DATE FORMAT 'MM-dd-y yyy') returns DATE'2017-05-01'.
- If fewer number of digits are in *expression* than specified in the *pattern*, the current date is used to complete the year pattern. For example, CAST('19/05' AS DATE FORMAT 'YYYY/MM') will return DATE'2019-05-01' when executed on August 8, 2019.

The following format patterns are supported in the FORMAT clause.

Pattern	Description	Details
YYYY	4-digit year.	
YYY	Last 3 digits of a year.	
YY	Last 2 digits of a year.	
Y	Last digit of a year	
RRRR	4-digit round year	If 1, 3 or 4-digit year values are provided in *expression*, treated as YYYY. If 2-digit years are provided in *expression*, treated as RR. For datetime to string conversions, treated as YYYY. If YYYY, YYY, YY, Y, or RR is given in the same *pattern* for a string to datetime conversion, an error returns.

RR	2-digit round year.	For datetime to string conversion, same as YY.
		For string to datetime conversions, the first 2 digits of the year in the return value depends on the specified two-digit year and the last two digits of the current year as follows:
		• If the specified 2-digit year is 00 to 49:
		  • If the last 2 digits of the current year are 00 to 49, the returned year has the same first 2 digits as the current year.
		  • If the last 2 digits of the current year are 50 to 99, the first 2 digits of the returned year are 1 greater than the first 2 digits of the current year.
		• If the specified 2-digit year is 50 to 99:
		  • If the last 2 digits of the current year are 00 to 49, the first 2 digits of the returned year are 1 less than the first 2 digits of the current year.
		  • If the last 2 digits of the current year are 50 to 99, the returned year has the same first 2 digits as the current year.
		If YYYY, YYY, YY, Y, or RR is given in the same *pattern* for a string to datetime conversion, an error returns.
		If 1-digit year values are provided in *expression*, it is treated as YYYY.

Pattern	Description	Details
MM	Month	In datetime to string conversions, 1-digit month is prefixed with a zero.
DD	Day of month (1-31)	In datetime to string conversions, one digit day is prefixed with a zero.
DDD	Day of year (1-366)	In string to datetime conversions, providing MM and DD along with DDD results an error, e.g. YYYY-MM-DDD.
HH HH12	Hour of day (1-12)	In datetime to string conversions, 1-digit hours are prefixed with a zero. If provided hour in *expression* is not between 1 and 12, returns an error. If no AM/PM is provided in *expression*, the default is AM.
HH24	Hour of day (0-23)	In string to datetime conversions, if HH12, AM, PM are given in the same *pattern*, an error returns.
MI	Minute of hour (0-59)	In datetime to string conversions, 1-digit minutes are prefixed with a zero.
SS	Second of minute (0-59)	In datetime to string conversions, 1-digit seconds are prefixed with a zero.
SSSSS	Second of Day (0-86399)	In string to timestamp conversions, if SS, HH, HH12, HH24, MI, AM/PM are given in the same *pattern*, an error returns.
FF FF1, ..., FF9	Fractional second	A number, 1 to 9, can be used to indicate the number of digits. FF specifies a 9 digits fractional second.
AM PM A.M. P.M.	Meridiem indicator	For datetime to string conversions, AM and PM are treated as synonyms. For example, casting '2019-01-01 11:00 am' to TIMESTAMP with the 'YYYY-MM-DD HH12:MI PM' pattern returns 01-JAN-19 11.00.00.000000 AM. For string to datetime conversion, HH24 in the same *pattern* returns an error.
TZH	Timezone offset hour	An optional sign, + or -, and 2 digits for the value of signed numbers are allowed for the source *expression*, e.g. “+10”, “-05”, "04".
TZM	Timezone offset minute	Unsigned numbers are allowed for the source *expression*.
- . / , ' ; : <space>	Separator	For string to datetime conversions, any separator character in the *pattern* string would match any separator character in the input *expression*. For example, CAST(“20191010” AS DATE FORMAT “YYYY-MM-DD”) returns an error, but CAST("2019-.;10 10" AS DATE FORMAT "YYYY-MM-DD") succeeds.
T	Separates the date from the time.	This pattern is used for accepting ISO 8601 datetime formats. Example: YYYY-MM-DDTHH24:MI:SS.FF9Z
Z	Indicates the zero hour offset from UTC.	This pattern is used for accepting ISO 8601 datetime formats.

Examples:

Input	Output
CAST("02-11-2014" AS DATE FORMAT 'dd-mm-yyyy')	2014-11-02

Input	Output
CAST("365 2014" AS DATE FORMAT 'DDD-YYYY')	2014-12-31
CAST("5-01-26" AS DATE FORMAT 'YY-MM-DD')	Executed at 2019-01-01 11:11:11: 2015-01-26
CAST('2018-11-10T15:11:04Z' AS TIMESTAMP FORMAT 'YYYY-MM-DDTHH24:MI:SSZ')	2018-11-10 15:11:04
CAST("95-01-28" AS DATE FORMAT 'YYY-MM-DD')	Executed at 2019-01-01 11:11:11: 2095-01-28
CAST("49-01-15" AS DATE FORMAT 'RR-MM-DD')	Round year when last 2 digits of current year is greater than 49. Executed at 2099-01-01 11:11:11: 2149-01-15
CAST('2019.10.10 13:30:40.123456 +01:30' AS TIMESTAMP FORMAT 'YYYY-MM-DD HH24:MI:SS.FF9 TZH:TZM')	2019-10-10 13:30:40.123456000

TYPEOF(expression)

Purpose: Returns the name of the data type corresponding to *expression*. For types with extra attributes, such as length for CHAR and VARCHAR, or precision and scale for DECIMAL, includes the full specification of the type.

Return type: STRING

Usage notes: Typically used in interactive exploration of a schema, or in application code that programmatically generates schema definitions such as CREATE TABLE statements, for example, to get the type of an expression such as col1 / col2 or CONCAT(col1, col2, col3). This function is especially useful for arithmetic expressions involving DECIMAL types because the precision and scale of the result is can be different than that of the operands.

Examples:

TYPEOF(2) returns TINYINT.

TYPEOF(NOW()) returns TIMESTAMP.

Impala date and time functions

The underlying Impala data type for date and time data is TIMESTAMP and DATE.

Some of the datetime functions are affected by the setting of the ##use_local_tz_for_unix_timestamp_conversions startup flag for the `impalad` daemon.

- This setting is off by default, meaning that functions such as FROM_UNIXTIME() and UNIX_TIMESTAMP() consider the input values to always represent the UTC time zone.
- This setting also applies when you CAST() a BIGINT value to TIMESTAMP, or a TIMESTAMP value to BIGINT. When this setting is enabled, these functions and operations convert to and from values representing the local time zone. See TIMESTAMP data type on page 40 for details about how Impala handles time zone considerations for the TIMESTAMP data type.

Function reference:

Impala supports the following data and time functions:

- ADD_MONTHS
- ADDDATE

- CURRENT_TIMESTAMP
- DATE_ADD
- DATE_PART
- DATE_SUB
- DATE_TRUNC
- DATEDIFF
- DAY
- DAYNAME
- DAYOFWEEK
- DAYOFYEAR
- DAYS_ADD
- DAYS_SUB
- EXTRACT
- FROM_TIMESTAMP
- FROM_UNIXTIME
- FROM_UTC_TIMESTAMP
- HOUR
- HOURS_ADD
- HOURS_SUB
- INT_MONTHS_BETWEEN
- MICROSECONDS_ADD
- MICROSECONDS_SUB
- MILLISECOND
- MILLISECONDS_ADD
- MILLISECONDS_SUB
- MINUTE
- MINUTES_ADD
- MINUTES_SUB
- MONTH
- MONTHNAME
- MONTHS_ADD
- MONTHS_BETWEEN
- MONTHS_SUB
- NANOSECONDS_ADD
- NANOSECONDS_SUB
- NEXT_DAY
- NOW
- QUARTER
- SECOND
- SECONDS_ADD
- SECONDS_SUB
- SUBDATE
- TIMEOFDAY
- TIMESTAMP_CMP
- TO_DATE
- TO_TIMESTAMP
- TO_UTC_TIMESTAMP
- TRUNC
- UNIX_TIMESTAMP
- UTC_TIMESTAMP

- WEEKOFYEAR
- WEEKS_ADD
- WEEKS_SUB
- YEAR
- YEARS_ADD
- YEARS_SUB

ADD_MONTHS(TIMESTAMP / DATE date, INT months), ADD_MONTHS(TIMESTAMP / DATE date, BIGINT months)

Purpose: Adds *months* to *date* and returns the new date value.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

Usage notes:

Same as MONTHS_ADD(). Available in Impala 1.4 and higher. For compatibility when porting code with vendor extensions.

ADDDATE(TIMESTAMP / DATE date, INT / BIGINT days)

Purpose: Adds *days* to *date* and returns the new date value.

The *days* value can be negative, which gives the same result as the SUBDATE() function.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

CURRENT_DATE()

Purpose: Returns the current date.

Any references to the CURRENT_DATE() function are evaluated at the start of a query. All calls to CURRENT_DATE() within the same query return the same value, and the value does not depend on how long the query takes.

Return type: DATE

CURRENT_TIMESTAMP()

Purpose: Alias for the NOW() function.

Return type: TIMESTAMP

DATE_ADD(TIMESTAMP / DATE date, INT / BIGINT days), DATE_ADD(TIMESTAMP / DATE date, interval_expression)

Purpose: Adds a specified number of days to the *date* argument.

With an INTERVAL expression as the second argument, you can calculate a delta value using other units such as weeks, years, hours, seconds, and so on.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

The following examples show the shorthand notation of an INTERVAL expression, instead of specifying the precise number of days. The INTERVAL notation also lets you work with units smaller than a single day.

```
select now() as right_now, date_add(now(), interval 3 weeks) as
 in_3_weeks;
```

```
+-------------------------------+-------------------------------+
| right_now                     | in_3_weeks                    |
+-------------------------------+-------------------------------+
| 2016-05-20 11:05:39.173331000 | 2016-06-10 11:05:39.173331000 |
+-------------------------------+-------------------------------+

select now() as right_now, date_add(now(), interval 6 hours) as
in_6_hours;
+-------------------------------+-------------------------------+
| right_now                     | in_6_hours                    |
+-------------------------------+-------------------------------+
| 2016-05-20 11:13:51.492536000 | 2016-05-20 17:13:51.492536000 |
+-------------------------------+-------------------------------+
```

Like all date/time functions that deal with months, date_add() handles nonexistent dates past the end of a month by setting the date to the last day of the month. The following example shows how the nonexistent date April 31st is normalized to April 30th:

```
select date_add(cast('2016-01-31' as timestamp), interval 3 mon
ths) as 'april_31st';
+---------------------+
| april_31st          |
+---------------------+
| 2016-04-30 00:00:00 |
+---------------------+
```

DATE_CMP(DATE date1, DATE date2)

Purpose: Compares *date1* and *date2* and returns:

- 0 if the dates are identical.
- 1 if *date1* > *date2*.
- -1 if *date1* < *date2*.
- NULL if *date1* or *date2* is NULL.

Return type: INT

DATE_PART(STRING part, TIMESTAMP / DATE date)

Purpose: Similar to EXTRACT(), with the argument order reversed. Supports the same date and time units as EXTRACT(). For compatibility with SQL code containing vendor extensions.

Return type: BIGINT

DATE_SUB(TIMESTAMP startdate, INT days), DATE_SUB(TIMESTAMP startdate, interval_expression)

Purpose: Subtracts a specified number of days from a TIMESTAMP value. With an INTERVAL expression as the second argument, you can calculate a delta value using other units such as weeks, years, hours, seconds, and so on.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

Examples:

The following example shows the simplest usage, of subtracting a specified number of days from a TIMESTAMP value:

```
select now() as right_now, date_sub(now(), 7) as last_week;
+-------------------------------+-------------------------------+
```

```
| right_now                     | last_week                     |
+-------------------------------+-------------------------------+
| 2016-05-20 11:21:30.491011000 | 2016-05-13 11:21:30.491011000 |
+-------------------------------+-------------------------------+
```

The following examples show the shorthand notation of an INTERVAL expression, instead of specifying the precise number of days. The INTERVAL notation also lets you work with units smaller than a single day.

```
select now() as right_now, date_sub(now(), interval 3 weeks) as
 3_weeks_ago;
+-------------------------------+-------------------------------+
| right_now                     | 3_weeks_ago                   |
+-------------------------------+-------------------------------+
| 2016-05-20 11:23:05.176953000 | 2016-04-29 11:23:05.176953000 |
+-------------------------------+-------------------------------+
select now() as right_now, date_sub(now(), interval 6 hours) as
 6_hours_ago;
+-------------------------------+-------------------------------+
| right_now                     | 6_hours_ago                   |
+-------------------------------+-------------------------------+
| 2016-05-20 11:23:35.439631000 | 2016-05-20 05:23:35.439631000 |
+-------------------------------+-------------------------------+
```

Like all date/time functions that deal with months, date_add() handles nonexistent dates past the end of a month by setting the date to the last day of the month. The following example shows how the nonexistent date April 31st is normalized to April 30th:

```
select date_sub(cast('2016-05-31' as timestamp), interval 1 mon
ths) as 'april_31st';
+---------------------+
| april_31st          |
+---------------------+
| 2016-04-30 00:00:00 |
+---------------------+
```

DATE_TRUNC(STRING unit, TIMESTAMP / DATE ts)

Purpose: Returns the *ts* value truncated to the specified *unit*.

Argument: The unit argument is not case-sensitive. This argument string can be one of:

Unit	Supported for TIMESTAMP	Supported for DATE
'MICROSECONDS'	Yes	No
'MILLISECONDS'	Yes	No
'SECOND'	Yes	No
'MINUTE'	Yes	No
'HOUR'	Yes	No
'DAY'	Yes	Yes
'WEEK'	Yes	Yes
'MONTH'	Yes	Yes
'YEAR'	Yes	Yes
'DECADE'	Yes	Yes
'CENTURY'	Yes	Yes

Unit	Supported for TIMESTAMP	Supported for DATE
'MILLENNIUM'	Yes	Yes

Usage notes:

Although this function is similar to calling TRUNC() with a TIMESTAMP or DATE argument, the order of arguments and the recognized units are different between TRUNC() and DATE_TRUNC(). Therefore, these functions are not interchangeable.

This function is typically used in GROUP BY queries to aggregate results from the same hour, day, week, month, quarter, and so on. You can also use this function in an INSERT ... SELECT into a partitioned table to divide TIMESTAMP or DATE values into the correct partition.

Return type:

- TIMESTAMP if the second argument, *ts*, is TIMESTAMP.
- DATE if the second argument, *ts*, is DATE.

Examples:

DATE_TRUNC('HOUR', NOW()) returns 2017-12-05 13:00:00.

DATE_TRUNC('MILLENNIUM', DATE'2019-08-02') returns 2001-01-01.

DATEDIFF(TIMESTAMP / DATE enddate, TIMESTAMP / DATE startdate)

Purpose: Returns the number of days from *startdate* to *enddate*.

If *enddate* > *startdate*, the return value is positive.

If *enddate* < *startdate*, the return value is negative.

If *enddate* = *startdate*, the return value is zero.

Return type: INT

Usage notes:

The time portions of the *enddate* and *startdate* values are ignored. For example, 11:59 PM on one day and 12:01 AM on the next day represent a DATEDIFF() of -1 because the date/time values represent different days, even though the TIMESTAMP values differ by only 2 minutes.

DAY(TIMESTAMP / DATE date), DAYOFMONTH(TIMESTAMP / DATE date)

Purpose: Returns the day value from the *date* argument. The value represents the day of the month, therefore is in the range 1-31, or less for months without 31 days.

Returns NULL for nonexistent dates, e.g. Feb 30, or misformatted date strings, e.g. '1999-02-013'.

Return type: INT

DAYNAME(TIMESTAMP / DATE date)

Purpose: Returns the day name of the *date* argument. The range of return values is 'Sunday' to 'Sat urday'. Used in report-generating queries, as an alternative to calling DAYOFWEEK() and turning that numeric return value into a string using a CASE expression.

Return type: STRING

DAYOFWEEK(TIMESTAMP / DATE date)

Purpose: Returns the day field of the *date* arguement, corresponding to the day of the week. The range of return values is 1 (Sunday) to 7 (Saturday).

Return type: INT

DAYOFYEAR(TIMESTAMP / DATE date)

Purpose: Returns the day field from the *date* argument, corresponding to the day of the year. The range of return values is 1 (January 1) to 366 (December 31 of a leap year).

Return type: INT

DAYS_ADD(TIMESTAMP / DATE date, INT / BIGINT days)

Purpose: Returns the value with the number of *days* added to *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

DAYS_SUB(TIMESTAMP / DATE date, INT / BIGINT days)

Purpose: Returns the value with the number of *days* subtracted from *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

EXTRACT(TIMESTAMP / DATE ts, STRING unit), EXTRACT(unit FROM TIMESTAMP / DATE ts)

Purpose: Returns one of the numeric date or time fields, specified by *unit*, from *ts*.

Argument: The unit argument value is not case-sensitive. The unit string can be one of:

Unit	Supported for TIMESTAMP ts	Supported for DATE ts
'EPOCH'	Yes	No
'MILLISECOND'	Yes	No
'SECOND'	Yes	No
'MINUTE'	Yes	No
'HOUR'	Yes	No
'DAY'	Yes	Yes
'MONTH'	Yes	Yes
'QUARTER'	Yes	Yes
'YEAR'	Yes	Yes

Usage notes:

Typically used in GROUP BY queries to arrange results by hour, day, month, and so on. You can also use this function in an INSERT ... SELECT statement to insert into a partitioned table to split up TIMESTAMP values into individual parts, if the partitioned table has separate partition key columns representing year, month, day, and so on. If you need to divide by more complex units of time, such as by week or by quarter, use the TRUNC() function instead.

Return type: BIGINT

Examples:

EXTRACT(DAY FROM DATE'2019-08-17') returns 17.

If you specify 'MILLISECOND' for the *unit* argument, the function returns the seconds component and the milliseconds component.

EXTRACT(CAST('2006-05-12 18:27:28.123456789' AS TIMESTAMP), 'MILLISECOND') returns 28123.

FROM_TIMESTAMP(TIMESTAMP datetime, STRING pattern), FROM_TIMESTAMP(STRING datetime, STRING pattern)

Purpose: Converts a TIMESTAMP value into a string representing the same value.

Return type: STRING

Usage notes:

The FROM_TIMESTAMP() function provides a flexible way to convert TIMESTAMP values into arbitrary string formats for reporting purposes.

Because Impala implicitly converts string values into TIMESTAMP, you can pass date/time values represented as strings (in the standard yyyy-MM-dd HH:mm:ss.SSS format) to this function. The result is a string using different separator characters, order of fields, spelled-out month names, or other variation of the date/time string representation.

The allowed tokens for the pattern string are the same as for the FROM_UNIXTIME() function.

FROM_UNIXTIME(BIGINT unixtime[, STRING pattern])

Purpose: Converts the number of seconds from the Unix epoch to the specified time into a string in the local time zone.

Return type: STRING

The *pattern* string supports the following subset of Java SimpleDateFormat.

Pattern	Description
y	Year
M	Month
d	Day
H	Hour
m	Minute
s	Second
S	Fractional second
+/-hh:mm	Time zone offset
+/-hhmm	Time zone offset
+/-hh	Time zone offset

The following rules apply to the *pattern* string:

- The *pattern* string is case-sensitive.
- All fields are variable length, and thus must use separators to specify the boundaries of the fields, with the exception of the time zone values.
- Time zone offset formats must be at the end of the *pattern* string.
- Formatting character groups can appear in any order along with any separators except for the time zone offset. For example:
  - yyyy/MM/dd
  - dd-MMM-yy
  - (dd)(MM)(yyyy) HH:mm:ss
  - yyyy-MM-dd HH:mm:ss+hh:mm

In Impala 1.3 and later, you can switch the order of elements, use alternative separator characters, and use a different number of placeholders for each unit. Adding more instances of y, d, H, and so on produces output strings zero-padded to the requested number of characters. The exception is M for months, where M produces a non-padded value such as 3, MM produces a zero-padded value such as 03, MMM produces an abbreviated month name such as Mar, and sequences of 4 or more M are not allowed.

A date string including all fields could be 'yyyy-MM-dd HH:mm:ss.SSSSSS', 'dd/MM/yyyy HH:mm:ss.SSSSSS', 'MMM dd, yyyy HH.mm.ss (SSSSSS)' or other combinations of placeholders and separator characters.

Usage notes:

The way this function deals with time zones when converting to or from TIMESTAMP values is affected by the ##use_local_tz_for_unix_timestamp_conversions startup flag for the `impalad` daemon.

FROM_UTC_TIMESTAMP(TIMESTAMP ts, STRING timezone)

Purpose: Converts a specified UTC timestamp value into the appropriate value for a specified time zone.

Return type: TIMESTAMP

Usage notes: Often used to translate UTC time zone data stored in a table back to the local date and time for reporting. The opposite of the TO_UTC_TIMESTAMP() function.

To determine the time zone of the server you are connected to, you can call the TIMEOFDAY() function, which includes the time zone specifier in its return value. Remember that with cloud computing, the server you interact with might be in a different time zone than you are, or different sessions might connect to servers in different time zones, or a cluster might include servers in more than one time zone.

See discussion of time zones in TIMESTAMP data type on page 40 for information about using this function for conversions between the local time zone and UTC.

HOUR(TIMESTAMP ts)

Purpose: Returns the hour field from a TIMESTAMP field.

Return type: INT

HOURS_ADD(TIMESTAMP date, INT hours), HOURS_ADD(TIMESTAMP date, BIGINT hours)

Purpose: Returns the specified date and time plus some number of hours.

Return type: TIMESTAMP

HOURS_SUB(TIMESTAMP date, INT hours), HOURS_SUB(TIMESTAMP date, BIGINT hours)

Purpose: Returns the specified date and time minus some number of hours.

Return type: TIMESTAMP

INT_MONTHS_BETWEEN(TIMESTAMP / DATE enddate, TIMESTAMP / DATE startdate)

Purpose: Returns the number of months from *startdate* to *enddate*, representing only the full months that passed.

Return type: INT

Usage notes:

Typically used in business contexts, for example to determine whether a specified number of months have passed or whether some end-of-month deadline was reached.

The method of determining the number of elapsed months includes some special handling of months with different numbers of days that creates edge cases for dates between the 28th and 31st days of certain months. See MONTHS_BETWEEN() for details. The INT_MONTHS_BETWEEN() result is essentially the FLOOR() of the MONTHS_BETWEEN() result.

If either value is NULL, which could happen for example when converting a nonexistent date string such as '2015-02-29' to a TIMESTAMP, the result is also NULL.

If the first argument represents an earlier time than the second argument, the result is negative.

LAST_DAY(TIMESTAMP / DATE ts)

Purpose: Returns the beginning of the last calendar day in the same month of *ts*.

Return type:

- Returns TIMESTAMP if *ts* is of the TIMESTAMP type.

- Returns DATE if *ts* is of the DATE type.

Usage notes:

If the input argument does not represent a valid Impala TIMESTAMP including both date and time portions, the function returns NULL. For example, if the input argument is a string that cannot be implicitly cast to TIMESTAMP, does not include a date portion, or is out of the allowed range for Impala TIMESTAMP values, the function returns NULL.

MICROSECONDS_ADD(TIMESTAMP date, INT microseconds), MICROSECONDS_ADD(TIMESTAMP date, BIGINT microseconds)

Purpose: Returns the specified date and time plus some number of microseconds.

Return type: TIMESTAMP

MICROSECONDS_SUB(TIMESTAMP date, INT microseconds), MICROSECONDS_SUB(TIMESTAMP date, BIGINT microseconds)

Purpose: Returns the specified date and time minus some number of microseconds.

Return type: TIMESTAMP

MILLISECOND(TIMESTAMP ts)

Purpose: Returns the millisecond portion of a TIMESTAMP value.

Return type: INT

Usage notes:

The millisecond value is truncated, not rounded, if the TIMESTAMP value contains more than 3 significant digits to the right of the decimal point.

MILLISECONDS_ADD(TIMESTAMP date, INT milliseconds), MILLISECONDS_ADD(TIMESTAMP date, BIGINT milliseconds)

Purpose: Returns the specified date and time plus some number of milliseconds.

Return type: TIMESTAMP

MILLISECONDS_SUB(TIMESTAMP date, INT milliseconds), MILLISECONDS_SUB(TIMESTAMP date, BIGINT milliseconds)

Purpose: Returns the specified date and time minus some number of milliseconds.

Return type: TIMESTAMP

MINUTE(TIMESTAMP date)

Purpose: Returns the minute field from a TIMESTAMP value.

Return type: INT

MINUTES_ADD(TIMESTAMP date, INT minutes), MINUTES_ADD(TIMESTAMP date, BIGINT minutes)

Purpose: Returns the specified date and time plus some number of minutes.

Return type: TIMESTAMP

MINUTES_SUB(TIMESTAMP date, INT minutes), MINUTES_SUB(TIMESTAMP date, BIGINT minutes)

Purpose: Returns the specified date and time minus some number of minutes.

Return type: TIMESTAMP

MONTH(TIMESTAMP / DATE date)

Purpose: Returns the month field, represented as an integer, from the *date* argument.

Return type: INT

MONTHNAME(TIMESTAMP / DATE date)

Purpose: Returns the month name of the *date* argument.

Return type: STRING

MONTHS_ADD(TIMESTAMP / DATE date, INT / BIGINT months)

Purpose: Returns the value with the number of *months* added to *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

Usage notes:

If *date* is the last day of a month, the return date will fall on the last day of the target month, e.g. MONTHS_ADD(DATE'2019-01-31', 1) returns DATE'2019-02-28'.

MONTHS_BETWEEN(TIMESTAMP / DATE enddate, TIMESTAMP / DATE startdate)

Purpose: Returns the number of months from *startdate* to *enddate*.

This result can include a fractional part representing extra days in addition to the full months between the dates. The fractional component is computed by dividing the difference in days by 31 (regardless of the month).

Return type: DOUBLE

Usage notes:

Typically used in business contexts, for example to determine whether a specified number of months have passed or whether some end-of-month deadline was reached.

If the only consideration is the number of full months and any fractional value is not significant, use INT_MONTHS_BETWEEN() instead.

The method of determining the number of elapsed months includes some special handling of months with different numbers of days that creates edge cases for dates between the 28th and 31st days of certain months.

If either value is NULL, which could happen for example when converting a nonexistent date string such as '2015-02-29' to a TIMESTAMP, the result is also NULL.

If the first argument represents an earlier time than the second argument, the result is negative.

The time portion of the input arguements are ignored.

MONTHS_SUB(TIMESTAMP / DATE date, INT / BIGINT months)

Purpose: Returns the value with the number of *months* subtracted from *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

Usage notes:

If *date* is the last day of a month, the return date will fall on the last day of the target month, e.g. MONTHS_SUB(DATE'2019-02-28', 1) returns DATE'2019-01-31'.

NANOSECONDS_ADD(TIMESTAMP date, INT nanoseconds), NANOSECONDS_ADD(TIMESTAMP date, BIGINT nanoseconds)

Purpose: Returns the specified date and time plus some number of nanoseconds.

Return type: TIMESTAMP

Kudu considerations:

The nanosecond portion of an Impala TIMESTAMP value is rounded to the nearest microsecond when that value is stored in a Kudu table.

NANOSECONDS_SUB(TIMESTAMP date, INT nanoseconds), NANOSECONDS_SUB(TIMESTAMP date, BIGINT nanoseconds)

Purpose: Returns the specified date and time minus some number of nanoseconds.

Return type: TIMESTAMP

Kudu considerations:

The nanosecond portion of an Impala TIMESTAMP value is rounded to the nearest microsecond when that value is stored in a Kudu table.

NEXT_DAY(TIMESTAMP / DATE date, STRING weekday)

Purpose: Returns the date of the *weekday* that follows the specified *date*.

Argument: The *weekday* is not case-sensitive.

The following values are accepted for *weekday*: "Sunday"/"Sun", "Monday"/"Mon", "Tue sday"/"Tue", "Wednesday"/"Wed", "Thursday"/"Thu", "Friday"/"Fri", "Saturday"/"Sat"

Return type:

- Returns TIMESTAMP if *date* is of the TIMESTAMP type.
- Returns DATE if *date* is of the DATE type.

Examples:

NEXT_DAY('2013-12-25','Saturday') returns '2013-12-28 00:00:00' which is the first Saturday after December 25, 2013.

NOW()

Purpose: Returns the current date and time (in the local time zone) as a TIMESTAMP value.

Return type: TIMESTAMP

Usage notes:

To find a date/time value in the future or the past relative to the current date and time, add or subtract an INTERVAL expression to the return value of NOW().

To produce a TIMESTAMP representing the current date and time that can be shared or stored without interoperability problems due to time zone differences, use the TO_UTC_TIMESTAMP() function and specify the time zone of the server. When TIMESTAMP data is stored in UTC form, any application that queries those values can convert them to the appropriate local time zone by calling the inverse function, FROM_UTC_TIMESTAMP().

To determine the time zone of the server you are connected to, you can call the timeofday() function, which includes the time zone specifier in its return value. Remember that with cloud computing, the server you interact with might be in a different time zone than you are, or different sessions might connect to servers in different time zones, or a cluster might include servers in more than one time zone.

Any references to the NOW() function are evaluated at the start of a query. All calls to NOW() within the same query return the same value, and the value does not depend on how long the query takes.

QUARTER(TIMESTAMP / DATE date)

Purpose: Returns the quarter in the input *date* argument as an integer value, 1, 2, 3, or 4, where 1 represents January 1 through March 31.

Return type: INT

SECOND(TIMESTAMP date)

Purpose: Returns the second field from a TIMESTAMP value.

Return type: INT

SECONDS_ADD(TIMESTAMP date, INT seconds), SECONDS_ADD(TIMESTAMP date, BIGINT seconds)

Purpose: Returns the specified date and time plus some number of seconds.

Return type: TIMESTAMP

SECONDS_SUB(TIMESTAMP date, INT seconds), SECONDS_SUB(TIMESTAMP date, BIGINT seconds)

Purpose: Returns the specified date and time minus some number of seconds.

Return type: TIMESTAMP

SUBDATE(TIMESTAMP / DATE date, INT / BIGINT days)

Purpose: Subtracts *days* from *date* and returns the new date value.

The *days* value can be negative, which gives the same result as the ADDDATE() function.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

TIMEOFDAY()

Purpose: Returns a string representation of the current date and time, according to the time of the local system, including any time zone designation.

Return type: STRING

Usage notes:

The result value represents similar information as the now() function, only as a STRING type and with somewhat different formatting. For example, the day of the week and the time zone identifier are included. This function is intended primarily for compatibility with SQL code from other systems that also have a timeofday() function. Prefer to use now() if practical for any new Impala code.

TIMESTAMP_CMP(TIMESTAMP t1, TIMESTAMP t2)

Purpose: Tests if one TIMESTAMP value is newer than, older than, or identical to another TIME STAMP

- If the first argument represents a later point in time than the second argument, the result is 1.
- If the first argument represents an earlier point in time than the second argument, the result is -1.
- If the first and second arguments represent identical points in time, the result is 0.
- If either argument is NULL, the result is NULL.

Return type: INT (either -1, 0, 1, or NULL)

Usage notes:

A comparison function for TIMESTAMP values that only tests whether the date and time increases, decreases, or stays the same. Similar to the SIGN() function for numeric values.

TO_DATE(TIMESTAMP ts)

Purpose: Returns a string representation of the date field from the *ts* argument.

Return type: STRING

TO_TIMESTAMP(BIGINT unixtime), TO_TIMESTAMP(STRING date, STRING pattern)

Purpose: Converts an integer or string representing a date/time value into the corresponding TIME STAMP value.

Return type: TIMESTAMP

Usage notes:

An integer argument represents the number of seconds past the epoch (midnight on January 1, 1970). It is the converse of the UNIX_TIMESTAMP() function, which produces a BIGINT representing the number of seconds past the epoch.

A string argument, plus another string argument representing the pattern, turns an arbitrary string representation of a date and time into a true TIMESTAMP value. The ability to parse many kinds of date and time formats allows you to deal with temporal data from diverse sources, and if desired to convert to efficient TIMESTAMP values during your ETL process. Using TIMESTAMP directly in queries and expressions lets you perform date and time calculations without the overhead of extra function calls and conversions each time you reference the applicable columns.

Examples:

The following examples demonstrate how to convert an arbitrary string representation to TIME STAMP based on a pattern string:

```
select to_timestamp('Sep 25, 1984', 'MMM dd, yyyy');
+-----------------------------------------------+
| to_timestamp('sep 25, 1984', 'mmm dd, yyyy') |
+-----------------------------------------------+
| 1984-09-25 00:00:00                           |
+-----------------------------------------------+

select to_timestamp('1984/09/25', 'yyyy/MM/dd');
+-----------------------------------------+
| to_timestamp('1984/09/25', 'yyyy/mm/dd') |
+-----------------------------------------+
| 1984-09-25 00:00:00                     |
+-----------------------------------------+
```

The following examples show how to convert a BIGINT representing seconds past epoch into a TIMESTAMP value:

```
-- One day past the epoch.
select to_timestamp(24 * 60 * 60);
+----------------------------+
| to_timestamp(24 * 60 * 60) |
+----------------------------+
| 1970-01-02 00:00:00        |
+----------------------------+

-- 60 seconds in the past.
select now() as 'current date/time',
  unix_timestamp(now()) 'now in seconds',
  to_timestamp(unix_timestamp(now()) - 60) as '60 seconds ago';
+-------------------------------+----------------+-------------
--------+
| current date/time             | now in seconds | 60 seconds
ago      |
+-------------------------------+----------------+--------------
------+
| 2017-10-01 22:03:46.885624000 | 1506895426     | 2017-10-01 22
:02:46 |
+-------------------------------+----------------+-------------
--------+
```

TO_UTC_TIMESTAMP(TIMESTAMP ts, STRING timezone)

Purpose: Converts a specified timestamp value in a specified time zone into the corresponding value for the UTC time zone.

Return type: TIMESTAMP

Usage notes:

Often used in combination with the now() function, to translate local date and time values to the UTC time zone for consistent representation on disk. The opposite of the FROM_UTC_TIMESTAMP() function.

See discussion of time zones in TIMESTAMP data type on page 40 for information about using this function for conversions between the local time zone and UTC.

Examples:

The simplest use of this function is to turn a local date/time value to one with the standardized UTC time zone. Because the time zone specifier is not saved as part of the Impala TIMESTAMP value, all applications that refer to such data must agree in advance which time zone the values represent. If different parts of the ETL cycle, or different instances of the application, occur in different time zones, the ideal reference point is to convert all TIMESTAMP values to UTC for storage.

```
select now() as 'Current time in California USA',
  to_utc_timestamp(now(), 'PDT') as 'Current time in Greenwich UK
';
+---------------------------------+-------------------------------
+
| current time in california usa | current time in greenwich uk
 |
+---------------------------------+-------------------------------
+
| 2016-06-01 15:52:08.980072000  | 2016-06-01 22:52:08.980072000
 |
+---------------------------------+-------------------------------
+
```

Once a value is converted to the UTC time zone by TO_UTC_TIMESTAMP(), it can be converted back to the local time zone with FROM_UTC_TIMESTAMP(). You can combine these functions using different time zone identifiers to convert a TIMESTAMP between any two time zones. This example starts with a TIMESTAMP value representing Pacific Daylight Time, converts it to UTC, and converts it to the equivalent value in Eastern Daylight Time.

```
select now() as 'Current time in California USA',
  from_utc_timestamp
  (
    to_utc_timestamp(now(), 'PDT'),
    'EDT'
  ) as 'Current time in New York, USA';
+---------------------------------+-------------------------------
+
| current time in california usa | current time in new york, usa
 |
+---------------------------------+-------------------------------
+
| 2016-06-01 18:14:12.743658000  | 2016-06-01 21:14:12.743658000
|
+---------------------------------+-------------------------------
+
```

TRUNC(TIMESTAMP / DATE ts, STRING unit)

Purpose: Returns the *ts* truncated to the *unit* specified.

Argument: The *unit* argument is not case-sensitive. This argument string can be one of:

Unit	Supported for TIMESTAMP ts	Supported for DATE ts	Description
'SYYYY' 'YYYY' 'YEAR' 'SYEAR' 'YYY' 'YY' 'Y'	Yes	Yes	Year
'Q'	Yes	Yes	Quarter
'MONTH' 'MON' 'MM' 'RM'	Yes	Yes	Month
'WW'	Yes	Yes	The most recent date that is the same day of the week as the first day of the year
'W'	Yes	Yes	The most recent date that is the same day of the week as the first day of the month
'DDD' 'DD' 'J'	Yes	Yes	Day
'DAY' 'DY' 'D'	Yes	Yes	Starting day of the week (Monday)
'HH' 'HH12' 'HH24'	Yes	No	Hour. A TIMESTAMP value truncated to the hour is always represented in 24-hour notation, even for the HH12 argument string.
'MI'	Yes	No	Minute

Usage notes:

The TIMESTAMP form is typically used in GROUP BY queries to aggregate results from the same hour, day, week, month, quarter, and so on. You can also use this function in an INSERT ... S ELECT into a partitioned table to divide TIMESTAMP values into the correct partition.

Return type:

- TIMESTAMP if the first argument, *ts*, is TIMESTAMP.
- DATE if the first argument, *ts*, is DATE.

Example:

TRUNC(DATE'2019-05-08','YEAR') returns 2019-01-01.

TRUNC(DATE'2019-05-08', 'QUARTER') returns 2019-04-01.

UNIX_TIMESTAMP(), UNIX_TIMESTAMP(STRING datetime), UNIX_TIMESTAMP(STRING datetime, STRING pattern), UNIX_TIMESTAMP(TIMESTAMP datetime)

Purpose: Returns a Unix time, which is a number of seconds elapsed since '1970-01-01 00:00:00' UTC. If called with no argument, the current date and time is converted to its Unix time. If called with arguments, the first argument represented as the TIMESTAMP or STRING is converted to its Unix time.

Return type: BIGINT

Usage notes:

See FROM_UNIXTIME() for details about the patterns you can use in the *pattern* string to represent the position of year, month, day, and so on in the date string. In Impala 1.3 and higher, you have more flexibility to switch the positions of elements and use different separator characters.

You can include a trailing uppercase Z qualifier to indicate “Zulu” time, a synonym for UTC.

You can include a timezone offset specified as minutes and hours, provided you also specify the details in the *pattern* string argument. The offset is specified in the *pattern* string as a plus or minus sign followed by hh:mm, hhmm, or hh. The hh must be lowercase, to distinguish it from the HH represent hours in the actual time value. Currently, only numeric timezone offsets are allowed, not symbolic names.

Built-in functions that accept or return integers representing TIMESTAMP values use the BIGINT type for parameters and return values, rather than INT. This change lets the date and time functions avoid an overflow error that would otherwise occur on January 19th, 2038 (known as the “Year 2038 problem” or “Y2K38 problem”). This change affects the FROM_UNIXTIME() and UNIX_TIMESTAMP() functions. You might need to change application code that interacts with these functions, change the types of columns that store the return values, or add CAST() calls to SQL statements that call these functions.

UNIX_TIMESTAMP() and FROM_UNIXTIME() are often used in combination to convert a TIMESTAMP value into a particular string format. For example, FROM_UNIXTIME(UNIX_TIMESTAMP(NOW() + interval 3 days), 'yyyy/MM/dd HH:mm').

The way this function deals with time zones when converting to or from TIMESTAMP values is affected by the ##use_local_tz_for_unix_timestamp_conversions startup flag for the `impalad` daemon. See TIMESTAMP data type on page 40 for details about how Impala handles time zone considerations for the TIMESTAMP data type.

Examples:

The following examples show different ways of turning the same date and time into an integer value. A *pattern* string that Impala recognizes by default is interpreted as a UTC date and time. The trailing Z is a confirmation that the timezone is UTC. If the date and time string is formatted differently, a second argument specifies the position and units for each of the date and time values.

The final two examples show how to specify a timezone offset of Pacific Daylight Saving Time, which is 7 hours earlier than UTC. You can use the numeric offset -07:00 and the equivalent suffix of -hh:mm in the *pattern* string, or specify the mnemonic name for the time zone in a call to TO_UTC_TIMESTAMP(). This particular date and time expressed in PDT translates to a different number than the same date and time expressed in UTC.

```
-- 3 ways of expressing the same date/time in UTC and converting
 to an integer.

select unix_timestamp('2015-05-15 12:00:00');
+-----------------------------------+
| unix_timestamp('2015-05-15 12:00:00') |
+-----------------------------------+
```

```
| 1431691200                              |
+-----------------------------------------+

select unix_timestamp('2015-05-15 12:00:00Z');
+-----------------------------------------+
| unix_timestamp('2015-05-15 12:00:00z') |
+-----------------------------------------+
| 1431691200                              |
+-----------------------------------------+

select unix_timestamp
(
  'May 15, 2015 12:00:00',
  'MMM dd, yyyy HH:mm:ss'
) as may_15_month_day_year;
+-----------------------+
| may_15_month_day_year |
+-----------------------+
| 1431691200            |
+-----------------------+

-- 2 ways of expressing the same date and time but in a differ
ent timezone.
-- The resulting integer is different from the previous examples.
select unix_timestamp
(
  '2015-05-15 12:00:00-07:00',
  'yyyy-MM-dd HH:mm:ss-hh:mm'
) as may_15_year_month_day;
+-----------------------+
| may_15_year_month_day |
+-----------------------+
| 1431716400            |
+-----------------------+
select unix_timestamp
  (to_utc_timestamp(
    '2015-05-15 12:00:00',
    'PDT')
  ) as may_15_pdt;
+------------+
| may_15_pdt |
+------------+
| 1431716400 |
+------------+
```

UTC_TIMESTAMP()

Purpose: Returns a TIMESTAMP corresponding to the current date and time in the UTC time zone.

Return type: TIMESTAMP

Examples:

Similar to the NOW() or CURRENT_TIMESTAMP() functions, but does not use the local time zone as those functions do. Use UTC_TIMESTAMP() to record TIMESTAMP values that are interoperable with servers around the world, in arbitrary time zones, without the need for additional conversion functions to standardize the time zone of each value representing a date/time.

For working with date/time values represented as integer values, you can convert back and forth between TIMESTAMP and BIGINT with the UNIX_MICROS_TO_UTC_TIMESTAMP() and UTC_TO_UNIX_MICROS() functions. The integer values represent the number of microseconds since the Unix epoch (midnight on January 1, 1970).

Examples:

The following example shows how NOW() and CURRENT_TIMESTAMP() represent the current date/time in the local time zone (in this case, UTC-7), while utc_timestamp() represents the same date/time in the standardized UTC time zone:

```
select now(), utc_timestamp();
+-------------------------------+-------------------------------+
| now()                         | utc_timestamp()               |
+-------------------------------+-------------------------------+
| 2017-10-01 23:33:58.919688000 | 2017-10-02 06:33:58.919688000 |
+-------------------------------+-------------------------------+

select current_timestamp(), utc_timestamp();
+-------------------------------+-------------------------------+
| current_timestamp()           | utc_timestamp()               |
+-------------------------------+-------------------------------+
| 2017-10-01 23:34:07.400642000 | 2017-10-02 06:34:07.400642000 |
+-------------------------------+-------------------------------+
```

WEEK(TIMESTAMP / DATE date), WEEKOFYEAR(TIMESTAMP / DATE date)

Purpose: Returns the corresponding week (1-53) from the *date* argument.

Return type: INT

WEEKS_ADD(TIMESTAMP / DATE date, INT / BIGINT weeks)

Purpose: Returns the value with the number of *weeks* added to *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

WEEKS_SUB(TIMESTAMP / DATE date, INT / BIGINT weeks)

Purpose: Returns the value with the number of *weeks* subtracted from *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

YEAR(TIMESTAMP / DATE date)

Purpose: Returns the year field from the *date* argument.

Return type: INT

YEARS_ADD(TIMESTAMP / DATE date, INT / BIGINT years)

Purpose: Returns the value with the number of *years* added to *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.
- If *date* is DATE, returns DATE.

Usage notes:

If the equivalent date does not exist in the year of the result due to a leap year, the date is changed to the last day of the appropriate month.

YEARS_SUB(TIMESTAMP / DATE date, INT / BIGINT years)

Purpose: Returns the value with the number of *years* subtracted from *date*.

Return type:

- If *date* is TIMESTAMP, returns TIMESTAMP.

- If *date* is DATE, returns DATE.

Usage notes:

If the equivalent date does not exist in the year of the result due to a leap year, the date is changed to the last day of the appropriate month.

Impala conditional functions

Impala supports the following conditional functions for testing equality, comparison operators, and nullity.

- CASE
- CASE2
- COALESCE
- DECODE
- IF
- IFNULL
- ISFALSE
- ISNOTFALSE
- ISNOTTRUE
- ISNULL
- ISTRUE
- NONNULLVALUE
- NULLIF
- NULLIFZERO
- NULLVALUE
- NVL
- NVL2
- ZEROIFNULL

CASE a WHEN b THEN c [WHEN d THEN e]... [ELSE f] END

Purpose: Compares an expression to one or more possible values, and returns a corresponding result when a match is found.

Return type: same as the initial argument value, except that integer values are promoted to BIGI NT and floating-point values are promoted to DOUBLE; use CAST() when inserting into a smaller numeric column

Usage notes:

In this form of the CASE expression, the initial value A being evaluated for each row it typically a column reference, or an expression involving a column. This form can only compare against a set of specified values, not ranges, multi-value comparisons such as BETWEEN or IN, regular expressions, or NULL.

Examples:

Although this example is split across multiple lines, you can put any or all parts of a CASE expression on a single line, with no punctuation or other separators between the WHEN, ELSE, and END clauses.

```
select case x
    when 1 then 'one'
    when 2 then 'two'
    when 0 then 'zero'
    else 'out of range'
  end
    from t1;
```

CASE WHEN a THEN b [WHEN c THEN d]... [ELSE e] END

Purpose: Tests whether any of a sequence of expressions is TRUE, and returns a corresponding result for the first TRUE expression.

Return type: same as the initial argument value, except that integer values are promoted to BIGI NT and floating-point values are promoted to DOUBLE; use CAST() when inserting into a smaller numeric column

Usage notes:

CASE expressions without an initial test value have more flexibility. For example, they can test different columns in different WHEN clauses, or use comparison operators such as BETWEEN, IN and IS NULL rather than comparing against discrete values.

CASE expressions are often the foundation of long queries that summarize and format results for easy-to-read reports. For example, you might use a CASE function call to turn values from a numeric column into category strings corresponding to integer values, or labels such as “Small”, “Medium” and “Large” based on ranges. Then subsequent parts of the query might aggregate based on the transformed values, such as how many values are classified as small, medium, or large. You can also use CASE to signal problems with out-of-bounds values, NULL values, and so on.

By using operators such as OR, IN, REGEXP, and so on in CASE expressions, you can build extensive tests and transformations into a single query. Therefore, applications that construct SQL statements often rely heavily on CASE calls in the generated SQL code.

Because this flexible form of the CASE expressions allows you to perform many comparisons and call multiple functions when evaluating each row, be careful applying elaborate CASE expressions to queries that process large amounts of data. For example, when practical, evaluate and transform values through CASE after applying operations such as aggregations that reduce the size of the result set; transform numbers to strings after performing joins with the original numeric values.

Examples:

Although this example is split across multiple lines, you can put any or all parts of a CASE expression on a single line, with no punctuation or other separators between the WHEN, ELSE, and END clauses.

```
select case
    when dayname(now()) in ('Saturday','Sunday') then 'result un
defined on weekends'
    when x > y then 'x greater than y'
    when x = y then 'x and y are equal'
    when x is null or y is null then 'one of the columns is null'
    else null
  end
    from t1;
```

COALESCE(type v1, type v2, ...)

Purpose: Returns the first specified argument that is not NULL, or NULL if all arguments are NULL.

Return type: same as the initial argument value, except that integer values are promoted to BIGI NT and floating-point values are promoted to DOUBLE; use CAST() when inserting into a smaller numeric column

DECODE(type expression, type search1, type result1 [, type search2, type result2 ...] [, type default])

Purpose: Compares the first argument, expression, to the search expressions using the IS NOT D ISTINCT operator, and returns:

- The corresponding result when a match is found.
- The first corresponding result if there are more than one matching search expressions.
- The default expression if none of the search expressions matches the first argument expression.

- NULL if the final default expression is omitted and none of the search expressions matches the first argument.

Return type: Same as the first argument with the following exceptions:

- Integer values are promoted to BIGINT.
- Floating-point values are promoted to DOUBLE.
- Use CAST() when inserting into a smaller numeric column.

Usage notes:

- Can be used as shorthand for a CASE expression.
- The first argument, expression, and the search expressions must be of the same type or convertible types.
- The result expression can be a different type, but all result expressions must be of the same type.
- Returns a successful match if the first argument is NULL and a search expression is also NULL.
- NULL can be used as a search expression.

Examples:

The following example translates numeric day values into weekday names, such as 1 to Monday, 2 to Tuesday, etc.

```
SELECT event, DECODE(day_of_week, 1, "Monday", 2, "Tuesday", 3,
"Wednesday",
  4, "Thursday", 5, "Friday", 6, "Saturday", 7, "Sunday", "Unkn
own day")
  FROM calendar;
```

IF(BOOLEAN condition, type ifTrue, type ifFalseOrNull)

Purpose: Tests an expression and returns a corresponding result depending on whether the result is TRUE, FALSE, or NULL.

Return type: Same as the ifTrue argument value

IFNULL(type a, type ifNull)

Purpose: Alias for the ISNULL() function, with the same behavior. To simplify porting SQL with vendor extensions to Impala.

Added in: Impala 1.3.0

ISFALSE(BOOLEAN expression)

Purpose: Returns TRUE if the expression is FALSE. Returns FALSE if the expression is TRUE or NULL.

Same as the IS FALSE operator.

Similar to ISNOTTRUE(), except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Usage notes:

In Impala 2.1.1 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE(), ISNOTTRUE(), ISFALSE(), and ISNOTFALSE().

ISNOTFALSE(BOOLEAN expression)

Purpose: Tests if a Boolean expression is not FALSE (that is, either TRUE or NULL). Returns TRUE if so. If the argument is NULL, returns TRUE.

Same as the IS NOT FALSE operator.

Similar to ISTRUE(), except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

Usage notes:

In Impala 2.1.1 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE(), ISNOTTRUE(), ISFALSE(), and ISNOTFALSE().

ISNOTTRUE(BOOLEAN expression)

Purpose: Tests if a Boolean expression is not TRUE (that is, either FALSE or NULL). Returns TRUE if so. If the argument is NULL, returns TRUE.

Same as the IS NOT TRUE operator.

Similar to ISFALSE(), except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Usage notes:

In Impala 2.1.1 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE(), ISNOTTRUE(), ISFALSE(), and ISNOTFALSE().

ISNULL(type a, type ifNull)

Purpose: Tests if an expression is NULL, and returns the expression result value if not. If the first argument is NULL, returns the second argument.

Compatibility notes: Equivalent to the NVL() function from Oracle Database or IFNULL() from MySQL. The NVL() and IFNULL() functions are also available in Impala.

Return type: Same as the first argument value

ISTRUE(BOOLEAN expression)

Purpose: Returns TRUE if the expression is TRUE. Returns FALSE if the expression is FALSE or NULL.

Same as the IS TRUE operator.

Similar to ISNOTFALSE(), except it returns the opposite value for a NULL argument.

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

Usage notes:

In Impala 2.1.1 and higher, you can use the operators IS [NOT] TRUE and IS [NOT] FALSE as equivalents for the built-in functions ISTRUE(), ISNOTTRUE(), ISFALSE(), and ISNOTFALSE().

NONNULLVALUE(type expression)

Purpose: Tests if an expression (of any type) is NULL or not. Returns FALSE if so. The converse of nullvalue().

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

NULLIF(expr1, expr2)

Purpose: Returns NULL if the two specified arguments are equal. If the specified arguments are not equal, returns the value of *expr1*. The data types of the expressions must be compatible, according to the conversion rules from Impala SQL data types on page 11. You cannot use an expression that evaluates to NULL for *expr1*; that way, you can distinguish a return value of NULL from an argument value of NULL, which would never match *expr2*.

Usage notes: This function is effectively shorthand for a CASE expression of the form:

```
CASE
  WHEN expr1 = expr2 THEN NULL
  ELSE expr1
END
```

It is commonly used in division expressions, to produce a NULL result instead of a divide-by-zero error when the divisor is equal to zero:

```
select 1.0 / nullif(c1,0) as reciprocal from t1;
```

You might also use it for compatibility with other database systems that support the same NULLIF() function.

Return type: same as the initial argument value, except that integer values are promoted to BIGI NT and floating-point values are promoted to DOUBLE; use CAST() when inserting into a smaller numeric column

NULLIFZERO(numeric_expr)

Purpose: Returns NULL if the numeric expression evaluates to 0, otherwise returns the result of the expression.

Usage notes: Used to avoid error conditions such as divide-by-zero in numeric calculations. Serves as shorthand for a more elaborate CASE expression, to simplify porting SQL with vendor extensions to Impala.

Return type: Same type as the input argument

NULLVALUE(expression)

Purpose: Tests if an expression (of any type) is NULL or not. Returns TRUE if so. The converse of nonnullvalue().

Return type: BOOLEAN

Usage notes: Primarily for compatibility with code containing industry extensions to SQL.

NVL(type a, type ifNull)

Purpose: Alias for the ISNULL() function. Returns the first argument if the first argument is not NULL. Returns the second argument if the first argument is NULL.

Equivalent to the NVL() function in Oracle Database or IFNULL() in MySQL.

Return type: Same as the first argument value

NVL2(type a, type ifNotNull, type ifNull)

Purpose: Returns the second argument, ifNotNull, if the first argument is not NULL. Returns the third argument, ifNull, if the first argument is NULL.

Equivalent to the NVL2() function in Oracle Database.

Return type: Same as the first argument value

Examples:

```
SELECT NVL2(NULL, 999, 0); -- Returns 0
SELECT NVL2('ABC', 'Is Not Null', 'Is Null'); -- Returns 'Is Not
 Null'
```

ZEROIFNULL(numeric_expr)

Purpose: Returns 0 if the numeric expression evaluates to NULL, otherwise returns the result of the expression.

Usage notes: Used to avoid unexpected results due to unexpected propagation of NULL values in numeric calculations. Serves as shorthand for a more elaborate CASE expression, to simplify porting SQL with vendor extensions to Impala.

Return type: Same type as the input argument

Impala string functions

String functions are classified as those primarily accepting or returning STRING, VARCHAR, or CHAR data types, for example to measure the length of a string or concatenate two strings together.

- All the functions that accept STRING arguments also accept the VARCHAR and CHAR types introduced in Impala 2.0.
- Whenever VARCHAR or CHAR values are passed to a function that returns a string value, the return type is normalized to STRING. For example, a call to concat() with a mix of STRING, VARCHAR, and CHAR arguments produces a STRING result.

Function reference:

Impala supports the following string functions:

- ASCII
- BASE64DECODE
- BASE64ENCODE
- BTRIM
- CHAR_LENGTH
- CHR
- CONCAT
- CONCAT_WS
- FIND_IN_SET
- GROUP_CONCAT
- INITCAP
- INSTR
- JARO_DISTANCE, JARO_DIST
- JARO_SIMILARITY, JARO_SIM
- JARO_WINKER_DISTANCE, JW_DST
- JARO_WINKER_SIMILARITY, JW_SIM
- LEFT
- LENGTH
- LEVENSHTEIN, LE_DST
- LOCATE
- LOWER, LCASE
- LPAD
- LTRI
- PARSE_URL
- REGEXP_ESCAPE
- REGEXP_EXTRACT
- REGEXP_LIKE
- REGEXP_REPLACE
- REPEAT
- REPLACE
- REVERSE
- RIGHT
- RPAD
- RTRIM

- SPACE
- SPLIT_PART
- STRLEFT
- STRRIGHT
- SUBSTR, SUBSTRING
- TRANSLATE
- TRIM
- UPPER, UCASE

ASCII(STRING str)

Purpose: Returns the numeric ASCII code of the first character of the argument.

Return type: INT

BASE64DECODE(STRING str)

Purpose:

Return type: STRING

Usage notes:

The functions BASE64ENCODE() and BASE64DECODE() are typically used in combination, to store in an Impala table string data that is problematic to store or transmit. For example, you could use these functions to store string data that uses an encoding other than UTF-8, or to transform the values in contexts that require ASCII values, such as for partition key columns. Keep in mind that base64-encoded values produce different results for string functions such as LENGTH(), MAX(), and MIN() than when those functions are called with the unencoded string values.

The set of characters that can be generated as output from BASE64ENCODE(), or specified in the argument string to BASE64DECODE(), are the ASCII uppercase and lowercase letters (A-Z, a-z), digits (0-9), and the punctuation characters +, /, and =.

All return values produced by BASE64ENCODE() are a multiple of 4 bytes in length. All argument values supplied to BASE64DECODE() must also be a multiple of 4 bytes in length. If a base64-encoded value would otherwise have a different length, it can be padded with trailing = characters to reach a length that is a multiple of 4 bytes.

If the argument string to BASE64DECODE() does not represent a valid base64-encoded value, subject to the constraints of the Impala implementation such as the allowed character set, the function returns NULL.

Examples:

The following examples show how to use BASE64ENCODE() and BASE64DECODE() together to store and retrieve string values:

```
-- An arbitrary string can be encoded in base 64.
-- The length of the output is a multiple of 4 bytes,
-- padded with trailing = characters if necessary.
select base64encode('hello world') as encoded,
  length(base64encode('hello world')) as length;
+------------------+--------+
| encoded          | length |
+------------------+--------+
| aGVsbG8gd29ybGQ= | 16     |
+------------------+--------+

-- Passing an encoded value to base64decode() produces
-- the original value.
select base64decode('aGVsbG8gd29ybGQ=') as decoded;
+-------------+
```

```
| decoded     |
+-------------+
| hello world |
+-------------+
```

These examples demonstrate incorrect encoded values that produce NULL return values when decoded:

```
-- The input value to base64decode() must be a multiple of 4 by
tes.
-- In this case, leaving off the trailing = padding character
-- produces a NULL return value.
select base64decode('aGVsbG8gd29ybGQ') as decoded;
+---------+
| decoded |
+---------+
| NULL    |
+---------+
WARNINGS: UDF WARNING: Invalid base64 string; input length is 15,
  which is not a multiple of 4.

-- The input to base64decode() can only contain certain charac
ters.
-- The $ character in this case causes a NULL return value.
select base64decode('abc$');
+----------------------+
| base64decode('abc$') |
+----------------------+
| NULL                 |
+----------------------+
WARNINGS: UDF WARNING: Could not base64 decode input in space 4;
 actual output length 0
```

These examples demonstrate “round-tripping” of an original string to an encoded string, and back again. This technique is applicable if the original source is in an unknown encoding, or if some intermediate processing stage might cause national characters to be misrepresented:

```
select 'circumflex accents: â, ê, î, ô, û' as original,
  base64encode('circumflex accents: â, ê, î, ô, û') as encoded;
+-----------------------------------+--------------------------
----------------------------+
| original                          | encoded
                          |
+-----------------------------------+--------------------------
--------------------------+
| circumflex accents: â, ê, î, ô, û | Y2lyY3VtZmxleCBhY2NlbnRzOi
DDoiwgw6osIMOuLCDDtCwgw7s= |
+-----------------------------------+--------------------------
----------------------------+

select base64encode('circumflex accents: â, ê, î, ô, û') as encod
ed,
  base64decode(base64encode('circumflex accents: â, ê, î, ô, û'))
 as decoded;
+-------------------------------------------------------+------
----------------------------+
| encoded                                               | decoded
                          |
+-------------------------------------------------------+--------
--------------------------+
```

```
| Y2lyY3VtZmxleCBhY2NlbnRzOiDDoiwgw6osIMOuLCDDtCwgw7s= | circum
flex accents: â, ê, î, ô, û |
+-----------------------------------------------------+------
---------------------------+
```

BASE64ENCODE(STRING str)

Purpose:

Return type: STRING

Usage notes:

The functions BASE64ENCODE() and BASE64DECODE() are typically used in combination, to store in an Impala table string data that is problematic to store or transmit. For example, you could use these functions to store string data that uses an encoding other than UTF-8, or to transform the values in contexts that require ASCII values, such as for partition key columns. Keep in mind that base64-encoded values produce different results for string functions such as LENGTH(), MAX(), and MIN() than when those functions are called with the unencoded string values.

The set of characters that can be generated as output from BASE64ENCODE(), or specified in the argument string to BASE64DECODE(), are the ASCII uppercase and lowercase letters (A-Z, a-z), digits (0-9), and the punctuation characters +, /, and =.

All return values produced by BASE64ENCODE() are a multiple of 4 bytes in length. All argument values supplied to BASE64DECODE() must also be a multiple of 4 bytes in length. If a base64-encoded value would otherwise have a different length, it can be padded with trailing = characters to reach a length that is a multiple of 4 bytes.

Examples:

The following examples show how to use BASE64ENCODE() and BASE64DECODE() together to store and retrieve string values:

```
-- An arbitrary string can be encoded in base 64.
-- The length of the output is a multiple of 4 bytes,
-- padded with trailing = characters if necessary.
select base64encode('hello world') as encoded,
  length(base64encode('hello world')) as length;
+------------------+--------+
| encoded          | length |
+------------------+--------+
| aGVsbG8gd29ybGQ= | 16     |
+------------------+--------+

-- Passing an encoded value to base64decode() produces
-- the original value.
select base64decode('aGVsbG8gd29ybGQ=') as decoded;
+-------------+
| decoded     |
+-------------+
| hello world |
+-------------+
```

These examples demonstrate incorrect encoded values that produce NULL return values when decoded:

```
-- The input value to base64decode() must be a multiple of 4 by
tes.
-- In this case, leaving off the trailing = padding character
-- produces a NULL return value.
select base64decode('aGVsbG8gd29ybGQ') as decoded;
```

```
+---------+
| decoded |
+---------+
| NULL    |
+---------+
WARNINGS: UDF WARNING: Invalid base64 string; input length is 15,
  which is not a multiple of 4.

-- The input to base64decode() can only contain certain charac
ters.
-- The $ character in this case causes a NULL return value.
select base64decode('abc$');
+----------------------+
| base64decode('abc$') |
+----------------------+
| NULL                 |
+----------------------+
WARNINGS: UDF WARNING: Could not base64 decode input in space 4;
 actual output length 0
```

These examples demonstrate “round-tripping” of an original string to an encoded string, and back again. This technique is applicable if the original source is in an unknown encoding, or if some intermediate processing stage might cause national characters to be misrepresented:

```
select 'circumflex accents: â, ê, î, ô, û' as original,
  base64encode('circumflex accents: â, ê, î, ô, û') as encoded;
+----------------------------------+--------------------------
----------------------------+
| original                         | encoded
                           |
+----------------------------------+--------------------------
--------------------------+
| circumflex accents: â, ê, î, ô, û | Y2lyY3VtZmxleCBhY2NlbnRzOi
DDoiwgw6osIMOuLCDDtCwgw7s= |
+----------------------------------+--------------------------
----------------------------+

select base64encode('circumflex accents: â, ê, î, ô, û') as encod
ed,
  base64decode(base64encode('circumflex accents: â, ê, î, ô, û'))
 as decoded;
+------------------------------------------------------+------
----------------------------+
| encoded                                                | decoded
                           |
+------------------------------------------------------+--------
--------------------------+
| Y2lyY3VtZmxleCBhY2NlbnRzOiDDoiwgw6osIMOuLCDDtCwgw7s= | circum
flex accents: â, ê, î, ô, û |
+------------------------------------------------------+------
----------------------------+
```

BTRIM(STRING a), BTRIM(STRING a, STRING chars_to_trim)

Purpose: Removes all instances of one or more characters from the start and end of a STRING value. By default, removes only spaces. If a non-NULL optional second argument is specified, the function removes all occurrences of characters in that second argument from the beginning and end of the string.

Return type: STRING

Examples:

The following examples show the default btrim() behavior, and what changes when you specify the optional second argument. All the examples bracket the output value with [ ] so that you can see any leading or trailing spaces in the btrim() result. By default, the function removes and number of both leading and trailing spaces. When the second argument is specified, any number of occurrences of any character in the second argument are removed from the start and end of the input string; in this case, spaces are not removed (unless they are part of the second argument) and any instances of the characters are not removed if they do not come right at the beginning or end of the string.

```
-- Remove multiple spaces before and one space after.
select concat('[',btrim('    hello '),']');
+--------------------------------------+
| concat('[', btrim('    hello '), ']') |
+--------------------------------------+
| [hello]                              |
+--------------------------------------+
-- Remove any instances of x or y or z at beginning or end. Leave
 spaces alone.
select concat('[',btrim('xy    hello zyzzxx','xyz'),']');
+-----------------------------------------------------+
| concat('[', btrim('xy    hello zyzzxx', 'xyz'), ']') |
+-----------------------------------------------------+
| [    hello ]                                        |
+-----------------------------------------------------+
-- Remove any instances of x or y or z at beginning or end.
-- Leave x, y, z alone in the middle of the string.
select concat('[',btrim('xyhelxyzlozyzzxx','xyz'),']');
+-----------------------------------------------------+
| concat('[', btrim('xyhelxyzlozyzzxx', 'xyz'), ']') |
+-----------------------------------------------------+
| [helxyzlo]                                          |
+-----------------------------------------------------+
```

CHAR_LENGTH(STRING a), CHARACTER_LENGTH(STRING a)

Purpose: Returns the length in characters of the argument string. Aliases for the length() function.

Return type: INT

CHR(INT character_code)

Purpose: Returns a character specified by a decimal code point value. The interpretation and display of the resulting character depends on your system locale. Because consistent processing of Impala string values is only guaranteed for values within the ASCII range, only use this function for values corresponding to ASCII characters. In particular, parameter values greater than 255 return an empty string.

Return type: STRING

Usage notes: Can be used as the inverse of the ascii() function, which converts a character to its numeric ASCII code.

Examples:

```
SELECT chr(65);
+---------+
| chr(65) |
+---------+
| A       |
+---------+
SELECT chr(97);
+---------+
| chr(97) |
+---------+
| a       |
```

```
+---------+
```

CONCAT(STRING a, STRING b...)

Purpose: Returns a single string representing all the argument values joined together.

If any argument is NULL, the function returns NULL.

Return type: STRING

Usage notes: concat() and concat_ws() are appropriate for concatenating the values of multiple columns within the same row, while group_concat() joins together values from different rows.

CONCAT_WS(STRING sep, STRING a, STRING b...)

Purpose: Returns a single string representing the second and following argument values joined together, delimited by a specified separator.

If any argument is NULL, the function returns NULL.

Return type: STRING

Usage notes: concat() and concat_ws() are appropriate for concatenating the values of multiple columns within the same row, while group_concat() joins together values from different rows.

FIND_IN_SET(STRING str, STRING strList)

Purpose: Returns the position (starting from 1) of the first occurrence of a specified string within a comma-separated string. Returns NULL if either argument is NULL, 0 if the search string is not found, or 0 if the search string contains a comma.

Return type: INT

GROUP_CONCAT(STRING s [, STRING sep])

Purpose: 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.

Return type: STRING

Usage notes: concat() and concat_ws() are appropriate for concatenating the values of multiple columns within the same row, while group_concat() joins together values from different rows.

By default, returns a single string covering the whole result set. To include other columns or values in the result set, or to produce multiple concatenated strings for subsets of rows, include a GROUP BY clause in the query.

Strictly speaking, group_concat() is an aggregate function, not a scalar function like the others in this list. For additional details and examples, see the GROUP_CONCAT function.

INITCAP(STRING str)

Purpose: Returns the input string with the first letter of each word capitalized and all other letters in lowercase.

Return type: STRING

Example:

INITCAP("i gOt mY ChiCkeNs in tHe yard.") returns "I Got My Chickens In The Yard.".

INSTR(STRING str, STRING substr [, BIGINT position [, BIGINT occurrence]])

Purpose: Returns the position (starting from 1) of the first occurrence of a substring within a longer string.

Return type: INT

Usage notes:

If the *substr* is not present in *str*, the function returns 0.

The optional third and fourth arguments let you find instances of the *substr* other than the first instance starting from the left.

- The third argument, *position*, lets you specify a starting point within the *str* other than 1.

```
-- Restricting the search to positions 7..end,
-- the first occurrence of 'b' is at position 9.
select instr('foo bar bletch', 'b', 7);
+---------------------------------+
| instr('foo bar bletch', 'b', 7) |
+---------------------------------+
| 9                               |
+---------------------------------+
```

- If there are no more occurrences after the specified position, the result is 0.
- If *position* is negative, the search works right-to-left starting that many characters from the right. The return value still represents the position starting from the left side of *str*.

```
-- Scanning right to left, the first occurrence of 'o'
-- is at position 8. (8th character from the left.)
select instr('hello world','o',-1);
+-------------------------------+
| instr('hello world', 'o', -1) |
+-------------------------------+
| 8                             |
+-------------------------------+
```

- The fourth argument, *occurrence*, lets you specify an occurrence other than the first.

```
-- 2nd occurrence of 'b' is at position 9.
select instr('foo bar bletch', 'b', 1, 2);
+------------------------------------+
| instr('foo bar bletch', 'b', 1, 2) |
+------------------------------------+
| 9                                  |
+------------------------------------+
```

- If *occurrence* is greater than the number of matching occurrences, the function returns 0.
- *occurrence* cannot be negative or zero. A non-positive value for this argument causes an error.
- If either of the optional arguments, *position* or *occurrence*, is NULL, the function also returns NULL.

JARO_DISTANCE(STRING str1, STRING str2)
JARO_DST(STRING str1, STRING str2)

Purpose: Returns the Jaro distance between two input strings. The Jaro distance is a measure of similarity between two strings and is the complementary of JARO_SIMILARITY(), i.e. (1 - JARO _SIMILARITY()).

Return type: DOUBLE

Usage notes:

If the two input strings are identical, the function returns 0.0.

If there is no matching character between the input strings, the function returns 1.0.

If either input strings is NULL, the function returns NULL.

If the length of either input string is bigger than 255 characters, the function returns an error.

JARO_SIMILARITY(STRING str1, STRING str2)
JARO_SIM(STRING str1, STRING str2)

Purpose: Returns the Jaro similarity of two strings. The higher the Jaro similarity for two strings is, the more similar the strings are.

Return type: DOUBLE

Usage notes:

If the two input strings are identical, the function returns 1.0.

If there is no matching character between the input strings, the function returns 0.0.

If either input strings is NULL, the function returns NULL.

If the length of either input string is bigger than 255 characters, the function returns an error.

JARO_WINKLER_DISTANCE(STRING str1, STRING str2[, DOUBLE scaling_factor, DOUBLE boost_threshold])
JW_DST(STRING str1, STRING str2[, DOUBLE scaling_factor, DOUBLE boost_threshold])

Purpose: Returns the Jaro-Winkler distance of two input strings. It is the complementary of JARO _WINKLER_SIMILARITY(), i.e. 1 - JARO_WINKLER_SIMILARITY().

Return type: DOUBLE

Usage notes:

If the two input strings are identical, the function returns 0.0.

If there is no matching character between the input strings, the function returns 1.0.

The function returns an error in the following cases:

- The length of either input string is bigger than 255 characters.
- *scaling_factor* < 0.0 or *scaling_factor* > 0.25
- *boost_threshold* < 0.0 or *boost_threshold* > 1.0

If either input strings is NULL, the function returns NULL.

The default *scaling factor* is 0.1.

The prefix weight will only be applied if the Jaro-distance exceeds the optional *boost_threshold*. By default, the *boost_threshold* value is 0.7.

JARO_WINKLER_SIMILARITY(STRING str1, STRING str2[, DOUBLE scaling_factor, DOUBLE boost_threshold])
JARO_SIM(STRING str1, STRING str2[, DOUBLE scaling_factor, DOUBLE boost_threshold])

Purpose: Returns the Jaro-Winkler Similarity between two input strings. The Jaro-Winkler similarity uses a prefix weight, specified by *scaling factor*, which gives more favorable ratings to strings that match from the beginning for a set prefix length, up to a maximum of four characters.

Use Jaro or Jaro-Winkler functions to perform fuzzy matches on relatively short strings, e.g. to scrub user inputs of names against the records in the database.

Return type: DOUBLE

Usage notes:

If the two input strings are identical, the function returns 1.0.

If there is no matching character between the input strings, the function returns 0.0.

The function returns an error in the following cases:

- The length of either input string is bigger than 255 characters.
- *scaling_factor* < 0.0 or *scaling_factor* > 0.25
- *boost_threshold* < 0.0 or *boost_threshold* > 1.0

If either input strings is NULL, the function returns NULL.

The default *scaling factor* is 0.1.

The prefix weight will only be applied if the Jaro-similarity exceeds the optional *boost_threshold*. By default, the *boost_threshold* value is 0.7.

LEFT(STRING a, INT num_chars)

See the STRLEFT() function.

LENGTH(STRING a)

Purpose: Returns the length in characters of the argument string.

Return type: INT

LEVENSHTEIN(STRING str1, STRING str2), LE_DST(STRING str1, STRING str2)

Purpose: Returns the Levenshtein distance between two input strings. The Levenshtein distance between two strings is the minimum number of single-character edits required to transform one string to other. The function indicates how different the input strings are.

Return type: INT

Usage notes:

If input strings are equal, the function returns 0.

If either input exceeds 255 characters, the function returns an error.

If either input string is NULL, the function returns NULL.

If the length of one input string is zero, the function returns the length of the other string.

Example:

LEVENSHTEIN ('welcome', 'We come') returns 2, first change to replace 'w' to 'W', and then to replace 'l' to a space character.

LOCATE(STRING substr, STRING str[, INT pos])

Purpose: Returns the position (starting from 1) of the first occurrence of a substring within a longer string, optionally after a particular position.

Return type: INT

LOWER(STRING a), LCASE(STRING a)

Purpose: Returns the argument string converted to all-lowercase.

Return type: STRING

Usage notes:

In Impala 2.5 and higher, you can simplify queries that use many UPPER() and LOWER() calls to do case-insensitive comparisons, by using the ILIKE or IREGEXP operators instead.

LPAD(STRING str, INT len, STRING pad)

Purpose: Returns a string of a specified length, based on the first argument string. If the specified string is too short, it is padded on the left with a repeating sequence of the characters from the pad string. If the specified string is too long, it is truncated on the right.

Return type: STRING

LTRIM(STRING a [, STRING chars_to_trim])

Purpose: Returns the argument string with all occurrences of characters specified by the second argument removed from the left side. Removes spaces if the second argument is not specified.

Return type: STRING

PARSE_URL(STRING urlString, STRING partToExtract [, STRING keyToExtract])

Purpose: Returns the portion of a URL corresponding to a specified part. The part argument can be 'PROTOCOL', 'HOST', 'PATH', 'REF', 'AUTHORITY', 'FILE', 'USERINFO', or 'QUERY'. Uppercase is required for these literal values. When requesting the QUERY portion of the URL, you can optionally specify a key to retrieve just the associated value from the key-value pairs in the query string.

Return type: STRING

Usage notes: This function is important for the traditional Hadoop use case of interpreting web logs. For example, if the web traffic data features raw URLs not divided into separate table columns, you can count visitors to a particular page by extracting the 'PATH' or 'FILE' field, or analyze search terms by extracting the corresponding key from the 'QUERY' field.

REGEXP_ESCAPE(STRING source)

Purpose: The REGEXP_ESCAPE function returns a string escaped for the special character in RE2 library so that the special characters are interpreted literally rather than as special characters. The following special characters are escaped by the function:

```
.\+*?[^]$(){}=!<>|:-
```

Return type: string

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the *RE2 documentation*. It has most idioms familiar from regular expressions in Perl, Python, and so on, including .*? for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the \ character for escaping, use \\ to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as [[:digit:]] instead of \d which you would have to escape as \\d.

Examples:

This example shows escaping one of special characters in RE2.

```
+-------------------------------------------------------+
| regexp_escape('Hello.world')                          |
+-------------------------------------------------------+
| Hello\.world                                          |
+-------------------------------------------------------+
```

This example shows escaping all the special characters in RE2.

```
+-------------------------------------------------------------+
| regexp_escape('a.b\\c+d*e?f[g]h$i(j)k{l}m=n!o<p>q|r:s-t')  |
+-------------------------------------------------------------+
| a\.b\\c\+d\*e\?f\[g\]h\$i\(j\)k\{l\}m\=n\!o\<p\>q\|r\:s\-t |
+-------------------------------------------------------------+
```

REGEXP_EXTRACT(STRING subject, STRING pattern, INT index)

Purpose: Returns the specified () group from a string based on a regular expression pattern. Group 0 refers to the entire extracted string, while group 1, 2, and so on refers to the first, second, and so on (...) portion.

Return type: STRING

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the *RE2 documentation*. It has most idioms familiar from regular expressions in Perl, Python, and so on, including .*? for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the \ character for escaping, use \\ to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as [[:digit:]] instead of \d which you would have to escape as \\d.

Examples:

This example shows how group 0 matches the full pattern string, including the portion outside any () group:

```
[localhost:21000] > select regexp_extract('abcdef123ghi456jkl','
.*?(\\d+)',0);
+-----------------------------------------------------+
| regexp_extract('abcdef123ghi456jkl', '.*?(\\d+)', 0) |
+-----------------------------------------------------+
| abcdef123ghi456                                     |
+-----------------------------------------------------+
Returned 1 row(s) in 0.11s
```

This example shows how group 1 matches just the contents inside the first () group in the pattern string:

```
[localhost:21000] > select regexp_extract('abcdef123ghi456jkl','
.*?(\\d+)',1);
+-----------------------------------------------------+
| regexp_extract('abcdef123ghi456jkl', '.*?(\\d+)', 1) |
+-----------------------------------------------------+
| 456                                                 |
+-----------------------------------------------------+
Returned 1 row(s) in 0.11s
```

Unlike in earlier Impala releases, the regular expression library used in Impala 2.0 and later supports the .*? idiom for non-greedy matches. This example shows how a pattern string starting with .*? matches the shortest possible portion of the source string, returning the rightmost set of lowercase letters. A pattern string both starting and ending with .*? finds two potential matches of equal length, and returns the first one found (the leftmost set of lowercase letters).

```
[localhost:21000] > select regexp_extract('AbcdBCdefGHI','.*?([[
:lower:]]+)',1);
+-----------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*?([[:lower:]]+)', 1) |
+-----------------------------------------------------+
| def                                                 |
+-----------------------------------------------------+
[localhost:21000] > select regexp_extract('AbcdBCdefGHI','.*?([[
:lower:]]+).*?',1);
+-----------------------------------------------------------+
| regexp_extract('abcdbcdefghi', '.*?([[:lower:]]+).*?', 1) |
+-----------------------------------------------------------+
| bcd                                                       |
+-----------------------------------------------------------+
```

REGEXP_LIKE(STRING source, STRING pattern[, STRING options])

Purpose: Returns true or false to indicate whether the source string contains anywhere inside it the regular expression given by the pattern. The optional third argument consists of letter flags that change how the match is performed, such as i for case-insensitive matching.

Syntax:

The flags that you can include in the optional third argument are:

- c: Case-sensitive matching (the default).
- i: Case-insensitive matching. If multiple instances of c and i are included in the third argument, the last such option takes precedence.
- m: Multi-line matching. The ^ and $ operators match the start or end of any line within the source string, not the start and end of the entire string.
- n: Newline matching. The . operator can match the newline character. A repetition operator such as .* can match a portion of the source string that spans multiple lines.

Return type: BOOLEAN

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the *RE2 documentation*. It has most idioms familiar from regular expressions in Perl, Python, and so on, including .*? for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the \ character for escaping, use \\ to represent the regular expression escape character in any regular expressions that you submit through `impala-shell`. You might prefer to use the equivalent character class names, such as [[:digit:]] instead of \d which you would have to escape as \\d.

Examples:

This example shows how REGEXP_LIKE() can test for the existence of various kinds of regular expression patterns within a source string:

```
-- Matches because the 'f' appears somewhere in 'foo'.
select regexp_like('foo','f');
+-------------------------+
| regexp_like('foo', 'f') |
+-------------------------+
| true                    |
+-------------------------+

-- Does not match because the comparison is case-sensitive by de
fault.
select regexp_like('foo','F');
+-------------------------+
| regexp_like('foo', 'f') |
+-------------------------+
| false                   |
+-------------------------+

-- The 3rd argument can change the matching logic, such as 'i' me
aning case-insensitive.
select regexp_like('foo','F','i');
+----------------------------+
| regexp_like('foo', 'f', 'i') |
+----------------------------+
| true                       |
```

```
+-----------------------------+
-- The familiar regular expression notations work, such as ^ and
 $ anchors...
select regexp_like('foo','f$');
+-------------------------+
| regexp_like('foo', 'f$') |
+-------------------------+
| false                   |
+-------------------------+
select regexp_like('foo','o$');
+-------------------------+
| regexp_like('foo', 'o$') |
+-------------------------+
| true                    |
+-------------------------+

-- ...and repetition operators such as * and +
select regexp_like('foooooobar','fo+b');
+---------------------------------+
| regexp_like('foooooobar', 'fo+b') |
+---------------------------------+
| true                            |
+---------------------------------+

select regexp_like('foooooobar','fx*y*o*b');
+--------------------------------------+
| regexp_like('foooooobar', 'fx*y*o*b') |
+--------------------------------------+
| true                                 |
+--------------------------------------+
```

REGEXP_REPLACE(STRING initial, STRING pattern, STRING replacement)

Purpose: Returns the initial argument with the regular expression pattern replaced by the final argument string.

Return type: STRING

In Impala 2.0 and later, the Impala regular expression syntax conforms to the POSIX Extended Regular Expression syntax used by the Google RE2 library. For details, see the *RE2 documentation*. It has most idioms familiar from regular expressions in Perl, Python, and so on, including .*? for non-greedy matches.

In Impala 2.0 and later, a change in the underlying regular expression library could cause changes in the way regular expressions are interpreted by this function. Test any queries that use regular expressions and adjust the expression patterns if necessary.

Because the `impala-shell` interpreter uses the \ character for escaping, use \\ to represent the regular expression escape character in any regular expressions that you submit through `impala-shell` . You might prefer to use the equivalent character class names, such as [[:digit:]] instead of \d which you would have to escape as \\d.

Examples:

These examples show how you can replace parts of a string matching a pattern with replacement text, which can include backreferences to any () groups in the pattern string. The backreference numbers start at 1, and any \ characters must be escaped as \\.

Replace a character pattern with new text:

```
[localhost:21000] > select regexp_replace('aaabbbaaa','b+','xyz'
);
+----------------------------------------+
```

```
| regexp_replace('aaabbbaaa', 'b+', 'xyz') |
+------------------------------------------+
| aaaxyzaaa                                |
+------------------------------------------+
Returned 1 row(s) in 0.11s
```

Replace a character pattern with substitution text that includes the original matching text:

```
[localhost:21000] > select regexp_replace('aaabbbaaa','(b+)','<\
\1>');
+-----------------------------------------------+
| regexp_replace('aaabbbaaa', '(b+)', '<\\1>') |
+-----------------------------------------------+
| aaa<bbb>aaa                                   |
+-----------------------------------------------+
Returned 1 row(s) in 0.11s
```

Remove all characters that are not digits:

```
[localhost:21000] > select regexp_replace('123-456-789','[^[:dig
it:]]','');
+-------------------------------------------------+
| regexp_replace('123-456-789', '[^[:digit:]]', '') |
+-------------------------------------------------+
| 123456789                                       |
+-------------------------------------------------+
Returned 1 row(s) in 0.12s
```

REPEAT(STRING str, INT n)

Purpose: Returns the argument string repeated a specified number of times.

Return type: STRING

REPLACE(STRING initial, STRING target, STRING replacement)

Purpose: Returns the initial argument with all occurrences of the target string replaced by the replacement string.

Return type: STRING

Usage notes:

Because this function does not use any regular expression patterns, it is typically faster than REGE XP_REPLACE() for simple string substitutions.

If any argument is NULL, the return value is NULL.

Matching is case-sensitive.

If the replacement string contains another instance of the target string, the expansion is only performed once, instead of applying again to the newly constructed string.

Examples:

```
-- Replace one string with another.
select replace('hello world','world','earth');
+-----------------------------------------+
| replace('hello world', 'world', 'earth') |
+-----------------------------------------+
| hello earth                             |
+-----------------------------------------+
-- All occurrences of the target string are replaced.
select replace('hello world','o','0');
+---------------------------------+
| replace('hello world', 'o', '0') |
```

```
+----------------------------------+
| hell0 w0rld                      |
+----------------------------------+

-- If no match is found, the original string is returned unchang
ed.
select replace('hello world','xyz','abc');
+--------------------------------------+
| replace('hello world', 'xyz', 'abc') |
+--------------------------------------+
| hello world                          |
+--------------------------------------+
```

REVERSE(STRING a)

Purpose: Returns the argument string with characters in reversed order.

Return type: STRING

RIGHT(STRING a, INT num_chars)

See the STRRIGHT function.

RPAD(STRING str, INT len, STRING pad)

Purpose: Returns a string of a specified length, based on the first argument string. If the specified string is too short, it is padded on the right with a repeating sequence of the characters from the pad string. If the specified string is too long, it is truncated on the right.

Return type: STRING

RTRIM(STRING a [, STRING chars_to_trim])

Purpose: Returns the argument string with all occurrences of characters specified by the second argument removed from the right side. Removes spaces if the second argument is not specified.

Return type: STRING

SPACE(INT n)

Purpose: Returns a concatenated string of the specified number of spaces. Shorthand for repeat(' ',*n*).

Return type: STRING

SPLIT_PART(STRING source, STRING delimiter, BIGINT index)

Purpose: Returns the requested indexth part of the input *source* string split by the *delimiter*.

- If *index* is a positive number, returns the *index*th part from the left within the *source* string.
- If *index* is a negative number, returns the *index*th part from the right within the *source* string.
- If *index* is 0, returns an error.

The *delimiter* can consist of multiple characters, not just a single character.

All matching of the delimiter is done exactly, not using any regular expression patterns.

Return type: STRING

Examples:

SPLIT_PART('x,y,z',',',2) returns 'y'.

SPLIT_PART('one***two***three','***',2) returns 'two'.

SPLIT_PART('abc@@def@@ghi', '@@', 3) returns 'ghi'.

SPLIT_PART('abc@@def@@ghi', '@@', -3) returns 'abc'.

STRLEFT(STRING a, INT num_chars)

Purpose: Returns the leftmost characters of the string. Shorthand for a call to substr() with 2 arguments.

Return type: STRING

STRRIGHT(STRING a, INT num_chars)

Purpose: Returns the rightmost characters of the string. Shorthand for a call to substr() with 2 arguments.

Return type: STRING

SUBSTR(STRING a, INT start [, INT len]), SUBSTRING(STRING a, INT start [, INT len])

Purpose: Returns the portion of the string starting at a specified point, optionally with a specified maximum length. The characters in the string are indexed starting at 1.

Return type: STRING

TRANSLATE(STRING input, STRING from, STRING to)

Purpose: Returns the input string with each character in the from argument replaced with the corresponding character in the to argument. The characters are matched in the order they appear in from and to.

For example: translate ('hello world','world','earth') returns 'hetta earth'.

Return type: STRING

Usage notes:

If from contains more characters than to, the from characters that are beyond the length of to are removed in the result.

For example:

translate('abcdedg', 'bcd', '1') returns 'a1eg'.

translate('Unit Number#2', '# ', '_') returns 'UnitNumber_2'.

If from is NULL, the function returns NULL.

If to contains more characters than from, the extra characters in to are ignored.

If from contains duplicate characters, the duplicate character is replaced with the first matching character in to.

For example: translate ('hello','ll','67') returns 'he66o'.

TRIM(STRING a)

Purpose: Returns the input string with both leading and trailing spaces removed. The same as passing the string through both LTRIM() and RTRIM().

Usage notes: Often used during data cleansing operations during the ETL cycle, if input values might still have surrounding spaces. For a more general-purpose function that can remove other leading and trailing characters besides spaces, see BTRIM().

Return type: STRING

UPPER(STRING a), UCASE(STRING a)

Purpose: Returns the argument string converted to all-uppercase.

Return type: STRING

Usage notes:

In Impala 2.5 and higher, you can simplify queries that use many UPPER() and LOWER() calls to do case-insensitive comparisons, by using the ILIKE or IREGEXP operators instead.

Related Information

General information on Base64 encoding

GROUP_CONCAT function

Impala miscellaneous functions

Impala supports some utility functions that do not operate on a particular column or data type.

For example:

- CURRENT_DATABASE
- EFFECTIVE_USER
- GET_JSON_OBJECT
- LOGGED_IN_USER
- PID
- SLEEP
- USER
- UUID
- VERSION

CURREN_DATABASE()

Purpose: Returns the database that the session is currently using, either default if no database has been selected, or whatever database the session switched to through a USE statement or the `impalad` -d option.

Return type: STRING

EFFECTIVE_USER()

Purpose: Typically returns the same value as USER(), except if delegation is enabled, in which case it returns the ID of the delegated user.

Return type: STRING

GET_JSON_OBJECT(STRING json_str, STRING selector)

Purpose: Extracts JSON object from the *json_str* based on the *selector* JSON path and returns the string of the extracted JSON object.

The function returns NULL if the input *json_str* is invalid or if nothing is selected based on the *selector* JSON path.

The following characters are supported in the *selector* JSON path:

- $: Denotes the root object
- . : Denotes the child operator
- [] : Denotes the subscript operator for array
- * : Denotes the wildcard for [] or .

Return type: STRING

Examples:

```
---- QUERY
SELECT GET_JSON_OBJECT ('{"a":true, "b":false, "c":true}', '$.*')
;
---- RESULTS
[true,false,true]
```

```
---- QUERY
SELECT GET_JSON_OBJECT(t.json, '$.a.b.c') FROM (VALUES (
 ('{"a": {"b": {"c": 1}}}' AS json),
 ('{"a": {"b": {"c": 2}}}'),
 ('{"a": {"b": {"c": 3}}}')
)) t
---- RESULTS
'1'
'2'
```

```
'3'
```

```
---- QUERY
SELECT GET_JSON_OBJECT(t.json, '$.a'),
 GET_JSON_OBJECT(t.json, '$.b'),
 GET_JSON_OBJECT(t.json, '$.c')
FROM (VALUES (
 ('{"a":1, "b":2, "c":3}' AS json),
 ('{"b":2, "c":3}'),
 ('{"c":3}')
)) t
---- RESULTS
'1','2','3'
'NULL','2','3'
'NULL','NULL','3'
```

```
---- QUERY
SELECT GET_JSON_OBJECT(t.json, '$[1]'),
 GET_JSON_OBJECT(t.json, '$[*]')
FROM (VALUES (
 ('["a", "b", "c"]' AS json),
 ('["a", "b"]'),
 ('["a"]')
)) t
---- RESULTS
'b','["a","b","c"]'
'b','["a","b"]'
'NULL','a'
```

Added in: Impala 3.1

LOGGED_IN_USER()

Purpose: Typically returns the same value as USER(). If delegation is enabled, it returns the ID of the delegated user.

LOGGED_IN_USER() is an alias of EFFECTIVE_USER().

Return type: STRING

Added in: Impala 3.1

PID()

Purpose: Returns the process ID of the `impalad` daemon that the session is connected to. You can use it during low-level debugging, to issue Linux commands that trace, show the arguments, and so on the `impalad` process.

Return type: INT

SLEEP(INT ms)

Purpose: Pauses the query for a specified number of milliseconds. For slowing down queries with small result sets enough to monitor runtime execution, memory usage, or other factors that otherwise would be difficult to capture during the brief interval of query execution. When used in the SELECT list, it is called once for each row in the result set; adjust the number of milliseconds accordingly. For example, a query SELECT *, sleep(5) FROM table_with_1000_rows would take at least 5 seconds to complete (5 milliseconds * 1000 rows in result set). To avoid an excessive number of concurrent queries, use this function for troubleshooting on test and development systems, not for production queries.

Return type: N/A

USER()

Purpose: Returns the username of the Linux user who is connected to the `impalad` daemon. Typically called a single time, in a query without any FROM clause, to understand how authorization settings apply in a security context; once you know the logged-in username, you can check which groups that user belongs to, and from the list of groups you can check which roles are available to those groups through the authorization policy file.

Impala 2.0 and later, USER() returns the full Kerberos principal string, such as user@example.com, in a Kerberized environment.

When delegation is enabled, consider calling the effective_user() function instead.

Return type: STRING

UUID()

Purpose: Returns a universal unique identifier, a 128-bit value encoded as a string with groups of hexadecimal digits separated by dashes.

Each call to UUID() produces a new arbitrary value.

If you get a UUID for each row of a result set, you can use it as a unique identifier within a table, or even a unique ID across tables.

Return type: STRING

Usage notes:

Ascending numeric sequences of type BIGINT are often used as identifiers within a table, and as join keys across multiple tables. The uuid() value is a convenient alternative that does not require storing or querying the highest sequence number. For example, you can use it to quickly construct new unique identifiers during a data import job, or to combine data from different tables without the likelihood of ID collisions.

VERSION()

Purpose: Returns information such as the precise version number and build date for the impalad daemon that you are currently connected to. Typically used to confirm that you are connected to the expected level of Impala to use a particular feature, or to connect to several nodes and confirm they are all running the same level of `impalad`.

Return type: STRING (with one or more embedded newlines)

COORDINATOR()

Purpose: Returns the name of the host which is running the impalad daemon that is acting as the coordinator for the current query.

Return type: STRING

Added in: Impala 3.1

Impala aggregate functions

Aggregate functions are a special category with different rules. These functions calculate a return value across all the items in a result set, so they require a FROM clause in the query.

```
select count(product_id) from product_catalog;
select max(height), avg(height) from census_data where age > 20;
```

Aggregate functions also ignore NULL values rather than returning a NULL result. For example, if some rows have NULL for a particular column, those rows are ignored when computing the AVG() for that column. Likewise, specifying COUNT(*col_name*) in a query counts only those rows where *col_name* contains a non-NULL value.

APPX_MEDIAN function

An aggregate function that returns a value that is approximately the median (midpoint) of values in the set of input values.

Syntax:

```
APPX_MEDIAN([DISTINCT | ALL] expression)
```

This function works with any input type, because the only requirement is that the type supports less-than and greater-than comparison operators.

Usage notes:

Because the return value represents the estimated midpoint, it might not reflect the precise midpoint value, especially if the cardinality of the input values is very high. If the cardinality is low (up to approximately 20,000), the result is more accurate because the sampling considers all or almost all of the different values.

Return type: Same as the input value, except for CHAR and VARCHAR arguments which produce a STRING result

The return value is always the same as one of the input values, not an “in-between” value produced by averaging.

Restrictions:

This function cannot be used in an analytic context. That is, the OVER() clause is not allowed at all with this function.

The APPX_MEDIAN function returns only the first 10 characters for string values (string, varchar, char). Additional characters are truncated.

Examples:

The following example uses a table of a million random floating-point numbers ranging up to approximately 50,000. The average is approximately 25,000. Because of the random distribution, we would expect the median to be close to this same number. Computing the precise median is a more intensive operation than computing the average, because it requires keeping track of every distinct value and how many times each occurs. The APPX_MEDIAN() function uses a sampling algorithm to return an approximate result, which in this case is close to the expected value. To make sure that the value is not substantially out of range due to a skewed distribution, subsequent queries confirm that there are approximately 500,000 values higher than the APPX_MEDIAN() value, and approximately 500,000 values lower than the APPX_MEDIAN() value.

```
[localhost:21000] > select min(x), max(x), avg(x) from million_numbers;
+-------------------+-------------------+-------------------+
| min(x)            | max(x)            | avg(x)            |
+-------------------+-------------------+-------------------+
| 4.725693727250069 | 49994.56852674231 | 24945.38563793553 |
+-------------------+-------------------+-------------------+
[localhost:21000] > select appx_median(x) from million_numbers;
+----------------+
| appx_median(x) |
+----------------+
| 24721.6        |
+----------------+
[localhost:21000] > select count(x) as higher from million_numbers where x >
 (select appx_median(x) from million_numbers);
+--------+
| higher |
+--------+
| 502013 |
+--------+
[localhost:21000] > select count(x) as lower from million_numbers where x <
 (select appx_median(x) from million_numbers);
+--------+
| lower  |
+--------+
| 497987 |
+--------+
```

The following example computes the approximate median using a subset of the values from the table, and then confirms that the result is a reasonable estimate for the midpoint.

```
[localhost:21000] > select appx_median(x) from million_numbers where x betwe
en 1000 and 5000;
+--------------------+
| appx_median(x)     |
+--------------------+
| 3013.107787358159 |
+--------------------+
[localhost:21000] > select count(x) as higher from million_numbers where x b
etween 1000 and 5000 and x > 3013.107787358159;
+--------+
| higher |
+--------+
| 37692  |
+--------+
[localhost:21000] > select count(x) as lower from million_numbers where x
 between 1000 and 5000 and x < 3013.107787358159;
+-------+
| lower |
+-------+
| 37089 |
+-------+
```

AVG function

An aggregate function that returns the average value from a set of numbers or TIMESTAMP values. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to AVG are NULL, AVG returns NULL.

Syntax:

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

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Return type: DOUBLE for numeric values; TIMESTAMP for TIMESTAMP values

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex types on page 49 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM() and AVG() are computed using the numeric R_NATIONKEY field, and the general-purpose MAX() and MIN() values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
```

```
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct
_vals |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5
    |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5
    |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5
    |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5
    |
```

```
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5
    |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
```

Examples:

```
-- Average all the non-NULL values in a column.
insert overwrite avg_t values (2),(4),(6),(null),(null);
-- The average of the above values is 4: (2+4+6) / 3. The 2 NULL values are
 ignored.
select avg(x) from avg_t;
-- Average only certain values from the column.
select avg(x) from t1 where month = 'January' and year = '2013';
-- Apply a calculation to the value of the column before averaging.
select avg(x/3) from t1;
-- Apply a function to the value of the column before averaging.
-- Here we are substituting a value of 0 for all NULLs in the column,
-- so that those rows do factor into the return value.
select avg(isnull(x,0)) from t1;
-- Apply some number-returning function to a string column and average the
results.
-- If column s contains any NULLs, length(s) also returns NULL and those ro
ws are ignored.
select avg(length(s)) from t1;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, avg(page_visits) from web_stats group by month, year;
-- Filter the input to eliminate duplicates before performing the calcula
tion.
select avg(distinct x) from t1;
-- Filter the output after performing the calculation.
select avg(x) from t1 group by y having avg(x) between 1 and 20;
```

The following examples show how to use AVG() in an analytic context. They use a table containing integers from 1 to 10. Notice how the AVG() is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, avg(x) over (partition by property) as avg from int_t wh
ere property in ('odd','even');
+----+----------+-----+
| x  | property | avg |
+----+----------+-----+
| 2  | even     | 6   |
| 4  | even     | 6   |
| 6  | even     | 6   |
| 8  | even     | 6   |
| 10 | even     | 6   |
| 1  | odd      | 5   |
| 3  | odd      | 5   |
| 5  | odd      | 5   |
| 7  | odd      | 5   |
| 9  | odd      | 5   |
+----+----------+-----+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the “window”). The following examples use AVG() in an analytic context (that is, with an OVER() clause) to produce a running average of all the even values, then a running average of all the odd values. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURREN

T ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property,
  avg(x) over (partition by property order by x) as 'cumulative average'
  from int_t where property in ('odd','even');
+----+----------+--------------------+
| x  | property | cumulative average |
+----+----------+--------------------+
| 2  | even     | 2                  |
| 4  | even     | 3                  |
| 6  | even     | 4                  |
| 8  | even     | 5                  |
| 10 | even     | 6                  |
| 1  | odd      | 1                  |
| 3  | odd      | 2                  |
| 5  | odd      | 3                  |
| 7  | odd      | 4                  |
| 9  | odd      | 5                  |
+----+----------+--------------------+

select x, property,
  avg(x) over
  (
    partition by property
    order by x
    range between unbounded preceding and current row
  ) as 'cumulative average'
from int_t where property in ('odd','even');
+----+----------+--------------------+
| x  | property | cumulative average |
+----+----------+--------------------+
| 2  | even     | 2                  |
| 4  | even     | 3                  |
| 6  | even     | 4                  |
| 8  | even     | 5                  |
| 10 | even     | 6                  |
| 1  | odd      | 1                  |
| 3  | odd      | 2                  |
| 5  | odd      | 3                  |
| 7  | odd      | 4                  |
| 9  | odd      | 5                  |
+----+----------+--------------------+
select x, property,
  avg(x) over
  (
    partition by property
    order by x
    rows between unbounded preceding and current row
  ) as 'cumulative average'
  from int_t where property in ('odd','even');
+----+----------+--------------------+
| x  | property | cumulative average |
+----+----------+--------------------+
| 2  | even     | 2                  |
| 4  | even     | 3                  |
| 6  | even     | 4                  |
| 8  | even     | 5                  |
| 10 | even     | 6                  |
| 1  | odd      | 1                  |
| 3  | odd      | 2                  |
| 5  | odd      | 3                  |
| 7  | odd      | 4                  |
```

```
| 9  | odd      | 5                  |
+----+----------+--------------------+
```

The following examples show how to construct a moving window, with a running average taking into account 1 row before and 1 row after the current row, within the same partition (all the even values or all the odd values). Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause:

```
select x, property,
  avg(x) over
  (
    partition by property
    order by x
    rows between 1 preceding and 1 following
  ) as 'moving average'
  from int_t where property in ('odd','even');
+----+----------+----------------+
| x  | property | moving average |
+----+----------+----------------+
| 2  | even     | 3              |
| 4  | even     | 4              |
| 6  | even     | 6              |
| 8  | even     | 8              |
| 10 | even     | 9              |
| 1  | odd      | 2              |
| 3  | odd      | 3              |
| 5  | odd      | 5              |
| 7  | odd      | 7              |
| 9  | odd      | 8              |
+----+----------+----------------+

-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  avg(x) over
  (
    partition by property
    order by x
    range between 1 preceding and 1 following
  ) as 'moving average'
from int_t where property in ('odd','even');
ERROR: AnalysisException: RANGE is only supported with both the lower and
upper bounds UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM() and AVG() for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

COUNT function

An aggregate function that returns the number of rows, or the number of non-NULL rows.

Syntax:

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

Depending on the argument, COUNT() considers rows that meet certain conditions:

- The notation COUNT(*) includes NULL values in the total.
- The notation COUNT(*column_name*) only considers rows where the column contains a non-NULL value.

- You can also combine COUNT with the DISTINCT operator to eliminate duplicates before counting, and to count the combinations of values across multiple columns.

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Return type: BIGINT

Usage notes:

If you frequently run aggregate functions such as MIN(), MAX(), and COUNT(DISTINCT) on partition key columns, consider enabling the OPTIMIZE_PARTITION_KEY_SCANS query option, which optimizes such queries. See the *OPTIMIZE_PARTITION_KEY_SCANS query option* topic for the kinds of queries that this option applies to, and slight differences in how partitions are evaluated when this query option is enabled.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex types on page 49 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM() and AVG() are computed using the numeric R_NATIONKEY field, and the general-purpose MAX() and MIN() values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
```

```
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct
_vals |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5
    |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5
    |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5
    |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5
    |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5
    |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
```

Examples:

```
-- How many rows total are in the table, regardless of NULL values?
select count(*) from t1;
-- How many rows are in the table with non-NULL values for a column?
select count(c1) from t1;
-- Count the rows that meet certain conditions.
-- Again, * includes NULLs, so COUNT(*) might be greater than COUNT(col).
select count(*) from t1 where x > 10;
select count(c1) from t1 where x > 10;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Combine COUNT and DISTINCT to find the number of unique values.
-- Must use column names rather than * with COUNT(DISTINCT ...) syntax.
-- Rows with NULL values are not counted.
select count(distinct c1) from t1;
-- Rows with a NULL value in _either_ column are not counted.
select count(distinct c1, c2) from t1;
-- Return more than one result.
select month, year, count(distinct visitor_id) from web_stats group by mon
th, year;
```

The following examples show how to use COUNT() in an analytic context. They use a table containing integers from 1 to 10. Notice how the COUNT() is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, count(x) over (partition by property) as count from int_
t where property in ('odd','even');
+----+----------+-------+
| x  | property | count |
+----+----------+-------+
| 2  | even     | 5     |
| 4  | even     | 5     |
| 6  | even     | 5     |
| 8  | even     | 5     |
| 10 | even     | 5     |
| 1  | odd      | 5     |
| 3  | odd      | 5     |
| 5  | odd      | 5     |
| 7  | odd      | 5     |
| 9  | odd      | 5     |
+----+----------+-------+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the “window”). The following examples use COUNT() in an analytic context (that is, with an OVER() clause) to produce a running count of all the even values, then a running count of all the odd values. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property,
  count(x) over (partition by property order by x) as 'cumulative count'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative count |
+----+----------+------------------+
| 2  | even     | 1                |
| 4  | even     | 2                |
| 6  | even     | 3                |
| 8  | even     | 4                |
| 10 | even     | 5                |
| 1  | odd      | 1                |
| 3  | odd      | 2                |
| 5  | odd      | 3                |
| 7  | odd      | 4                |
| 9  | odd      | 5                |
+----+----------+------------------+

select x, property,
  count(x) over
  (
    partition by property
    order by x
    range between unbounded preceding and current row
  ) as 'cumulative total'
from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative count |
+----+----------+------------------+
| 2  | even     | 1                |
| 4  | even     | 2                |
| 6  | even     | 3                |
| 8  | even     | 4                |
| 10 | even     | 5                |
```

```
| 1  | odd      | 1                |
| 3  | odd      | 2                |
| 5  | odd      | 3                |
| 7  | odd      | 4                |
| 9  | odd      | 5                |
+----+----------+------------------+

select x, property,
  count(x) over
  (
    partition by property
    order by x
    rows between unbounded preceding and current row
  ) as 'cumulative total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative count |
+----+----------+------------------+
| 2  | even     | 1                |
| 4  | even     | 2                |
| 6  | even     | 3                |
| 8  | even     | 4                |
| 10 | even     | 5                |
| 1  | odd      | 1                |
| 3  | odd      | 2                |
| 5  | odd      | 3                |
| 7  | odd      | 4                |
| 9  | odd      | 5                |
+----+----------+------------------+
```

The following examples show how to construct a moving window, with a running count taking into account 1 row before and 1 row after the current row, within the same partition (all the even values or all the odd values). Therefore, the count is consistently 3 for rows in the middle of the window, and 2 for rows near the ends of the window, where there is no preceding or no following row in the partition. Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause:

```
select x, property,
  count(x) over
  (
    partition by property
    order by x
    rows between 1 preceding and 1 following
  ) as 'moving total'
  from int_t where property in ('odd','even');
+----+----------+--------------+
| x  | property | moving total |
+----+----------+--------------+
| 2  | even     | 2            |
| 4  | even     | 3            |
| 6  | even     | 3            |
| 8  | even     | 3            |
| 10 | even     | 2            |
| 1  | odd      | 2            |
| 3  | odd      | 3            |
| 5  | odd      | 3            |
| 7  | odd      | 3            |
| 9  | odd      | 2            |
+----+----------+--------------+
-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  count(x) over
  (
    partition by property
```

```
    order by x
    range between 1 preceding and 1 following
  ) as 'moving total'
from int_t where property in ('odd','even');
ERROR: AnalysisException: RANGE is only supported with both the lower and up
per bounds UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

GROUP_CONCAT function

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.

Syntax:

```
GROUP_CONCAT([ALL | DISTINCT] expression [, separator])
```

Usage notes: concat() and concat_ws() are appropriate for concatenating the values of multiple columns within the same row, while group_concat() joins together values from different rows.

By default, returns a single string covering the whole result set. To include other columns or values in the result set, or to produce multiple concatenated strings for subsets of rows, include a GROUP BY clause in the query.

Return type: STRING

This function cannot be used in an analytic context. That is, the OVER() clause is not allowed at all with this function.

Currently, Impala returns an error if the result value grows larger than 1 GiB.

Examples:

The following examples illustrate various aspects of the GROUP_CONCAT() function.

You can call the function directly on a STRING column. To use it with a numeric column, cast the value to STRING.

```
[localhost:21000] > create table t1 (x int, s string);
[localhost:21000] > insert into t1 values (1, "one"), (3, "three"), (2, "
two"), (1, "one");
[localhost:21000] > select group_concat(s) from t1;
+----------------------+
| group_concat(s)      |
+----------------------+
| one, three, two, one |
+----------------------+
[localhost:21000] > select group_concat(cast(x as string)) from t1;
+---------------------------------+
| group_concat(cast(x as string)) |
+---------------------------------+
| 1, 3, 2, 1                      |
+---------------------------------+
```

Specify the DISTINCT keyword to eliminate duplicate values from the concatenated result:

```
[localhost:21000] > select group_concat(distinct s) from t1;
+--------------------------+
| group_concat(distinct s) |
+--------------------------+
| three, two, one          |
+--------------------------+
```

The optional separator lets you format the result in flexible ways. The separator can be an arbitrary string expression, not just a single character.

```
[localhost:21000] > select group_concat(s,"|") from t1;
+----------------------+
| group_concat(s, '|') |
+----------------------+
| one|three|two|one    |
+----------------------+
[localhost:21000] > select group_concat(s,'---') from t1;
+-------------------------+
| group_concat(s, '---')  |
+-------------------------+
| one---three---two---one |
+-------------------------+
```

The default separator is a comma followed by a space. To get a comma-delimited result without extra spaces, specify a delimiter character that is only a comma.

```
[localhost:21000] > select group_concat(s,',') from t1;
+----------------------+
| group_concat(s, ',') |
+----------------------+
| one,three,two,one    |
+----------------------+
```

Including a GROUP BY clause lets you produce a different concatenated result for each group in the result set. In this example, the only X value that occurs more than once is 1, so that is the only row in the result set where GROUP_CONCAT() returns a delimited value. For groups containing a single value, GROUP_CONCAT() returns the original value of its STRING argument.

```
[localhost:21000] > select x, group_concat(s) from t1 group by x;
+---+-----------------+
| x | group_concat(s) |
+---+-----------------+
| 2 | two             |
| 3 | three           |
| 1 | one, one        |
+---+-----------------+
```

MAX function

An aggregate function that returns the maximum value from a set of numbers. Opposite of the MIN function. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to MAX are NULL, MAX returns NULL.

Syntax:

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

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Restrictions: In Impala 2.0 and higher, this function can be used as an analytic function, but with restrictions on any window clause. For MAX() and MIN(), the window clause is only allowed if the start bound is UNBOUNDED PRECEDING.

Return type: Same as the input value, except for CHAR and VARCHAR arguments which produce a STRING result

Usage notes:

If you frequently run aggregate functions such as MIN(), MAX(), and COUNT(DISTINCT) on partition key columns, consider enabling the OPTIMIZE_PARTITION_KEY_SCANS query option, which optimizes such queries. See the *OPTIMIZE_PARTITION_KEY_SCANS query option* topic for the kinds of queries that this option applies to, and slight differences in how partitions are evaluated when this query option is enabled.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex types on page 49 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM() and AVG() are computed using the numeric R_NATIONKEY field, and the general-purpose MAX() and MIN() values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
```

```
+-------------+-----------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+---------------+-----------
----+
| r_name      | count | sum | avg  | minimum   | maximum       | distinct
_vals |
+-------------+-------+-----+------+-----------+---------------+-----------
----+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE    | 5
    |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES | 5
    |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM       | 5
    |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5
    |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA  | 5
    |
+-------------+-------+-----+------+-----------+---------------+-----------
----+
```

Examples:

```
-- Find the largest value for this column in the table.
select max(c1) from t1;
-- Find the largest value for this column from a subset of the table.
select max(c1) from t1 where month = 'January' and year = '2013';
-- Find the largest value from a set of numeric function results.
select max(length(s)) from t1;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, max(purchase_price) from store_stats group by month,
year;
-- Filter the input to eliminate duplicates before performing the calculat
ion.
select max(distinct x) from t1;
```

The following examples show how to use MAX() in an analytic context. They use a table containing integers from 1 to 10. Notice how the MAX() is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, max(x) over (partition by property) as max from int_t wh
ere property in ('odd','even');
+----+----------+-----+
| x  | property | max |
+----+----------+-----+
| 2  | even     | 10  |
| 4  | even     | 10  |
| 6  | even     | 10  |
| 8  | even     | 10  |
```

```
| 10 | even     | 10  |
| 1  | odd      | 9   |
| 3  | odd      | 9   |
| 5  | odd      | 9   |
| 7  | odd      | 9   |
| 9  | odd      | 9   |
+----+----------+-----+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the “window”). The following examples use MAX() in an analytic context (that is, with an OVER() clause) to display the smallest value of X encountered up to each row in the result set. The examples use two columns in the ORDER BY clause to produce a sequence of values that rises and falls, to illustrate how the MAX() result only increases or stays the same throughout each partition within the result set. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property,
  max(x) over (order by property, x desc) as 'maximum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | maximum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 7                     |
| 3 | prime    | 7                     |
| 2 | prime    | 7                     |
| 9 | square   | 9                     |
| 4 | square   | 9                     |
| 1 | square   | 9                     |
+---+----------+-----------------------+
select x, property,
  max(x) over
  (
    order by property, x desc
    rows between unbounded preceding and current row
  ) as 'maximum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | maximum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 7                     |
| 3 | prime    | 7                     |
| 2 | prime    | 7                     |
| 9 | square   | 9                     |
| 4 | square   | 9                     |
| 1 | square   | 9                     |
+---+----------+-----------------------+
select x, property,
  max(x) over
  (
    order by property, x desc
    range between unbounded preceding and current row
  ) as 'maximum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | maximum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 7                     |
| 3 | prime    | 7                     |
```

```
| 2 | prime    | 7                    |
| 9 | square   | 9                    |
| 4 | square   | 9                    |
| 1 | square   | 9                    |
+---+----------+----------------------+
```

The following examples show how to construct a moving window, with a running maximum taking into account all rows before and 1 row after the current row. Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause. Because of an extra Impala restriction on the MAX() and MIN() functions in an analytic context, the lower bound must be UNBO UNDED PRECEDING.

```
select x, property,
  max(x) over
  (
    order by property, x
    rows between unbounded preceding and 1 following
  ) as 'local maximum'
from int_t where property in ('prime','square');
+---+----------+---------------+
| x | property | local maximum |
+---+----------+---------------+
| 2 | prime    | 3             |
| 3 | prime    | 5             |
| 5 | prime    | 7             |
| 7 | prime    | 7             |
| 1 | square   | 7             |
| 4 | square   | 9             |
| 9 | square   | 9             |
+---+----------+---------------+
-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  max(x) over
  (
    order by property, x
    range between unbounded preceding and 1 following
  ) as 'local maximum'
from int_t where property in ('prime','square');
ERROR: AnalysisException: RANGE is only supported with both the lower and u
pper bounds UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

MIN function

An aggregate function that returns the minimum value from a set of numbers. Opposite of the MAX function. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to MIN are NULL, MIN returns NULL.

Syntax:

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

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Restrictions: In Impala 2.0 and higher, this function can be used as an analytic function, but with restrictions on any window clause. For MAX() and MIN(), the window clause is only allowed if the start bound is UNBOUNDED PR ECEDING.

Return type: Same as the input value, except for CHAR and VARCHAR arguments which produce a STRING result

Usage notes:

If you frequently run aggregate functions such as MIN(), MAX(), and COUNT(DISTINCT) on partition key columns, consider enabling the OPTIMIZE_PARTITION_KEY_SCANS query option, which optimizes such queries. See the *OPTIMIZE_PARTITION_KEY_SCANS query option* topic for the kinds of queries that this option applies to, and slight differences in how partitions are evaluated when this query option is enabled.

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex types on page 49 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM() and AVG() are computed using the numeric R_NATIONKEY field, and the general-purpose MAX() and MIN() values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
```

```
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+---------------+-----------
----+
| r_name      | count | sum | avg  | minimum   | maximum       | distinct
_vals |
+-------------+-------+-----+------+-----------+---------------+-----------
----+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE    | 5
    |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES | 5
    |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM       | 5
    |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5
    |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA  | 5
    |
+-------------+-------+-----+------+-----------+---------------+-----------
----+
```

Examples:

```
-- Find the smallest value for this column in the table.
select min(c1) from t1;
-- Find the smallest value for this column from a subset of the table.
select min(c1) from t1 where month = 'January' and year = '2013';
-- Find the smallest value from a set of numeric function results.
select min(length(s)) from t1;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, min(purchase_price) from store_stats group by month, y
ear;
-- Filter the input to eliminate duplicates before performing the calculati
on.
select min(distinct x) from t1;
```

The following examples show how to use MIN() in an analytic context. They use a table containing integers from 1 to 10. Notice how the MIN() is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, min(x) over (partition by property) as min from int_t wh
ere property in ('odd','even');
+----+----------+-----+
| x  | property | min |
+----+----------+-----+
| 2  | even     | 2   |
| 4  | even     | 2   |
| 6  | even     | 2   |
| 8  | even     | 2   |
```

```
| 10 | even     | 2   |
| 1  | odd      | 1   |
| 3  | odd      | 1   |
| 5  | odd      | 1   |
| 7  | odd      | 1   |
| 9  | odd      | 1   |
+----+----------+-----+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the “window”). The following examples use MIN() in an analytic context (that is, with an OVER() clause) to display the smallest value of X encountered up to each row in the result set. The examples use two columns in the ORDER BY clause to produce a sequence of values that rises and falls, to illustrate how the MIN() result only decreases or stays the same throughout each partition within the result set. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property, min(x) over (order by property, x desc) as 'minimum to
this point'
  from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | minimum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 5                     |
| 3 | prime    | 3                     |
| 2 | prime    | 2                     |
| 9 | square   | 2                     |
| 4 | square   | 2                     |
| 1 | square   | 1                     |
+---+----------+-----------------------+

select x, property,
  min(x) over
  (
    order by property, x desc
    range between unbounded preceding and current row
  ) as 'minimum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | minimum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 5                     |
| 3 | prime    | 3                     |
| 2 | prime    | 2                     |
| 9 | square   | 2                     |
| 4 | square   | 2                     |
| 1 | square   | 1                     |
+---+----------+-----------------------+
select x, property,
  min(x) over
  (
    order by property, x desc
    rows between unbounded preceding and current row
  ) as 'minimum to this point'
from int_t where property in ('prime','square');
+---+----------+-----------------------+
| x | property | minimum to this point |
+---+----------+-----------------------+
| 7 | prime    | 7                     |
| 5 | prime    | 5                     |
```

```
| 3 | prime    | 3                     |
| 2 | prime    | 2                     |
| 9 | square   | 2                     |
| 4 | square   | 2                     |
| 1 | square   | 1                     |
+---+----------+-----------------------+
```

The following examples show how to construct a moving window, with a running minimum taking into account all rows before and 1 row after the current row. Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause. Because of an extra Impala restriction on the MAX() and MIN() functions in an analytic context, the lower bound must be UNBO UNDED PRECEDING.

```
select x, property,
  min(x) over
  (
    order by property, x desc
    rows between unbounded preceding and 1 following
  ) as 'local minimum'
from int_t where property in ('prime','square');
+---+----------+---------------+
| x | property | local minimum |
+---+----------+---------------+
| 7 | prime    | 5             |
| 5 | prime    | 3             |
| 3 | prime    | 2             |
| 2 | prime    | 2             |
| 9 | square   | 2             |
| 4 | square   | 1             |
| 1 | square   | 1             |
+---+----------+---------------+
-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  min(x) over
  (
    order by property, x desc
    range between unbounded preceding and 1 following
  ) as 'local minimum'
from int_t where property in ('prime','square');
ERROR: AnalysisException: RANGE is only supported with both the lower and u
pper bounds UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

NDV function

An aggregate function that returns an approximate value similar to the result of COUNT(DISTINCT *col*), the “number of distinct values”. It is much faster than the combination of COUNT and DISTINCT, and uses a constant amount of memory and thus is less memory-intensive for columns with high cardinality.

Syntax:

```
NDV([DISTINCT | ALL] expression)
```

Usage notes:

This is the mechanism used internally by the COMPUTE STATS statement for computing the number of distinct values in a column.

Because this number is an estimate, it might not reflect the precise number of different values in the column, especially if the cardinality is very low or very high. If the estimated number is higher than the number of rows in the table, Impala adjusts the value internally during query planning.

Return type: DOUBLE in Impala 2.0 and higher; STRING in earlier releases

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex types on page 49 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM() and AVG() are computed using the numeric R_NATIONKEY field, and the general-purpose MAX() and MIN() values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
```

```
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct
_vals |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5
    |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5
    |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5
    |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5
    |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5
    |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
```

Restrictions:

This function cannot be used in an analytic context. That is, the OVER() clause is not allowed at all with this function.

Examples:

The following example queries a billion-row table to illustrate the relative performance of COUNT(DISTINCT) and NDV(). It shows how COUNT(DISTINCT) gives a precise answer, but is inefficient for large-scale data where an approximate result is sufficient. The NDV() function gives an approximate result but is much faster.

```
select count(distinct col1) from sample_data;
+----------------------+
| count(distinct col1)|
+----------------------+
| 100000               |
+----------------------+
Fetched 1 row(s) in 20.13s

select cast(ndv(col1) as bigint) as col1 from sample_data;
+----------+
| col1     |
+----------+
| 139017   |
+----------+
Fetched 1 row(s) in 8.91s
```

The following example shows how you can code multiple NDV() calls in a single query, to easily learn which columns have substantially more or fewer distinct values. This technique is faster than running a sequence of queries with COUNT(DISTINCT) calls.

```
select cast(ndv(col1) as bigint) as col1, cast(ndv(col2) as bigint) as col2,
    cast(ndv(col3) as bigint) as col3, cast(ndv(col4) as bigint) as col4
  from sample_data;
+----------+-----------+------------+-----------+
```

```
| col1     | col2      | col3       | col4      |
+----------+-----------+------------+-----------+
| 139017   | 282       | 46         | 145636240 |
+----------+-----------+------------+-----------+
Fetched 1 row(s) in 34.97s

select count(distinct col1) from sample_data;
+---------------------+
| count(distinct col1)|
+---------------------+
| 100000              |
+---------------------+
Fetched 1 row(s) in 20.13s

select count(distinct col2) from sample_data;
+----------------------+
| count(distinct col2) |
+----------------------+
| 278                  |
+----------------------+
Fetched 1 row(s) in 20.09s
select count(distinct col3) from sample_data;
+----------------------+
| count(distinct col3)  |
+----------------------+
| 46                    |
+----------------------+
Fetched 1 row(s) in 19.12s
select count(distinct col4) from sample_data;
+----------------------+
| count(distinct col4) |
+----------------------+
| 147135880            |
+----------------------+
Fetched 1 row(s) in 266.95s
```

STDDEV, STDDEV_SAMP, STDDEV_POP functions

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

Syntax:

```
{ STDDEV | STDDEV_SAMP | STDDEV_POP } ([DISTINCT | ALL] expression)
```

This function works with any numeric data type.

Return type: DOUBLE in Impala 2.0 and higher; STRING in earlier releases

This function is typically used in mathematical formulas related to probability distributions.

The STDDEV_POP() and STDDEV_SAMP() functions compute the population standard deviation and sample standard deviation, respectively, of the input values. (STDDEV() is an alias for STDDEV_SAMP().) Both functions evaluate all input rows matched by the query. The difference is that STDDEV_SAMP() is scaled by 1/(N-1) while STDDEV_POP() is scaled by 1/N.

If no input rows match the query, the result of any of these functions is NULL. If a single input row matches the query, the result of any of these functions is "0.0".

Examples:

This example demonstrates how STDDEV() and STDDEV_SAMP() return the same result, while STDDEV_POP() uses a slightly different calculation to reflect that the input data is considered part of a larger “population”.

```
[localhost:21000] > select stddev(score) from test_scores;
+---------------+
```

```
| stddev(score) |
+---------------+
| 28.5          |
+---------------+
[localhost:21000] > select stddev_samp(score) from test_scores;
+--------------------+
| stddev_samp(score) |
+--------------------+
| 28.5               |
+--------------------+
[localhost:21000] > select stddev_pop(score) from test_scores;
+-------------------+
| stddev_pop(score) |
+-------------------+
| 28.4858           |
+-------------------+
```

This example demonstrates that, because the return value of these aggregate functions is a STRING, you must currently convert the result with CAST.

```
[localhost:21000] > create table score_stats as select cast(stddev(score) as
 decimal(7,4)) `standard_deviation`, cast(variance(score) as decimal(7,4)) `
variance` from test_scores;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc score_stats;
+--------------------+--------------+---------+
| name               | type         | comment |
+--------------------+--------------+---------+
| standard_deviation | decimal(7,4) |         |
| variance           | decimal(7,4) |         |
+--------------------+--------------+---------+
```

Restrictions:

This function cannot be used in an analytic context. That is, the OVER() clause is not allowed at all with this function.

Related information:

The STDDEV(), STDDEV_POP(), and STDDEV_SAMP() functions compute the standard deviation (square root of the variance) based on the results of VARIANCE(), VARIANCE_POP(), and VARIANCE_SAMP() respectively. See VARIANCE, VARIANCE_SAMP, VARIANCE_POP, VAR_SAMP, VAR_POP functions on page 383 for details about the variance property.

Related Information

Standard deviation

SUM function

An aggregate function that returns the sum of a set of numbers. Its single argument can be numeric column, or the numeric result of a function or expression applied to the column value. Rows with a NULL value for the specified column are ignored. If the table is empty, or all the values supplied to MIN are NULL, SUM returns NULL.

Syntax:

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

When the query contains a GROUP BY clause, returns one value for each combination of grouping values.

Return type: BIGINT for integer arguments, DOUBLE for floating-point arguments

Complex type considerations:

To access a column with a complex type (ARRAY, STRUCT, or MAP) in an aggregation function, you unpack the individual elements using join notation in the query, and then apply the function to the final scalar item, field, key, or value at the bottom of any nested type hierarchy in the column. See Complex types on page 49 for details about using complex types in Impala.

The following example demonstrates calls to several aggregation functions using values from a column containing nested complex types (an ARRAY of STRUCT items). The array is unpacked inside the query using join notation. The array elements are referenced using the ITEM pseudocolumn, and the structure fields inside the array elements are referenced using dot notation. Numeric values such as SUM() and AVG() are computed using the numeric R_NATIONKEY field, and the general-purpose MAX() and MIN() values are computed from the string N_NAME field.

```
describe region;
+-------------+-------------------------+---------+
| name        | type                    | comment |
+-------------+-------------------------+---------+
| r_regionkey | smallint                |         |
| r_name      | string                  |         |
| r_comment   | string                  |         |
| r_nations   | array<struct<           |         |
|             |   n_nationkey:smallint, |         |
|             |   n_name:string,        |         |
|             |   n_comment:string      |         |
|             | >>                      |         |
+-------------+-------------------------+---------+

select r_name, r_nations.item.n_nationkey
  from region, region.r_nations as r_nations
order by r_name, r_nations.item.n_nationkey;
+-------------+------------------+
| r_name      | item.n_nationkey |
+-------------+------------------+
| AFRICA      | 0                |
| AFRICA      | 5                |
| AFRICA      | 14               |
| AFRICA      | 15               |
| AFRICA      | 16               |
| AMERICA     | 1                |
| AMERICA     | 2                |
| AMERICA     | 3                |
| AMERICA     | 17               |
| AMERICA     | 24               |
| ASIA        | 8                |
| ASIA        | 9                |
| ASIA        | 12               |
| ASIA        | 18               |
| ASIA        | 21               |
| EUROPE      | 6                |
| EUROPE      | 7                |
| EUROPE      | 19               |
| EUROPE      | 22               |
| EUROPE      | 23               |
| MIDDLE EAST | 4                |
| MIDDLE EAST | 10               |
| MIDDLE EAST | 11               |
| MIDDLE EAST | 13               |
| MIDDLE EAST | 20               |
+-------------+------------------+

select
  r_name,
  count(r_nations.item.n_nationkey) as count,
```

```
  sum(r_nations.item.n_nationkey) as sum,
  avg(r_nations.item.n_nationkey) as avg,
  min(r_nations.item.n_name) as minimum,
  max(r_nations.item.n_name) as maximum,
  ndv(r_nations.item.n_nationkey) as distinct_vals
from
  region, region.r_nations as r_nations
group by r_name
order by r_name;
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| r_name      | count | sum | avg  | minimum   | maximum        | distinct
_vals |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
| AFRICA      | 5     | 50  | 10   | ALGERIA   | MOZAMBIQUE     | 5
    |
| AMERICA     | 5     | 47  | 9.4  | ARGENTINA | UNITED STATES  | 5
    |
| ASIA        | 5     | 68  | 13.6 | CHINA     | VIETNAM        | 5
    |
| EUROPE      | 5     | 77  | 15.4 | FRANCE    | UNITED KINGDOM | 5
    |
| MIDDLE EAST | 5     | 58  | 11.6 | EGYPT     | SAUDI ARABIA   | 5
    |
+-------------+-------+-----+------+-----------+----------------+-----------
----+
```

Examples:

The following example shows how to use SUM() to compute the total for all the values in the table, a subset of values, or the sum for each combination of values in the GROUP BY clause:

```
-- Total all the values for this column in the table.
select sum(c1) from t1;
-- Find the total for this column from a subset of the table.
select sum(c1) from t1 where month = 'January' and year = '2013';
-- Find the total from a set of numeric function results.
select sum(length(s)) from t1;
-- Often used with functions that return predefined values to compute a s
core.
select sum(case when grade = 'A' then 1.0 when grade = 'B' then 0.75 else 0)
 as class_honors from test_scores;
-- Can also be used in combination with DISTINCT and/or GROUP BY.
-- Return more than one result.
select month, year, sum(purchase_price) from store_stats group by month,
year;
-- Filter the input to eliminate duplicates before performing the calculat
ion.
select sum(distinct x) from t1;
```

The following examples show how to use SUM() in an analytic context. They use a table containing integers from 1 to 10. Notice how the SUM() is reported for each input value, as opposed to the GROUP BY clause which condenses the result set.

```
select x, property, sum(x) over (partition by property) as sum from int_t w
here property in ('odd','even');
+----+----------+-----+
| x  | property | sum |
+----+----------+-----+
| 2  | even     | 30  |
| 4  | even     | 30  |
| 6  | even     | 30  |
```

```
| 8  | even     | 30  |
| 10 | even     | 30  |
| 1  | odd      | 25  |
| 3  | odd      | 25  |
| 5  | odd      | 25  |
| 7  | odd      | 25  |
| 9  | odd      | 25  |
+----+----------+-----+
```

Adding an ORDER BY clause lets you experiment with results that are cumulative or apply to a moving set of rows (the “window”). The following examples use SUM() in an analytic context (that is, with an OVER() clause) to produce a running total of all the even values, then a running total of all the odd values. The basic ORDER BY x clause implicitly activates a window clause of RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, which is effectively the same as ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW, therefore all of these examples produce the same results:

```
select x, property,
  sum(x) over (partition by property order by x) as 'cumulative total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 2  | even     | 2                |
| 4  | even     | 6                |
| 6  | even     | 12               |
| 8  | even     | 20               |
| 10 | even     | 30               |
| 1  | odd      | 1                |
| 3  | odd      | 4                |
| 5  | odd      | 9                |
| 7  | odd      | 16               |
| 9  | odd      | 25               |
+----+----------+------------------+

select x, property,
  sum(x) over
  (
    partition by property
    order by x
    range between unbounded preceding and current row
  ) as 'cumulative total'
from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 2  | even     | 2                |
| 4  | even     | 6                |
| 6  | even     | 12               |
| 8  | even     | 20               |
| 10 | even     | 30               |
| 1  | odd      | 1                |
| 3  | odd      | 4                |
| 5  | odd      | 9                |
| 7  | odd      | 16               |
| 9  | odd      | 25               |
+----+----------+------------------+

select x, property,
  sum(x) over
  (
    partition by property
    order by x
```

```
    rows between unbounded preceding and current row
  ) as 'cumulative total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 2  | even     | 2                |
| 4  | even     | 6                |
| 6  | even     | 12               |
| 8  | even     | 20               |
| 10 | even     | 30               |
| 1  | odd      | 1                |
| 3  | odd      | 4                |
| 5  | odd      | 9                |
| 7  | odd      | 16               |
| 9  | odd      | 25               |
+----+----------+------------------+
```

Changing the direction of the ORDER BY clause causes the intermediate results of the cumulative total to be calculated in a different order:

```
select sum(x) over (partition by property order by x desc) as 'cumulative
 total'
  from int_t where property in ('odd','even');
+----+----------+------------------+
| x  | property | cumulative total |
+----+----------+------------------+
| 10 | even     | 10               |
| 8  | even     | 18               |
| 6  | even     | 24               |
| 4  | even     | 28               |
| 2  | even     | 30               |
| 9  | odd      | 9                |
| 7  | odd      | 16               |
| 5  | odd      | 21               |
| 3  | odd      | 24               |
| 1  | odd      | 25               |
+----+----------+------------------+
```

The following examples show how to construct a moving window, with a running total taking into account 1 row before and 1 row after the current row, within the same partition (all the even values or all the odd values). Because of a restriction in the Impala RANGE syntax, this type of moving window is possible with the ROWS BETWEEN clause but not the RANGE BETWEEN clause:

```
select x, property,
  sum(x) over
  (
    partition by property
    order by x
    rows between 1 preceding and 1 following
  ) as 'moving total'
  from int_t where property in ('odd','even');
+----+----------+--------------+
| x  | property | moving total |
+----+----------+--------------+
| 2  | even     | 6            |
| 4  | even     | 12           |
| 6  | even     | 18           |
| 8  | even     | 24           |
| 10 | even     | 18           |
| 1  | odd      | 4            |
| 3  | odd      | 9            |
```

```
| 5  | odd      | 15           |
| 7  | odd      | 21           |
| 9  | odd      | 16           |
+----+----------+--------------+
-- Doesn't work because of syntax restriction on RANGE clause.
select x, property,
  sum(x) over
  (
    partition by property
    order by x
    range between 1 preceding and 1 following
  ) as 'moving total'
from int_t where property in ('odd','even');
ERROR: AnalysisException: RANGE is only supported with both the lower and up
per bounds UNBOUNDED or one UNBOUNDED and the other CURRENT ROW.
```

Restrictions:

Due to the way arithmetic on FLOAT and DOUBLE columns uses high-performance hardware instructions, and distributed queries can perform these operations in different order for each query, results can vary slightly for aggregate function calls such as SUM() and AVG() for FLOAT and DOUBLE columns, particularly on large data sets where millions or billions of values are summed or averaged. For perfect consistency and repeatability, use the DECIMAL data type for such operations instead of FLOAT or DOUBLE.

VARIANCE, VARIANCE_SAMP, VARIANCE_POP, VAR_SAMP, VAR_POP functions

An aggregate function that returns the variance of a set of numbers. This is a mathematical property that signifies how far the values spread apart from the mean. The return value can be zero (if the input is a single value, or a set of identical values), or a positive number otherwise.

Syntax:

```
{ VARIANCE | VAR[IANCE]_SAMP | VAR[IANCE]_POP } ([DISTINCT | ALL
] expression)
```

This function works with any numeric data type.

Return type: DOUBLE in Impala 2.0 and higher; STRING in earlier releases

This function is typically used in mathematical formulas related to probability distributions.

The VARIANCE_SAMP() and VARIANCE_POP() functions compute the sample variance and population variance, respectively, of the input values. (VARIANCE() is an alias for VARIANCE_SAMP().) Both functions evaluate all input rows matched by the query. The difference is that STDDEV_SAMP() is scaled by 1/(N-1) while STDDEV_P OP() is scaled by 1/N.

The functions VAR_SAMP() and VAR_POP() are the same as VARIANCE_SAMP() and VARIANCE_POP(), respectively. These aliases are available in Impala 2.0 and later.

If no input rows match the query, the result of any of these functions is NULL. If a single input row matches the query, the result of any of these functions is "0.0".

Examples:

This example demonstrates how VARIANCE() and VARIANCE_SAMP() return the same result, while VARIANCE _POP() uses a slightly different calculation to reflect that the input data is considered part of a larger “population”.

```
[localhost:21000] > select variance(score) from test_scores;
+-----------------+
| variance(score) |
+-----------------+
| 812.25          |
+-----------------+
[localhost:21000] > select variance_samp(score) from test_scores;
+----------------------+
```

```
| variance_samp(score) |
+----------------------+
| 812.25               |
+----------------------+
[localhost:21000] > select variance_pop(score) from test_scores;
+----------------------+
| variance_pop(score) |
+----------------------+
| 811.438             |
+----------------------+
```

This example demonstrates that, because the return value of these aggregate functions is a STRING, you convert the result with CAST if you need to do further calculations as a numeric value.

```
[localhost:21000] > create table score_stats as select cast(stddev(score) as
 decimal(7,4)) `standard_deviation`, cast(variance(score) as decimal(7,4)) `
variance` from test_scores;
+-------------------+
| summary           |
+-------------------+
| Inserted 1 row(s) |
+-------------------+
[localhost:21000] > desc score_stats;
+--------------------+--------------+---------+
| name               | type         | comment |
+--------------------+--------------+---------+
| standard_deviation | decimal(7,4) |         |
| variance           | decimal(7,4) |         |
+--------------------+--------------+---------+
```

Restrictions:

This function cannot be used in an analytic context. That is, the OVER() clause is not allowed at all with this function.

Related information:

The STDDEV(), STDDEV_POP(), and STDDEV_SAMP() functions compute the standard deviation (square root of the variance) based on the results of VARIANCE(), VARIANCE_POP(), and VARIANCE_SAMP() respectively. See STDDEV, STDDEV_SAMP, STDDEV_POP functions on page 377 for details about the standard deviation property.

Related Information

Mathematical variance

Impala analytic functions

Analytic functions (also known as window functions) are a special category of built-in functions. Like aggregate functions, they examine the contents of multiple input rows to compute each output value. However, rather than being limited to one result value per GROUP BY group, they operate on *windows* where the input rows are ordered and grouped using flexible conditions expressed through an OVER() clause.

Some functions, such as LAG() and RANK(), can only be used in this analytic context. Some aggregate functions do double duty: when you call the aggregation functions such as MAX(), SUM(), AVG(), and so on with an OVER() clause, they produce an output value for each row, based on computations across other rows in the window.

Although analytic functions often compute the same value you would see from an aggregate function in a GROU P BY query, the analytic functions produce a value for each row in the result set rather than a single value for each group. This flexibility lets you include additional columns in the SELECT list, offering more opportunities for organizing and filtering the result set.

Analytic function calls are only allowed in the SELECT list and in the outermost ORDER BY clause of the query. During query processing, analytic functions are evaluated after other query stages such as joins, WHERE, and GROU P BY,

The rows that are part of each partition are analyzed by computations across an ordered or unordered set of rows. For example, COUNT() and SUM() might be applied to all the rows in the partition, in which case the order of analysis does not matter. The ORDER BY clause might be used inside the OVER() clause to defines the ordering that applies to functions such as LAG() and FIRST_VALUE().

Analytic functions are frequently used in fields such as finance and science to provide trend, outlier, and bucketed analysis for large data sets. You might also see the term “window functions” in database literature, referring to the sequence of rows (the “window”) that the function call applies to, particularly when the OVER clause includes a ROWS or RANGE keyword.

Related Information

OVER
WINDOW
AVG
COUNT
CUME_DIST
DENSE_RANK
FIRST_VALUE
LAG
LAST_VALUE
LEAD
MAX
MIN
NTILE
PERCENT_RANK
RANK
ROW_NUMBER
SUM

OVER

The OVER clause is required for calls to pure analytic functions such as LEAD(), RANK(), and FIRST_VALUE(). When you include an OVER clause with calls to aggregate functions such as MAX(), COUNT(), or SUM(), they operate as analytic functions.

Syntax:

```
function(args) OVER([partition_by_clause] [order_by_clause [window_clause]])
partition_by_clause ::= PARTITION BY expr [, expr ...]
order_by_clause ::= ORDER BY expr  [ASC | DESC] [NULLS FIRST | NULLS LAST]
              [, expr [ASC | DESC] [NULLS FIRST | NULLS LAST] ...]
window_clause: See Window clause
```

PARTITION BY clause:

The PARTITION BY clause acts much like the GROUP BY clause in the outermost block of a query. It divides the rows into groups containing identical values in one or more columns. These logical groups are known as *partitions*. Throughout the discussion of analytic functions, “partitions” refers to the groups produced by the PARTITION BY clause, not to partitioned tables. However, note the following limitation that applies specifically to analytic function calls involving partitioned tables.

In queries involving both analytic functions and partitioned tables, partition pruning only occurs for columns named in the PARTITION BY clause of the analytic function call. For example, if an analytic function query has a clause such as WHERE year=2016, the way to make the query prune all other YEAR partitions is to include

PARTITION BY year in the analytic function call; for example, OVER (PARTITION BY year,*other_columns other_analytic_clauses*).

The sequence of results from an analytic function “resets” for each new partition in the result set. That is, the set of preceding or following rows considered by the analytic function always come from a single partition. Any MAX(), SUM(), ROW_NUMBER(), and so on apply to each partition independently. Omit the PARTITION BY clause to apply the analytic operation to all the rows in the table.

ORDER BY clause:

The ORDER BY clause works much like the ORDER BY clause in the outermost block of a query. It defines the order in which rows are evaluated for the entire input set, or for each group produced by a PARTITION BY clause. You can order by one or multiple expressions, and for each expression optionally choose ascending or descending order and whether nulls come first or last in the sort order. Because this ORDER BY clause only defines the order in which rows are evaluated, if you want the results to be output in a specific order, also include an ORDER BY clause in the outer block of the query.

When the ORDER BY clause is omitted, the analytic function applies to all items in the group produced by the PART ITION BY clause. When the ORDER BY clause is included, the analysis can apply to all or a subset of the items in the group, depending on the optional window clause.

The order in which the rows are analyzed is only defined for those columns specified in ORDER BY clauses.

One difference between the analytic and outer uses of the ORDER BY clause: inside the OVER clause, ORDER BY 1 or other integer value is interpreted as a constant sort value (effectively a no-op) rather than referring to column 1.

Window clause:

The window clause is only allowed in combination with an ORDER BY clause. If the ORDER BY clause is specified but the window clause is not, the default window is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

HBase considerations:

Because HBase tables are optimized for single-row lookups rather than full scans, analytic functions using the OVER () clause are not recommended for HBase tables. Although such queries work, their performance is lower than on comparable tables using HDFS data files.

Parquet considerations:

Analytic functions are very efficient for Parquet tables. The data that is examined during evaluation of the OVER() clause comes from a specified set of columns, and the values for each column are arranged sequentially within each data file.

Text table considerations:

Analytic functions are convenient to use with text tables for exploratory business intelligence. When the volume of data is substantial, prefer to use Parquet tables for performance-critical analytic queries.

Examples:

The following example shows how to synthesize a numeric sequence corresponding to all the rows in a table. The new table has the same columns as the old one, plus an additional column ID containing the integers 1, 2, 3, and so on, corresponding to the order of a TIMESTAMP column in the original table.

```
CREATE TABLE events_with_id AS
  SELECT
    row_number() OVER (ORDER BY date_and_time) AS id,
    c1, c2, c3, c4
  FROM events;
```

The following example shows how to determine the number of rows containing each value for a column. Unlike a corresponding GROUP BY query, this one can analyze a single column and still return all values (not just the distinct ones) from the other columns.

```
SELECT x, y, z,
  count() OVER (PARTITION BY x) AS how_many_x
FROM t1;
```

Restrictions:

You cannot directly combine the DISTINCT operator with analytic function calls. You can put the analytic function call in a WITH clause or an inline view, and apply the DISTINCT operator to its result set.

```
WITH t1 AS (SELECT x, sum(x) OVER (PARTITION BY x) AS total FROM t1)
  SELECT DISTINCT x, total FROM t1;
```

WINDOW

Certain analytic functions accept an optional *window clause*, which makes the function analyze only certain rows "around" the current row rather than all rows in the partition. For example, you can get a moving average by specifying some number of preceding and following rows, or a running count or running total by specifying all rows up to the current position. This clause can result in different analytic results for rows within the same partition.

The window clause is supported with the AVG(), COUNT(), FIRST_VALUE(), LAST_VALUE(), and SUM() functions. For MAX() and MIN(), the window clause only allowed if the start bound is UNBOUNDED PRECED ING

Syntax:

```
ROWS BETWEEN [ { m | UNBOUNDED } PRECEDING | CURRENT ROW] [ AND [CURRENT ROW
 | { UNBOUNDED | n } FOLLOWING] ]
RANGE BETWEEN [ {m | UNBOUNDED } PRECEDING | CURRENT ROW] [ AND [CURRENT ROW
 | { UNBOUNDED | n } FOLLOWING] ]
```

ROWS BETWEEN defines the size of the window in terms of the indexes of the rows in the result set. The size of the window is predictable based on the clauses the position within the result set.

RANGE BETWEEN does not currently support numeric arguments to define a variable-size sliding window.

Currently, Impala supports only some combinations of arguments to the RANGE clause:

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW (the default when ORDER BY is specified and the window clause is omitted)
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

When RANGE is used, CURRENT ROW includes not just the current row but all rows that are tied with the current row based on the ORDER BY expressions.

Examples:

The following examples show financial data for a fictional stock symbol JDR. The closing price moves up and down each day.

```
create table stock_ticker (stock_symbol string, closing_price decimal(8,2),
closing_date timestamp);
...load some data...
select * from stock_ticker order by stock_symbol, closing_date
+--------------+---------------+---------------------+
| stock_symbol | closing_price | closing_date        |
+--------------+---------------+---------------------+
| JDR          | 12.86         | 2014-10-02 00:00:00 |
| JDR          | 12.89         | 2014-10-03 00:00:00 |
```

```
| JDR          | 12.94         | 2014-10-04 00:00:00 |
| JDR          | 12.55         | 2014-10-05 00:00:00 |
| JDR          | 14.03         | 2014-10-06 00:00:00 |
| JDR          | 14.75         | 2014-10-07 00:00:00 |
| JDR          | 13.98         | 2014-10-08 00:00:00 |
+--------------+---------------+---------------------+
```

The queries use analytic functions with window clauses to compute moving averages of the closing price. For example, ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING produces an average of the value from a 3-day span, producing a different value for each row. The first row, which has no preceding row, only gets averaged with the row following it. If the table contained more than one stock symbol, the PARTITION BY clause would limit the window for the moving average to only consider the prices for a single stock.

```
select stock_symbol, closing_date, closing_price,
  avg(closing_price) over (partition by stock_symbol order by closing_date
    rows between 1 preceding and 1 following) as moving_average
  from stock_ticker;
+--------------+---------------------+---------------+----------------+
| stock_symbol | closing_date        | closing_price | moving_average |
+--------------+---------------------+---------------+----------------+
| JDR          | 2014-10-02 00:00:00 | 12.86         | 12.87          |
| JDR          | 2014-10-03 00:00:00 | 12.89         | 12.89          |
| JDR          | 2014-10-04 00:00:00 | 12.94         | 12.79          |
| JDR          | 2014-10-05 00:00:00 | 12.55         | 13.17          |
| JDR          | 2014-10-06 00:00:00 | 14.03         | 13.77          |
| JDR          | 2014-10-07 00:00:00 | 14.75         | 14.25          |
| JDR          | 2014-10-08 00:00:00 | 13.98         | 14.36          |
+--------------+---------------------+---------------+----------------+
```

The clause ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW produces a cumulative moving average, from the earliest data up to the value for each day.

```
select stock_symbol, closing_date, closing_price,
  avg(closing_price) over (partition by stock_symbol order by closing_date
    rows between unbounded preceding and current row) as moving_average
  from stock_ticker;
+--------------+---------------------+---------------+----------------+
| stock_symbol | closing_date        | closing_price | moving_average |
+--------------+---------------------+---------------+----------------+
| JDR          | 2014-10-02 00:00:00 | 12.86         | 12.86          |
| JDR          | 2014-10-03 00:00:00 | 12.89         | 12.87          |
| JDR          | 2014-10-04 00:00:00 | 12.94         | 12.89          |
| JDR          | 2014-10-05 00:00:00 | 12.55         | 12.81          |
| JDR          | 2014-10-06 00:00:00 | 14.03         | 13.05          |
| JDR          | 2014-10-07 00:00:00 | 14.75         | 13.33          |
| JDR          | 2014-10-08 00:00:00 | 13.98         | 13.42          |
+--------------+---------------------+---------------+----------------+
```

AVG

You can include an OVER clause with a call to this function to use it as an analytic function.

Related Information

AVG function

COUNT

You can include an OVER clause with a call to this function to use it as an analytic function.

Related Information

COUNT function

CUME_DIST

Returns the cumulative distribution of a value. The value for each row in the result set is greater than 0 and less than or equal to 1.

Syntax:

```
CUME_DIST (expr)
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Within each partition of the result set, the CUME_DIST() value represents an ascending sequence that ends at 1. Each value represents the proportion of rows in the partition whose values are less than or equal to the value in the current row.

If the sequence of input values contains ties, the CUME_DIST() results are identical for the tied values.

Impala only supports the CUME_DIST() function in an analytic context, not as a regular aggregate function.

Examples:

This example uses a table with 9 rows. The CUME_DIST() function evaluates the entire table because there is no PARTITION BY clause, with the rows ordered by the weight of the animal. the sequence of values shows that 1/9 of the values are less than or equal to the lightest animal (mouse), 2/9 of the values are less than or equal to the second-lightest animal, and so on up to the heaviest animal (elephant), where 9/9 of the rows are less than or equal to its weight.

```
create table animals (name string, kind string, kilos decimal(9,3));
insert into animals values
  ('Elephant', 'Mammal', 4000), ('Giraffe', 'Mammal', 1200), ('Mouse', 'Mamm
al', 0.020),
  ('Condor', 'Bird', 15), ('Horse', 'Mammal', 500), ('Owl', 'Bird', 2.5),
  ('Ostrich', 'Bird', 145), ('Polar bear', 'Mammal', 700), ('Housecat', '
Mammal', 5);

select name, cume_dist() over (order by kilos) from animals;
+------------+-----------------------+
| name       | cume_dist() OVER(...) |
+------------+-----------------------+
| Elephant   | 1                     |
| Giraffe    | 0.8888888888888888    |
| Polar bear | 0.7777777777777778    |
| Horse      | 0.6666666666666666    |
| Ostrich    | 0.5555555555555556    |
| Condor     | 0.4444444444444444    |
| Housecat   | 0.3333333333333333    |
| Owl        | 0.2222222222222222    |
| Mouse      | 0.1111111111111111    |
+------------+-----------------------+
```

Using a PARTITION BY clause produces a separate sequence for each partition group, in this case one for mammals and one for birds. Because there are 3 birds and 6 mammals, the sequence illustrates how 1/3 of the “Bird” rows have a kilos value that is less than or equal to the lightest bird, 1/6 of the “Mammal” rows have a kilos value that is less than or equal to the lightest mammal, and so on until both the heaviest bird and heaviest mammal have a CUME_DIS T() value of 1.

```
select name, kind, cume_dist() over (partition by kind order by kilos) from
animals
+------------+--------+-----------------------+
| name       | kind   | cume_dist() OVER(...) |
+------------+--------+-----------------------+
```

```
| Ostrich    | Bird   | 1                    |
| Condor     | Bird   | 0.6666666666666666   |
| Owl        | Bird   | 0.3333333333333333   |
| Elephant   | Mammal | 1                    |
| Giraffe    | Mammal | 0.8333333333333334   |
| Polar bear | Mammal | 0.6666666666666666   |
| Horse      | Mammal | 0.5                  |
| Housecat   | Mammal | 0.3333333333333333   |
| Mouse      | Mammal | 0.1666666666666667   |
+------------+--------+----------------------+
```

We can reverse the ordering within each partition group by using an ORDER BY ... DESC clause within the OVER() clause. Now the lightest (smallest value of kilos) animal of each kind has a CUME_DIST() value of 1.

```
select name, kind, cume_dist() over (partition by kind order by kilos desc)
from animals
+------------+--------+----------------------+
| name       | kind   | cume_dist() OVER(...) |
+------------+--------+----------------------+
| Owl        | Bird   | 1                    |
| Condor     | Bird   | 0.6666666666666666   |
| Ostrich    | Bird   | 0.3333333333333333   |
| Mouse      | Mammal | 1                    |
| Housecat   | Mammal | 0.8333333333333334   |
| Horse      | Mammal | 0.6666666666666666   |
| Polar bear | Mammal | 0.5                  |
| Giraffe    | Mammal | 0.3333333333333333   |
| Elephant   | Mammal | 0.1666666666666667   |
+------------+--------+----------------------+
```

The following example manufactures some rows with identical values in the kilos column, to demonstrate how the results look in case of tie values. For simplicity, it only shows the CUME_DIST() sequence for the “Bird” rows. Now with 3 rows all with a value of 15, all of those rows have the same CUME_DIST() value. 4/5 of the rows have a value for kilos that is less than or equal to 15.

```
insert into animals values ('California Condor', 'Bird', 15), ('Andean Condo
r', 'Bird', 15)
select name, kind, cume_dist() over (order by kilos) from animals where kind
 = 'Bird';
+-------------------+------+----------------------+
| name              | kind | cume_dist() OVER(...) |
+-------------------+------+----------------------+
| Ostrich           | Bird | 1                    |
| Condor            | Bird | 0.8                  |
| California Condor | Bird | 0.8                  |
| Andean Condor     | Bird | 0.8                  |
| Owl               | Bird | 0.2                  |
+-------------------+------+----------------------+
```

The following example shows how to use an ORDER BY clause in the outer block to order the result set in case of ties. Here, all the “Bird” rows are together, then in descending order by the result of the CUME_DIST() function, and all tied CUME_DIST() values are ordered by the animal name.

```
select name, kind, cume_dist() over (partition by kind order by kilos) as or
dering
  from animals
where
  kind = 'Bird'
order by kind, ordering desc, name;
+-------------------+------+----------+
| name              | kind | ordering |
+-------------------+------+----------+
```

```
| Ostrich           | Bird | 1        |
| Andean Condor     | Bird | 0.8      |
| California Condor | Bird | 0.8      |
| Condor            | Bird | 0.8      |
| Owl               | Bird | 0.2      |
+-------------------+------+----------+
```

DENSE_RANK

Returns an ascending sequence of integers, starting with 1. The output sequence produces duplicate integers for duplicate values of the ORDER BY expressions. After generating duplicate output values for the “tied” input values, the function continues the sequence with the next higher integer. Therefore, the sequence contains duplicates but no gaps when the input contains duplicates. Starts the sequence over for each group produced by the PARTITIONED BY clause.

Syntax:

```
DENSE_RANK() OVER([partition_by_clause] order_by_clause)
```

The PARTITION BY clause is optional. The ORDER BY clause is required. The window clause is not allowed.

Usage notes:

Often used for top-N and bottom-N queries. For example, it could produce a “top 10” report including all the items with the 10 highest values, even if several items tied for 1st place.

Similar to ROW_NUMBER and RANK. These functions differ in how they treat duplicate combinations of values.

Examples:

The following example demonstrates how the DENSE_RANK() function identifies where each value “places” in the result set, producing the same result for duplicate values, but with a strict sequence from 1 to the number of groups. For example, when results are ordered by the X column, both 1 values are tied for first; both 2 values are tied for second; and so on.

```
select x, dense_rank() over(order by x) as rank, property from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | square   |
| 1  | 1    | odd      |
| 2  | 2    | even     |
| 2  | 2    | prime    |
| 3  | 3    | prime    |
| 3  | 3    | odd      |
| 4  | 4    | even     |
| 4  | 4    | square   |
| 5  | 5    | odd      |
| 5  | 5    | prime    |
| 6  | 6    | even     |
| 6  | 6    | perfect  |
| 7  | 7    | lucky    |
| 7  | 7    | lucky    |
| 7  | 7    | lucky    |
| 7  | 7    | odd      |
| 7  | 7    | prime    |
| 8  | 8    | even     |
| 9  | 9    | square   |
| 9  | 9    | odd      |
| 10 | 10   | round    |
| 10 | 10   | even     |
+----+------+----------+
```

The following examples show how the DENSE_RANK() function is affected by the PARTITION property within the ORDER BY clause.

Partitioning by the PROPERTY column groups all the even, odd, and so on values together, and DENSE_RANK() returns the place of each value within the group, producing several ascending sequences.

```
select x, dense_rank() over(partition by property order by x) as rank, prope
rty from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 2  | 1    | even     |
| 4  | 2    | even     |
| 6  | 3    | even     |
| 8  | 4    | even     |
| 10 | 5    | even     |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 1  | 1    | odd      |
| 3  | 2    | odd      |
| 5  | 3    | odd      |
| 7  | 4    | odd      |
| 9  | 5    | odd      |
| 6  | 1    | perfect  |
| 2  | 1    | prime    |
| 3  | 2    | prime    |
| 5  | 3    | prime    |
| 7  | 4    | prime    |
| 10 | 1    | round    |
| 1  | 1    | square   |
| 4  | 2    | square   |
| 9  | 3    | square   |
+----+------+----------+
```

Partitioning by the X column groups all the duplicate numbers together and returns the place each value within the group; because each value occurs only 1 or 2 times, DENSE_RANK() designates each X value as either first or second within its group.

```
select x, dense_rank() over(partition by x order by property) as rank, prope
rty from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | odd      |
| 1  | 2    | square   |
| 2  | 1    | even     |
| 2  | 2    | prime    |
| 3  | 1    | odd      |
| 3  | 2    | prime    |
| 4  | 1    | even     |
| 4  | 2    | square   |
| 5  | 1    | odd      |
| 5  | 2    | prime    |
| 6  | 1    | even     |
| 6  | 2    | perfect  |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 2    | odd      |
| 7  | 3    | prime    |
| 8  | 1    | even     |
| 9  | 1    | odd      |
```

```
| 9  | 2    | square   |
| 10 | 1    | even     |
| 10 | 2    | round    |
+----+------+----------+
```

The following example shows how DENSE_RANK() produces a continuous sequence while still allowing for ties. In this case, Croesus and Midas both have the second largest fortune, while Crassus has the third largest. (In the *RANK function* section, you see a similar query with the RANK() function that shows that while Crassus has the third largest fortune, he is the fourth richest person.)

```
select dense_rank() over (order by net_worth desc) as placement, name, net_w
orth from wealth order by placement, name;
+-----------+---------+---------------+
| placement | name    | net_worth     |
+-----------+---------+---------------+
| 1         | Solomon | 2000000000.00 |
| 2         | Croesus | 1000000000.00 |
| 2         | Midas   | 1000000000.00 |
| 3         | Crassus | 500000000.00  |
| 4         | Scrooge | 80000000.00   |
+-----------+---------+---------------+
```

FIRST_VALUE

Returns the expression value from the first row in the window. If your table has null values, you can use the IGNORE NULLS clause to return the first non-null value from the window. This same value is repeated for all result rows for the group. The return value is NULL if the input expression is NULL.

Syntax:

```
FIRST_VALUE(expr [IGNORE NULLS]) OVER([partition_by_clause] order_by_clause
 [window_clause])
```

The PARTITION BY clause is optional. The ORDER BY clause is required. The window clause is optional.

Usage notes:

If any duplicate values occur in the tuples evaluated by the ORDER BY clause, the result of this function is not deterministic. Consider adding additional ORDER BY columns to ensure consistent ordering.

Examples:

The following example shows a table with a wide variety of country-appropriate greetings. For consistency, we want to standardize on a single greeting for each country. The FIRST_VALUE() function helps to produce a mail merge report where every person from the same country is addressed with the same greeting.

```
select name, country, greeting from mail_merge;
+---------+---------+--------------+
| name    | country | greeting     |
+---------+---------+--------------+
| Pete    | USA     | Hello        |
| John    | USA     | Hi           |
| Boris   | Germany | Guten tag    |
| Michael | Germany | Guten morgen |
| Bjorn   | Sweden  | Hej          |
| Mats    | Sweden  | Tja          |
+---------+---------+--------------+

select country, name,
  first_value(greeting)
    over (partition by country order by name, greeting) as greeting
  from mail_merge;
+---------+---------+-----------+
```

```
| country | name    | greeting  |
+---------+---------+-----------+
| Germany | Boris   | Guten tag |
| Germany | Michael | Guten tag |
| Sweden  | Bjorn   | Hej       |
| Sweden  | Mats    | Hej       |
| USA     | John    | Hi        |
| USA     | Pete    | Hi        |
+---------+---------+-----------+
```

Changing the order in which the names are evaluated changes which greeting is applied to each group.

```
select country, name,
  first_value(greeting)
    over (partition by country order by name desc, greeting) as greeting
  from mail_merge;
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Germany | Michael | Guten morgen |
| Germany | Boris   | Guten morgen |
| Sweden  | Mats    | Tja          |
| Sweden  | Bjorn   | Tja          |
| USA     | Pete    | Hello        |
| USA     | John    | Hello        |
+---------+---------+--------------+
```

If you introduce null values in the mail_merge table, the FIRST_VALUE() function will produce a different result with the IGNORE NULLS clause.

```
select * from mail_merge;
+---------+---------+--------------+
| name    | country | greeting     |
+---------+---------+--------------+
| Boris   | Germany | Guten tag    |
| Peng    | China   | Nihao        |
| Mats    | Sweden  | Tja          |
| Bjorn   | Sweden  | Hej          |
| Kei     | Japan   | NULL         |
| Li      | China   | NULL         |
| John    | USA     | Hi           |
| Pete    | USA     | Hello        |
| Michael | Germany | Guten morgen |
+---------+---------+--------------+

select country, name,
  first_value(greeting ignore nulls)
    over (partition by country order by name,greeting) as greeting
  from mail_merge;
+---------+---------+-----------+
| country | name    | greeting  |
+---------+---------+-----------+
| Japan   | Kei     | NULL      |
| Germany | Boris   | Guten tag |
| Germany | Michael | Guten tag |
| China   | Li      | NULL      |
| China   | Peng    | Nihao     |
| Sweden  | Bjorn   | Hej       |
| Sweden  | Mats    | Hej       |
| USA     | John    | Hi        |
| USA     | Pete    | Hi        |
+---------+---------+-----------+
```

Changing the order in which the names are evaluated changes the result because null values are now encountered in a different order.

```
select country, name,
  first_value(greeting ignore nulls)
    over (partition by country order by name desc, greeting) as greeting
  from mail_merge
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Japan   | Kei     | NULL         |
| China   | Peng    | Nihao        |
| China   | Li      | Nihao        |
| Sweden  | Mats    | Tja          |
| Sweden  | Bjorn   | Tja          |
| USA     | Pete    | Hello        |
| USA     | John    | Hello        |
| Germany | Michael | Guten morgen |
| Germany | Boris   | Guten morgen |
+---------+---------+--------------+
```

LAG

This function returns the value of an expression using column values from a preceding row. You specify an integer offset, which designates a row position some number of rows previous to the current row. Any column references in the expression argument refer to column values from that prior row. Typically, the table contains a time sequence or numeric sequence column that clearly distinguishes the ordering of the rows.

Syntax:

```
LAG (expr [, offset] [, default])
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Sometimes used an an alternative to doing a self-join.

Examples:

The following example uses the same stock data created in the *Window clause* section. For each day, the query prints the closing price alongside the previous day's closing price. The first row for each stock symbol has no previous row, so that LAG() value is NULL.

```
select stock_symbol, closing_date, closing_price,
    lag(closing_price,1) over (partition by stock_symbol order by closing_
date) as "yesterday closing"
  from stock_ticker
    order by closing_date;
+--------------+---------------------+---------------+-------------------+
| stock_symbol | closing_date        | closing_price | yesterday closing |
+--------------+---------------------+---------------+-------------------+
| JDR          | 2014-09-13 00:00:00 | 12.86         | NULL              |
| JDR          | 2014-09-14 00:00:00 | 12.89         | 12.86             |
| JDR          | 2014-09-15 00:00:00 | 12.94         | 12.89             |
| JDR          | 2014-09-16 00:00:00 | 12.55         | 12.94             |
| JDR          | 2014-09-17 00:00:00 | 14.03         | 12.55             |
| JDR          | 2014-09-18 00:00:00 | 14.75         | 14.03             |
| JDR          | 2014-09-19 00:00:00 | 13.98         | 14.75             |
+--------------+---------------------+---------------+-------------------+
```

The following example does an arithmetic operation between the current row and a value from the previous row, to produce a delta value for each day. This example also demonstrates how ORDER BY works independently in the different parts of the query. The ORDER BY closing_date in the OVER clause makes the query analyze the rows in chronological order. Then the outer query block uses ORDER BY closing_date DESC to present the results with the most recent date first.

```
select stock_symbol, closing_date, closing_price,
    cast(
      closing_price - lag(closing_price,1) over
        (partition by stock_symbol order by closing_date)
      as decimal(8,2)
    )
    as "change from yesterday"
  from stock_ticker
    order by closing_date desc;
+--------------+---------------------+--------------+--------------------
--+
| stock_symbol | closing_date        | closing_price | change from yesterday
 |
+--------------+---------------------+--------------+-------------------
----+
| JDR          | 2014-09-19 00:00:00 | 13.98         | -0.76
 |
| JDR          | 2014-09-18 00:00:00 | 14.75         | 0.72
 |
| JDR          | 2014-09-17 00:00:00 | 14.03         | 1.47
 |
| JDR          | 2014-09-16 00:00:00 | 12.55         | -0.38
  |
| JDR          | 2014-09-15 00:00:00 | 12.94         | 0.04
 |
| JDR          | 2014-09-14 00:00:00 | 12.89         | 0.03
 |
| JDR          | 2014-09-13 00:00:00 | 12.86         | NULL
 |
+--------------+---------------------+--------------+-------------------
----+
```

Related information:

This function is the converse of the LEAD function.

LAST_VALUE

Returns the expression value from the last row in the window. If your table has null values, you can use the IGNORE NULLS clause to return the last non-null value from the window. This same value is repeated for all result rows for the group. The return value is NULL if the input expression is NULL.

Syntax:

```
LAST_VALUE(expr [IGNORE NULLS]) OVER([partition_by_clause] order_by_clause
 [window_clause])
```

The PARTITION BY clause is optional. The ORDER BY clause is required. The window clause is optional.

Usage notes:

If any duplicate values occur in the tuples evaluated by the ORDER BY clause, the result of this function is not deterministic. Consider adding additional ORDER BY columns to ensure consistent ordering.

Examples:

The following example uses the same MAIL_MERGE table as in the example for the FIRST_VALUE function. Because the default window when ORDER BY is used is BETWEEN UNBOUNDED PRECEDING AND CURR

ENT ROW, the query requires the UNBOUNDED FOLLOWING to look ahead to subsequent rows and find the last value for each country.

```
select country, name,
  last_value(greeting) over (
    partition by country order by name, greeting
    rows between unbounded preceding and unbounded following
  ) as greeting
  from mail_merge
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Germany | Boris   | Guten morgen |
| Germany | Michael | Guten morgen |
| Sweden  | Bjorn   | Tja          |
| Sweden  | Mats    | Tja          |
| USA     | John    | Hello        |
| USA     | Pete    | Hello        |
+---------+---------+--------------+
```

Introducing null values into the MAIL_MERGE table as in the example for the FIRST VALUE function, the result set changes when you use the IGNORE NULLS clause.

```
select * from mail_merge;
+---------+---------+--------------+
| name    | country | greeting     |
+---------+---------+--------------+
| Kei     | Japan   | NULL         |
| Boris   | Germany | Guten tag    |
| Li      | China   | NULL         |
| Michael | Germany | Guten morgen |
| Bjorn   | Sweden  | Hej          |
| Peng    | China   | Nihao        |
| Pete    | USA     | Hello        |
| Mats    | Sweden  | Tja          |
| John    | USA     | Hi           |
+---------+---------+--------------+

select country, name,
  last_value(greeting ignore nulls) over (
    partition by country order by name, greeting
    rows between unbounded preceding and unbounded following
  ) as greeting
  from mail_merge;
+---------+---------+--------------+
| country | name    | greeting     |
+---------+---------+--------------+
| Japan   | Kei     | NULL         |
| Germany | Boris   | Guten morgen |
| Germany | Michael | Guten morgen |
| China   | Li      | Nihao        |
| China   | Peng    | Nihao        |
| Sweden  | Bjorn   | Tja          |
| Sweden  | Mats    | Tja          |
| USA     | John    | Hello        |
| USA     | Pete    | Hello        |
+---------+---------+--------------+
```

LEAD

This function returns the value of an expression using column values from a following row. You specify an integer offset, which designates a row position some number of rows after to the current row. Any column references in

the expression argument refer to column values from that later row. Typically, the table contains a time sequence or numeric sequence column that clearly distinguishes the ordering of the rows.

Syntax:

```
LEAD (expr [, offset] [, default])
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Sometimes used an an alternative to doing a self-join.

Examples:

The following example uses the same stock data created in the Window clause. The query analyzes the closing price for a stock symbol, and for each day evaluates if the closing price for the following day is higher or lower.

```
select stock_symbol, closing_date, closing_price,
  case
    (lead(closing_price,1)
      over (partition by stock_symbol order by closing_date)
        - closing_price) > 0
    when true then "higher"
    when false then "flat or lower"
  end as "trending"
from stock_ticker
  order by closing_date;
+--------------+---------------------+---------------+---------------+
| stock_symbol | closing_date        | closing_price | trending      |
+--------------+---------------------+---------------+---------------+
| JDR          | 2014-09-13 00:00:00 | 12.86         | higher        |
| JDR          | 2014-09-14 00:00:00 | 12.89         | higher        |
| JDR          | 2014-09-15 00:00:00 | 12.94         | flat or lower |
| JDR          | 2014-09-16 00:00:00 | 12.55         | higher        |
| JDR          | 2014-09-17 00:00:00 | 14.03         | higher        |
| JDR          | 2014-09-18 00:00:00 | 14.75         | flat or lower |
| JDR          | 2014-09-19 00:00:00 | 13.98         | NULL          |
+--------------+---------------------+---------------+---------------+
```

Related information:

This function is the converse of the LAG function.

MAX

You can include an OVER clause with a call to this function to use it as an analytic function.

Related Information

MAX function

MIN

You can include an OVER clause with a call to this function to use it as an analytic function.

Related Information

MIN function

NTILE

Returns the “bucket number” associated with each row, between 1 and the value of an expression. For example, creating 100 buckets puts the lowest 1% of values in the first bucket, while creating 10 buckets puts the lowest 10% of values in the first bucket. Each partition can have a different number of buckets.

Syntax:

```
NTILE (expr [, offset ...]
```

```
  OVER ([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

The “ntile” name is derived from the practice of dividing result sets into fourths (quartile), tenths (decile), and so on. The NTILE() function divides the result set based on an arbitrary percentile value.

The number of buckets must be a positive integer.

The number of items in each bucket is identical or almost so, varying by at most 1. If the number of items does not divide evenly between the buckets, the remaining N items are divided evenly among the first N buckets.

If the number of buckets N is greater than the number of input rows in the partition, then the first N buckets each contain one item, and the remaining buckets are empty.

Examples:

The following example shows divides groups of animals into 4 buckets based on their weight. The ORDER BY ... DESC clause in the OVER() clause means that the heaviest 25% are in the first group, and the lightest 25% are in the fourth group. (The ORDER BY in the outermost part of the query shows how you can order the final result set independently from the order in which the rows are evaluated by the OVER() clause.) Because there are 9 rows in the group, divided into 4 buckets, the first bucket receives the extra item.

```
create table animals (name string, kind string, kilos decimal(9,3));

insert into animals values
  ('Elephant', 'Mammal', 4000), ('Giraffe', 'Mammal', 1200), ('Mouse', 'Mam
mal', 0.020),
  ('Condor', 'Bird', 15), ('Horse', 'Mammal', 500), ('Owl', 'Bird', 2.5),
  ('Ostrich', 'Bird', 145), ('Polar bear', 'Mammal', 700), ('Housecat', 'Mam
mal', 5);

select name, ntile(4) over (order by kilos desc) as quarter
  from animals
order by quarter desc;
+------------+---------+
| name       | quarter |
+------------+---------+
| Owl        | 4       |
| Mouse      | 4       |
| Condor     | 3       |
| Housecat   | 3       |
| Horse      | 2       |
| Ostrich    | 2       |
| Elephant   | 1       |
| Giraffe    | 1       |
| Polar bear | 1       |
+------------+---------+
```

The following examples show how the PARTITION clause works for the NTILE() function. Here, we divide each kind of animal (mammal or bird) into 2 buckets, the heavier half and the lighter half.

```
select name, kind, ntile(2) over (partition by kind order by kilos desc) as
half
  from animals
order by kind;
+------------+--------+------+
| name       | kind   | half |
+------------+--------+------+
| Ostrich    | Bird   | 1    |
| Condor     | Bird   | 1    |
| Owl        | Bird   | 2    |
```

```
| Elephant   | Mammal | 1    |
| Giraffe    | Mammal | 1    |
| Polar bear | Mammal | 1    |
| Horse      | Mammal | 2    |
| Housecat   | Mammal | 2    |
| Mouse      | Mammal | 2    |
+------------+--------+------+
```

Again, the result set can be ordered independently from the analytic evaluation. This next example lists all the animals heaviest to lightest, showing that elephant and giraffe are in the “top half” of mammals by weight, while housecat and mouse are in the “bottom half”.

```
select name, kind, ntile(2) over (partition by kind order by kilos desc) as
half
  from animals
order by kilos desc;
+------------+--------+------+
| name       | kind   | half |
+------------+--------+------+
| Elephant   | Mammal | 1    |
| Giraffe    | Mammal | 1    |
| Polar bear | Mammal | 1    |
| Horse      | Mammal | 2    |
| Ostrich    | Bird   | 1    |
| Condor     | Bird   | 1    |
| Housecat   | Mammal | 2    |
| Owl        | Bird   | 2    |
| Mouse      | Mammal | 2    |
+------------+--------+------+
```

PERCENT_RANK

The PERCENT_RANK() is a window function that calculates the percentile ranking of rows in a result set.

Syntax:

```
PERCENT_RANK (expr)
  OVER ([partition_by_clause] order_by_clause)
```

Calculates the rank, expressed as a percentage, of each row within a group of rows. If rank is the value for that same row from the RANK() function (from 1 to the total number of rows in the partition group), then the PERCENT_RANK() value is calculated as (*rank* - 1) / (*rows_in_group* - 1) . If there is only a single item in the partition group, its PERCENT_RANK() value is 0.

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

This function is similar to the RANK and CUME_DIST() functions: it returns an ascending sequence representing the position of each row within the rows of the same partition group. The actual numeric sequence is calculated differently, and the handling of duplicate (tied) values is different.

The return values range from 0 to 1 inclusive. The first row in each partition group always has the value 0. A NULL value is considered the lowest possible value. In the case of duplicate input values, all the corresponding rows in the result set have an identical value: the lowest PERCENT_RANK() value of those tied rows. (In contrast to CUME_DIST(), where all tied rows have the highest CUME_DIST() value.)

Examples:

The following example uses the same ANIMALS table as the examples for CUME_DIST() and NTILE(), with a few additional rows to illustrate the results where some values are NULL or there is only a single row in a partition group.

```
insert into animals values ('Komodo dragon', 'Reptile', 70);
```

```
insert into animals values ('Unicorn', 'Mythical', NULL);
insert into animals values ('Fire-breathing dragon', 'Mythical', NULL);
```

As with CUME_DIST(), there is an ascending sequence for each kind of animal. For example, the “Birds” and “Mammals” rows each have a PERCENT_RANK() sequence that ranges from 0 to 1. The “Reptile” row has a PERC ENT_RANK() of 0 because that partition group contains only a single item. Both “Mythical” animals have a PERC ENT_RANK() of 0 because a NULL is considered the lowest value within its partition group.

```
select name, kind, percent_rank() over (partition by kind order by kilos) fr
om animals;
+----------------------+----------+--------------------------+
| name                 | kind     | percent_rank() OVER(...) |
+----------------------+----------+--------------------------+
| Mouse                | Mammal   | 0                        |
| Housecat             | Mammal   | 0.2                      |
| Horse                | Mammal   | 0.4                      |
| Polar bear           | Mammal   | 0.6                      |
| Giraffe              | Mammal   | 0.8                      |
| Elephant             | Mammal   | 1                        |
| Komodo dragon        | Reptile  | 0                        |
| Owl                  | Bird     | 0                        |
| California Condor    | Bird     | 0.25                     |
| Andean Condor        | Bird     | 0.25                     |
| Condor               | Bird     | 0.25                     |
| Ostrich              | Bird     | 1                        |
| Fire-breathing dragon | Mythical | 0                        |
| Unicorn              | Mythical | 0                        |
+----------------------+----------+--------------------------+
```

RANK

Returns an ascending sequence of integers, starting with 1. The output sequence produces duplicate integers for duplicate values of the ORDER BY expressions. After generating duplicate output values for the “tied” input values, the function increments the sequence by the number of tied values. Therefore, the sequence contains both duplicates and gaps when the input contains duplicates. Starts the sequence over for each group produced by the PARTITIO NED BY clause.

Syntax:

```
RANK() OVER([partition_by_clause] order_by_clause)
```

The PARTITION BY clause is optional. The ORDER BY clause is required. The window clause is not allowed.

Usage notes:

Often used for top-N and bottom-N queries. For example, it could produce a “top 10” report including several items that were tied for 10th place.

Similar to ROW_NUMBER and DENSE_RANK. These functions differ in how they treat duplicate combinations of values.

Examples:

The following example demonstrates how the RANK() function identifies where each value “places” in the result set, producing the same result for duplicate values, and skipping values in the sequence to account for the number of duplicates. For example, when results are ordered by the X column, both 1 values are tied for first; both 2 values are tied for third; and so on.

```
select x, rank() over(order by x) as rank, property from int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | square   |
```

```
| 1  | 1    | odd      |
| 2  | 3    | even     |
| 2  | 3    | prime    |
| 3  | 5    | prime    |
| 3  | 5    | odd      |
| 4  | 7    | even     |
| 4  | 7    | square   |
| 5  | 9    | odd      |
| 5  | 9    | prime    |
| 6  | 11   | even     |
| 6  | 11   | perfect  |
| 7  | 13   | lucky    |
| 7  | 13   | lucky    |
| 7  | 13   | lucky    |
| 7  | 13   | odd      |
| 7  | 13   | prime    |
| 8  | 18   | even     |
| 9  | 19   | square   |
| 9  | 19   | odd      |
| 10 | 21   | round    |
| 10 | 21   | even     |
+----+------+----------+
```

The following examples show how the RANK() function is affected by the PARTITION property within the ORDER BY clause.

Partitioning by the PROPERTY column groups all the even, odd, and so on values together, and RANK() returns the place of each value within the group, producing several ascending sequences.

```
select x, rank() over(partition by property order by x) as rank, property fr
om int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 2  | 1    | even     |
| 4  | 2    | even     |
| 6  | 3    | even     |
| 8  | 4    | even     |
| 10 | 5    | even     |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 1  | 1    | odd      |
| 3  | 2    | odd      |
| 5  | 3    | odd      |
| 7  | 4    | odd      |
| 9  | 5    | odd      |
| 6  | 1    | perfect  |
| 2  | 1    | prime    |
| 3  | 2    | prime    |
| 5  | 3    | prime    |
| 7  | 4    | prime    |
| 10 | 1    | round    |
| 1  | 1    | square   |
| 4  | 2    | square   |
| 9  | 3    | square   |
+----+------+----------+
```

Partitioning by the X column groups all the duplicate numbers together and returns the place each value within the group; because each value occurs only 1 or 2 times, RANK() designates each X value as either first or second within its group.

```
select x, rank() over(partition by x order by property) as rank, property fr
om int_t;
+----+------+----------+
| x  | rank | property |
+----+------+----------+
| 1  | 1    | odd      |
| 1  | 2    | square   |
| 2  | 1    | even     |
| 2  | 2    | prime    |
| 3  | 1    | odd      |
| 3  | 2    | prime    |
| 4  | 1    | even     |
| 4  | 2    | square   |
| 5  | 1    | odd      |
| 5  | 2    | prime    |
| 6  | 1    | even     |
| 6  | 2    | perfect  |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 1    | lucky    |
| 7  | 4    | odd      |
| 7  | 5    | prime    |
| 8  | 1    | even     |
| 9  | 1    | odd      |
| 9  | 2    | square   |
| 10 | 1    | even     |
| 10 | 2    | round    |
+----+------+----------+
```

The following example shows how a magazine might prepare a list of history's wealthiest people. Croesus and Midas are tied for second, then Crassus is fourth.

```
select rank() over (order by net_worth desc) as rank, name, net_worth from w
ealth order by rank, name;
+------+---------+---------------+
| rank | name    | net_worth     |
+------+---------+---------------+
| 1    | Solomon | 2000000000.00 |
| 2    | Croesus | 1000000000.00 |
| 2    | Midas   | 1000000000.00 |
| 4    | Crassus | 500000000.00  |
| 5    | Scrooge | 80000000.00   |
+------+---------+---------------+
```

ROW_NUMBER

Returns an ascending sequence of integers, starting with 1. Starts the sequence over for each group produced by the PARTITIONED BY clause. The output sequence includes different values for duplicate input values. Therefore, the sequence never contains any duplicates or gaps, regardless of duplicate input values.

Syntax:

```
ROW_NUMBER() OVER([partition_by_clause] order_by_clause)
```

The ORDER BY clause is required. The PARTITION BY clause is optional. The window clause is not allowed.

Usage notes:

Often used for top-N and bottom-N queries where the input values are known to be unique, or precisely N rows are needed regardless of duplicate values.

Because its result value is different for each row in the result set (when used without a PARTITION BY clause), ROW_NUMBER() can be used to synthesize unique numeric ID values, for example for result sets involving unique values or tuples.

Similar to RANK and DENSE_RANK. These functions differ in how they treat duplicate combinations of values.

Examples:

The following example demonstrates how ROW_NUMBER() produces a continuous numeric sequence, even though some values of X are repeated.

```
select x, row_number() over(order by x, property) as row_number, property fr
om int_t;
+----+------------+----------+
| x  | row_number | property |
+----+------------+----------+
| 1  | 1          | odd      |
| 1  | 2          | square   |
| 2  | 3          | even     |
| 2  | 4          | prime    |
| 3  | 5          | odd      |
| 3  | 6          | prime    |
| 4  | 7          | even     |
| 4  | 8          | square   |
| 5  | 9          | odd      |
| 5  | 10         | prime    |
| 6  | 11         | even     |
| 6  | 12         | perfect  |
| 7  | 13         | lucky    |
| 7  | 14         | lucky    |
| 7  | 15         | lucky    |
| 7  | 16         | odd      |
| 7  | 17         | prime    |
| 8  | 18         | even     |
| 9  | 19         | odd      |
| 9  | 20         | square   |
| 10 | 21         | even     |
| 10 | 22         | round    |
+----+------------+----------+
```

The following example shows how a financial institution might assign customer IDs to some of history's wealthiest figures. Although two of the people have identical net worth figures, unique IDs are required for this purpose. ROW_NUMBER() produces a sequence of five different values for the five input rows.

```
select row_number() over (order by net_worth desc) as account_id, name, net_
worth
  from wealth order by account_id, name;
+------------+---------+---------------+
| account_id | name    | net_worth     |
+------------+---------+---------------+
| 1          | Solomon | 2000000000.00 |
| 2          | Croesus | 1000000000.00 |
| 3          | Midas   | 1000000000.00 |
| 4          | Crassus | 500000000.00  |
| 5          | Scrooge | 80000000.00   |
+------------+---------+---------------+
```

SUM

You can include an OVER clause with a call to this function to use it as an analytic function.

Related Information

SUM function

User-defined functions (UDFs)

User-defined functions (frequently abbreviated as UDFs) let you code your own application logic for processing column values during an Impala query. For example, a UDF could perform calculations using an external math library, combine several column values into one, do geospatial calculations, or other kinds of tests and transformations that are outside the scope of the built-in SQL operators and functions.

You can use UDFs to simplify query logic when producing reports, or to transform data in flexible ways when copying from one table to another with the INSERT ... SELECT syntax.

You might be familiar with this feature from other database products, under names such as stored functions or stored routines.

Impala support for UDFs is available in Impala 1.2 and higher:

- In Impala 1.1, using UDFs in a query required using the Hive shell. (Because Impala and Hive share the same metastore database, you could switch to Hive to run just those queries requiring UDFs, then switch back to Impala.)
- Starting in Impala 1.2, Impala can run both high-performance native code UDFs written in C++, and Java-based Hive UDFs that you might already have written.
- Impala can run scalar UDFs that return a single value for each row of the result set, and user-defined aggregate functions (UDAFs) that return a value based on a set of rows. Currently, Impala does not support user-defined table functions (UDTFs) or window functions.

Related Information

Sample UDF and UDAs for Impala
Memoization
UDF concepts
Runtime environment for UDFs
Installing the UDF development package
Writing UDFs
Writing user-defined aggregate functions (UDAFs)
Building and deploying UDFs
Performance considerations for UDFs
Examples of creating and using UDFs
Security considerations for UDFs
Limitations and restrictions for Impala UDFs

UDF concepts

Depending on your use case, you might write all-new functions, reuse Java UDFs that you have already written for Hive, or port Hive Java UDF code to higher-performance native Impala UDFs in C++. You can code either scalar functions for producing results one row at a time, or more complex aggregate functions for doing analysis across. The following sections discuss these different aspects of working with UDFs.

UDFs and UDAFs

Depending on your use case, the user-defined functions (UDFs) you write might accept or produce different numbers of input and output values:

- The most general kind of user-defined function (the one typically referred to by the abbreviation UDF) takes a single input value and produces a single output value. When used in a query, it is called once for each row in the result set. For example:

```
select customer_name, is_frequent_customer(customer_id) from
 customers;
select obfuscate(sensitive_column) from sensitive_data;
```

- A user-defined aggregate function (UDAF) accepts a group of values and returns a single value. You use UDAFs to summarize and condense sets of rows, in the same style as the built-in COUNT, MAX(), SUM(), and AVG() functions. When called in a query that uses the GROU P BY clause, the function is called once for each combination of GROUP BY values. For example:

```
-- Evaluates multiple rows but returns a single value.
select closest_restaurant(latitude, longitude) from places;

-- Evaluates batches of rows and returns a separate value for
 each batch.
select most_profitable_location(store_id, sales, expenses, tax
_rate, depreciation) from franchise_data group by year;
```

- Currently, Impala does not support other categories of user-defined functions, such as user-defined table functions (UDTFs) or window functions.

Native Impala UDFs

Impala supports UDFs written in C++, in addition to supporting existing Hive UDFs written in Java. Cloudera recommends using C++ UDFs because the compiled native code can yield higher performance, with UDF execution time often 10x faster for a C++ UDF than the equivalent Java UDF.

Using Hive UDFs with Impala

Impala can run Java-based user-defined functions (UDFs), originally written for Hive, with no changes, subject to the following conditions:

- The parameters and return value must all use scalar data types supported by Impala. For example, complex or nested types are not supported.
- Hive/Java UDFs must extend org.apache.hadoop.hive.ql.exec.UDF class.
- Currently, Hive UDFs that accept or return the TIMESTAMP type are not supported.
- Prior to Impala 2.5 the return type must be a “Writable” type such as Text or IntWritable, rather than a Java primitive type such as String or int. Otherwise, the UDF returns NULL. In Impala 2.5 and higher, this restriction is lifted, and both UDF arguments and return values can be Java primitive types.
- Hive UDAFs and UDTFs are not supported.
- Typically, a Java UDF will execute several times slower in Impala than the equivalent native UDF written in C++.
- In Impala 2.5 and higher, you can transparently call Hive Java UDFs through Impala, or call Impala Java UDFs through Hive. This feature does not apply to built-in Hive functions. Any Impala Java UDFs created with older versions must be re-created using new CREATE FUNCT ION syntax, without any signature for arguments or the return value.

To take full advantage of the Impala architecture and performance features, you can also write Impala-specific UDFs in C++.

For background about Java-based Hive UDFs, see the Hive documentation for UDF. For examples or tutorials for writing such UDFs, search the web for related blog posts.

The ideal way to understand how to reuse Java-based UDFs (originally written for Hive) with Impala is to take some of the Hive built-in functions (implemented as Java UDFs) and take the

applicable JAR files through the UDF deployment process for Impala, creating new UDFs with different names:

1. Take a copy of the Hive JAR file containing the Hive built-in functions.
2. Use jar tf *jar_file* to see a list of the classes inside the JAR. You will see names like org/apache/hadoop/hive/ql/udf/UDFLower.class and org/apache/hadoop/hive/ql/udf/UDFOPNegative.class. Make a note of the names of the functions you want to experiment with. When you specify the entry points for the Impala CREATE FUNCTION statement, change the slash characters to dots and strip off the .class suffix, for example org.apache.hadoop.hive.ql.udf.UDFLower and org.apache.hadoop.hive.ql.udf.UDFOPNegative.
3. Copy that file to an HDFS location that Impala can read. (In the examples here, we renamed the file to hive-builtins.jar in HDFS for simplicity.)
4. For each Java-based UDF that you want to call through Impala, issue a CREATE FUNCTION statement, with a LOCATION clause containing the full HDFS path of the JAR file, and a SYMBOL clause with the fully qualified name of the class, using dots as separators and without the .class extension. Remember that user-defined functions are associated with a particular database, so issue a USE statement for the appropriate database first, or specify the SQL function name as *db_name*.*function_name*. Use completely new names for the SQL functions, because Impala UDFs cannot have the same name as Impala built-in functions.
5. Call the function from your queries, passing arguments of the correct type to match the function signature. These arguments could be references to columns, arithmetic or other kinds of expressions, the results of CAST functions to ensure correct data types, and so on.

Note:

In Impala 2.9 and higher, you can refresh the user-defined functions (UDFs) that Impala recognizes, at the database level, by running the REFRESH FUNCTIONS statement with the database name as an argument. Java-based UDFs can be added to the metastore database through Hive CREATE FUNCTION statements, and made visible to Impala by subsequently running REFRESH FUNCTIONS. For example:

```
CREATE DATABASE shared_udfs;
USE shared_udfs;
...use CREATE FUNCTION statements in Hive to create so
me Java-based UDFs
   that Impala is not initially aware of...
REFRESH FUNCTIONS shared_udfs;
SELECT udf_created_by_hive(c1) FROM ...
```

Java UDF example: Reusing lower() function

For example, the following `impala-shell` session creates an Impala UDF my_lower() that reuses the Java code for the Hive lower(): built-in function. We cannot call it lower() because Impala does not allow UDFs to have the same name as built-in functions. From SQL, we call the function in a basic way (in a query with no WHERE clause), directly on a column, and on the results of a string expression:

```
[localhost:21000] > create database udfs;
[localhost:21000] > use udfs;
localhost:21000] > create function lower(string) returns string
location '/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.hi
ve.ql.udf.UDFLower';
ERROR: AnalysisException: Function cannot have the same name as a
 builtin: lower
[localhost:21000] > create function my_lower(string) returns s
tring location '/user/hive/udfs/hive.jar' symbol='org.apache.had
oop.hive.ql.udf.UDFLower';
[localhost:21000] > select my_lower('Some String NOT ALREADY LOWE
RCASE');
+----------------------------------------------------+
```

```
| udfs.my_lower('some string not already lowercase') |
+----------------------------------------------------+
| some string not already lowercase                  |
+----------------------------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create table t2 (s string);
[localhost:21000] > insert into t2 values ('lower'),('UPPER'),('I
nit cap'),('CamelCase');
Inserted 4 rows in 2.28s
[localhost:21000] > select * from t2;
+-----------+
| s         |
+-----------+
| lower     |
| UPPER     |
| Init cap  |
| CamelCase |
+-----------+
Returned 4 row(s) in 0.47s
[localhost:21000] > select my_lower(s) from t2;
+------------------+
| udfs.my_lower(s) |
+------------------+
| lower            |
| upper            |
| init cap         |
| camelcase        |
+------------------+
Returned 4 row(s) in 0.54s
[localhost:21000] > select my_lower(concat('ABC ',s,' XYZ')) f
rom t2;
+-----------------------------------------+
| udfs.my_lower(concat('abc ', s, ' xyz')) |
+-----------------------------------------+
| abc lower xyz                           |
| abc upper xyz                           |
| abc init cap xyz                        |
| abc camelcase xyz                       |
+-----------------------------------------+
Returned 4 row(s) in 0.22s
```

Java UDF example: Reusing negative() function

Here is an example that reuses the Hive Java code for the negative() built-in function. This example demonstrates how the data types of the arguments must match precisely with the function signature. At first, we create an Impala SQL function that can only accept an integer argument. Impala cannot find a matching function when the query passes a floating-point argument, although we can call the integer version of the function by casting the argument. Then we overload the same function name to also accept a floating-point argument.

```
[localhost:21000] > create table t (x int);
[localhost:21000] > insert into t values (1), (2), (4), (100);
Inserted 4 rows in 1.43s
[localhost:21000] > create function my_neg(bigint) returns bigin
t location '/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.
hive.ql.udf.UDFOPNegative';
[localhost:21000] > select my_neg(4);
+----------------+
| udfs.my_neg(4) |
+----------------+
| -4             |
+----------------+
[localhost:21000] > select my_neg(x) from t;
```

```
+----------------+
| udfs.my_neg(x) |
+----------------+
| -2             |
| -4             |
| -100           |
+----------------+
Returned 3 row(s) in 0.60s
[localhost:21000] > select my_neg(4.0);
ERROR: AnalysisException: No matching function with signature:
 udfs.my_neg(FLOAT).
[localhost:21000] > select my_neg(cast(4.0 as int));
+------------------------------+
| udfs.my_neg(cast(4.0 as int)) |
+------------------------------+
| -4                           |
+------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > create function my_neg(double) returns double
 location '/user/hive/udfs/hive.jar' symbol='org.apache.hadoop.h
ive.ql.udf.UDFOPNegative';
[localhost:21000] > select my_neg(4.0);
+------------------+
| udfs.my_neg(4.0) |
+------------------+
| -4               |
+------------------+
Returned 1 row(s) in 0.11s
```

You can find the sample files mentioned here in the Impala github repo.

Runtime environment for UDFs

By default, Impala copies UDFs into /tmp, and you can configure this location through the --local_library_dir startup flag for the `impalad` daemon.

Installing the UDF development package

To develop UDFs for Impala, download and install the impala-udf-devel package (RHEL-based distributions). This package contains header files, sample source, and build configuration files.

1. In Cloudera Archive (at https://archive.cloudera.com/cdh7/), locate the appropriate package installation file for your operating system version.

   Access to Cloudera Runtime packages for production purposes requires authentication.
2. Install the Impala UDF package using the package management command, e.g., yum, zypper, or apt-get, depending on your operating system. For the package name, specify impala-udf-devel (RHEL-based distributions) or impala-udf-dev (Ubuntu and Debian).

Note: The UDF development code does not rely on Impala being installed on the same machine. You can write and compile UDFs on a minimal development system, then deploy them on a different one for use with Impala. If you develop UDFs on a server managed by Cloudera Manager through the parcel mechanism, you still install the UDF development kit through the package mechanism; this small standalone package does not interfere with the parcels containing the main Impala code.

When you are ready to start writing your own UDFs, download the sample code and build scripts from the Cloudera sample UDF github. Then see Writing UDFs on page 410 for how to code UDFs, and Examples of creating and using UDFs on page 415 for how to build and run UDFs.

Writing UDFs

Before starting UDF development, make sure to install the development package and download the UDF code samples. Install the Impala UDF development package as described in Installing the UDF development package on page 409.

When writing UDFs:

- Keep in mind the data type differences as you transfer values from the high-level SQL to your lower-level UDF code. For example, in the UDF code you might be much more aware of how many bytes different kinds of integers require.
- Use best practices for function-oriented programming: choose arguments carefully, avoid side effects, make each function do a single thing, and so on.

Getting started with UDF coding

To understand the layout and member variables and functions of the predefined UDF data types, examine the header file /usr/include/impala_udf/udf.h:

```
// This is the only Impala header required to develop UDFs and U
DAs. This header
// contains the types that need to be used and the FunctionCont
ext object. The context
// object serves as the interface object between the UDF/UDA and
 the impala process.
```

For the basic declarations needed to write a scalar UDF, see the header file udf-sample.h within the sample build environment, which defines a simple function named AddUdf():

```
#ifndef IMPALA_UDF_SAMPLE_UDF_H
#define IMPALA_UDF_SAMPLE_UDF_H
#include <impala_udf/udf.h>

using namespace impala_udf;

IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const
 IntVal& arg2);
#endif
```

For sample C++ code for a simple function named AddUdf(), see the source file udf-sample.cc within the sample build environment:

```
#include "udf-sample.h"
// In this sample we are declaring a UDF that adds two ints and
 returns an int.
IntVal AddUdf(FunctionContext* context, const IntVal& arg1, const
 IntVal& arg2) {
  if (arg1.is_null || arg2.is_null) return IntVal::null();
  return IntVal(arg1.val + arg2.val);
}

// Multiple UDFs can be defined in the same file
```

Data types for function arguments and return values

Each value that a user-defined function can accept as an argument or return as a result value must map to a SQL data type that you could specify for a table column.

Currently, Impala UDFs cannot accept arguments or return values of the Impala complex types (STRUCT, ARRAY, or MAP).

Each data type has a corresponding structure defined in the C++ and Java header files, with two member fields and some predefined comparison operators and constructors:

- is_null indicates whether the value is NULL or not. val holds the actual argument or return value when it is non-NULL.
- Each struct also defines a null() member function that constructs an instance of the struct with the is_null flag set.
- The built-in SQL comparison operators and clauses such as <, >=, BETWEEN, and ORDER BY all work automatically based on the SQL return type of each UDF. For example, Impala knows how to evaluate BETWEEN 1 AND udf_returning_int(col1) or ORDER BY udf_returni ng_string(col2) without you declaring any comparison operators within the UDF itself.

  For convenience within your UDF code, each struct defines == and != operators for comparisons with other structs of the same type. These are for typical C++ comparisons within your own code, not necessarily reproducing SQL semantics. For example, if the is_null flag is set in both structs, they compare as equal. That behavior of null comparisons is different from SQL (where NULL == NULL is NULL rather than true), but more in line with typical C++ behavior.
- Each kind of struct has one or more constructors that define a filled-in instance of the struct, optionally with default values.
- Impala cannot process UDFs that accept the composite or nested types as arguments or return them as result values. This limitation applies both to Impala UDFs written in C++ and Java-based Hive UDFs.
- You can overload functions by creating multiple functions with the same SQL name but different argument types. For overloaded functions, you must use different C++ or Java entry point names in the underlying functions.

The data types defined on the C++ side (in /usr/include/impala_udf/udf.h) are:

- IntVal represents an INT column.
- BigIntVal represents a BIGINT column. Even if you do not need the full range of a BIGINT value, it can be useful to code your function arguments as BigIntVal to make it convenient to call the function with different kinds of integer columns and expressions as arguments. Impala automatically casts smaller integer types to larger ones when appropriate, but does not implicitly cast large integer types to smaller ones.
- SmallIntVal represents a SMALLINT column.
- TinyIntVal represents a TINYINT column.
- StringVal represents a STRING column. It has a len field representing the length of the string, and a ptr field pointing to the string data. It has constructors that create a new StringVal struct based on a null-terminated C-style string, or a pointer plus a length; these new structs still refer to the original string data rather than allocating a new buffer for the data. It also has a constructor that takes a pointer to a FunctionContext struct and a length, that does allocate space for a new copy of the string data, for use in UDFs that return string values.
- BooleanVal represents a BOOLEAN column.
- FloatVal represents a FLOAT column.
- DoubleVal represents a DOUBLE column.
- TimestampVal represents a TIMESTAMP column. It has a date field, a 32-bit integer representing the Gregorian date, that is, the days past the epoch date. It also has a time_of_day field, a 64-bit integer representing the current time of day in nanoseconds.

Variable-length argument lists

UDFs typically take a fixed number of arguments, with each one named explicitly in the signature of your C++ function. Your function can also accept additional optional arguments, all of the same type. For example, you can concatenate two strings, three strings, four strings, and so on. Or you can compare two numbers, three numbers, four numbers, and so on.

To accept a variable-length argument list, code the signature of your function like this:

```
StringVal Concat(FunctionContext* context, const StringVal& sepa
rator,
```

```
  int num_var_args, const StringVal* args);
```

In the CREATE FUNCTION statement, after the type of the first optional argument, include ... to indicate it could be followed by more arguments of the same type. For example, the following function accepts a STRING argument, followed by one or more additional STRING arguments:

```
[localhost:21000] > create function my_concat(string, string ...
) returns string location '/user/test_user/udfs/sample.so' symbo
l='Concat';
```

The call from the SQL query must pass at least one argument to the variable-length portion of the argument list.

When Impala calls the function, it fills in the initial set of required arguments, then passes the number of extra arguments and a pointer to the first of those optional arguments.

Handling NULL values

For correctness, performance, and reliability, it is important for each UDF to handle all situations where any NULL values are passed to your function. For example, when passed a NULL, UDFs typically also return NULL. In an aggregate function, which could be passed a combination of real and NULL values, you might make the final value into a NULL (as in CONCAT()), ignore the NULL value (as in AVG()), or treat it the same as a numeric zero or empty string.

Each parameter type, such as IntVal or StringVal, has an is_null Boolean member. Test this flag immediately for each argument to your function, and if it is set, do not refer to the val field of the argument structure. The val field is undefined when the argument is NULL, so your function could go into an infinite loop or produce incorrect results if you skip the special handling for NULL.

If your function returns NULL when passed a NULL value, or in other cases such as when a search string is not found, you can construct a null instance of the return type by using its null() member function.

Memory allocation for UDFs

By default, memory allocated within a UDF is deallocated when the function exits, which could be before the query is finished. The input arguments remain allocated for the lifetime of the function, so you can refer to them in the expressions for your return values. If you use temporary variables to construct all-new string values, use the StringVal() constructor that takes an initial FunctionCont ext* argument followed by a length, and copy the data into the newly allocated memory buffer.

Thread-safe work area for UDFs

One way to improve performance of UDFs is to specify the optional PREPARE_FN and CLOS E_FN clauses on the CREATE FUNCTION statement. The “prepare” function sets up a thread-safe data structure in memory that you can use as a work area. The “close” function deallocates that memory. Each subsequent call to the UDF within the same thread can access that same memory area. There might be several such memory areas allocated on the same host, as UDFs are parallelized using multiple threads.

Within this work area, you can set up predefined lookup tables, or record the results of complex operations on data types such as STRING or TIMESTAMP. Saving the results of previous computations rather than repeating the computation each time is an optimization known as Memoization. For example, if your UDF performs a regular expression match or date manipulation on a column that repeats the same value over and over, you could store the last-computed value or a hash table of already-computed values, and do a fast lookup to find the result for subsequent iterations of the UDF.

Each such function must have the signature:

```
void function_name(impala_udf::FunctionContext*, impala_udf::F
unctionContext::FunctionScope)
```

Currently, only THREAD_SCOPE is implemented, not FRAGMENT_SCOPE. See udf.h for details about the scope values.

Error handling for UDFs

To handle errors in UDFs, you call functions that are members of the initial FunctionContext* argument passed to your function.

A UDF can record one or more warnings, for conditions that indicate minor, recoverable problems that do not cause the query to stop. The signature for this function is:

```
bool AddWarning(const char* warning_msg);
```

For a serious problem that requires cancelling the query, a UDF can set an error flag that prevents the query from returning any results. The signature for this function is:

```
void SetError(const char* error_msg);
```

Writing user-defined aggregate functions (UDAFs)

User-defined aggregate functions (UDAFs or UDAs) are a powerful and flexible category of user-defined functions. If a query processes N rows, calling a UDAF during the query condenses the result set, anywhere from a single value (such as with the SUM or MAX functions), or some number less than or equal to N (as in queries using the GROUP BY or HAVING clause).

The underlying functions for a UDA

A UDAF must maintain a state value across subsequent calls, so that it can accumulate a result across a set of calls, rather than derive it purely from one set of arguments. For that reason, a UDAF is represented by multiple underlying functions:

- An initialization function that sets any counters to zero, creates empty buffers, and does any other one-time setup for a query.
- An update function that processes the arguments for each row in the query result set and accumulates an intermediate result for each node. For example, this function might increment a counter, append to a string buffer, or set flags.
- A merge function that combines the intermediate results from two different nodes.
- A serialize function that flattens any intermediate values containing pointers, and frees any memory allocated during the init, update, and merge phases.
- A finalize function that either passes through the combined result unchanged, or does one final transformation.

In the SQL syntax, you create a UDAF by using the statement CREATE AGGREGATE FUN CTION. You specify the entry points of the underlying C++ functions using the clauses INIT_FN, UPDATE_FN, MERGE_FN, SERIALIZE_FN, and FINALIZE_FN.

For convenience, you can use a naming convention for the underlying functions and Impala automatically recognizes those entry points. Specify the UPDATE_FN clause, using an entry point name containing the string update or Update. When you omit the other _FN clauses from the SQL statement, Impala looks for entry points with names formed by substituting the update or Update portion of the specified name.

uda-sample.h:

See this file online at: uda-sample.h

uda-sample.cc:

See this file online at: uda-sample.cc

Intermediate results for UDAs

A user-defined aggregate function might produce and combine intermediate results during some phases of processing, using a different data type than the final return value. For example, if you implement a function similar to the built-in AVG() function, it must keep track of two values, the number of values counted and the sum of those values. Or, you might accumulate a string value over the course of a UDA, then in the end return a numeric or Boolean result.

In such a case, specify the data type of the intermediate results using the optional INTERMEDIATE *type_name* clause of the CREATE AGGREGATE FUNCTION statement. If the intermediate data is a typeless byte array (for example, to represent a C++ struct or array), specify the type name as CHAR(*n*), with *n* representing the number of bytes in the intermediate result buffer.

For an example of this technique, see the trunc_sum() aggregate function, which accumulates intermediate results of type DOUBLE and returns BIGINT at the end. View the appropriate CREATE FUNCTION statement and the implementation of the underlying TruncSum*() functions on Github.

- test_udfs.py
- test-udas.cc

Building and deploying UDFs

This section explains the steps to compile Impala UDFs from C++ source code, and deploy the resulting libraries for use in Impala queries.

Impala UDF development package ships with a sample build environment for UDFs, that you can study, experiment with, and adapt for your own use.

To build the sample environment:

1. Install the Impala UDF development package as described in Installing the UDF development package on page 409.
2. Run the following commands:

```
cmake .
make
```

The `cmake` configuration command reads the file CMakeLists.txt and generates a Makefile customized for your particular directory paths. Then the `make` command runs the actual build steps based on the rules in the Makefile.

Impala loads the shared library from an HDFS location. After building a shared library containing one or more UDFs, use hdfs dfs or hadoop fs commands to copy the binary file to an HDFS location readable by Impala.

The final step in deployment is to issue a CREATE FUNCTION statement in the `impala-shell` interpreter to make Impala aware of the new function. Because each function is associated with a particular database, always issue a USE statement to the appropriate database before creating a function, or specify a fully qualified name, that is, CREATE FUNCTION *db_name.function_name*.

As you update the UDF code and redeploy updated versions of a shared library, use DROP FUNCTION and CREATE FUNCTION to let Impala pick up the latest version of the code.

Note:

In Impala 2.5 and higher, Impala UDFs and UDAs written in C++ are persisted in the metastore database. Java UDFs are also persisted, if they were created with the new CREATE FUNCTION syntax for Java UDFs, where the Java function argument and return types are omitted. Java-based UDFs created with the old CREATE FUNCTION syntax do not persist across restarts because they are held in the memory of the `catalogd` daemon. Until you re-create such Java UDFs using the new CREATE FUNCTION syntax, you must reload those Java-based UDFs by running the original CREATE FUNCTION statements again each time you restart the `catalogd` daemon. Prior to Impala 2.5 the requirement to reload functions after a restart applied to both C++ and Java functions.

See CREATE FUNCTION statement on page 125 and DROP FUNCTION statement on page 163 for the new syntax for the persistent Java UDFs.

Prerequisites for the build environment are:

1. Install the packages using the appropriate package installation command for your Linux distribution.

```
sudo yum install gcc-c++ cmake boost-devel
sudo yum install impala-udf-devel
# The package name on Ubuntu and Debian is impala-udf-dev.
```

2. Download the UDF sample code:

```
git clone https://github.com/cloudera/impala-udf-samples
cd impala-udf-samples &amp;&amp; cmake . &amp;&amp; make
```

3. Unpack the sample code in udf_samples.tar.gz and use that as a template to set up your build environment.

To build the original samples:

```
# Process CMakeLists.txt and set up appropriate Makefiles.
cmake .
# Generate shared libraries from UDF and UDAF sample code,
# udf_samples/libudfsample.so and udf_samples/libudasample.so
make
```

The sample code to examine, experiment with, and adapt is in these files:

- udf-sample.h: Header file that declares the signature for a scalar UDF (AddUDF).
- udf-sample.cc: Sample source for a simple UDF that adds two integers. Because Impala can reference multiple function entry points from the same shared library, you could add other UDF functions in this file and add their signatures to the corresponding header file.
- udf-sample-test.cc: Basic unit tests for the sample UDF.
- uda-sample.h: Header file that declares the signature for sample aggregate functions. The SQL functions will be called COUNT, AVG, and STRINGCONCAT. Because aggregate functions require more elaborate coding to handle the processing for multiple phases, there are several underlying C++ functions such as CountInit, AvgU pdate, and StringConcatFinalize.
- uda-sample.cc: Sample source for simple UDAFs that demonstrate how to manage the state transitions as the underlying functions are called during the different phases of query processing.
  - The UDAF that imitates the COUNT function keeps track of a single incrementing number; the merge functions combine the intermediate count values from each Impala node, and the combined number is returned verbatim by the finalize function.
  - The UDAF that imitates the AVG function keeps track of two numbers, a count of rows processed and the sum of values for a column. These numbers are updated and merged as with COUNT, then the finalize function divides them to produce and return the final average value.
  - The UDAF that concatenates string values into a comma-separated list demonstrates how to manage storage for a string that increases in length as the function is called for multiple rows.
- uda-sample-test.cc: basic unit tests for the sample UDAFs.

Performance considerations for UDFs

Because a UDF typically processes each row of a table, potentially being called billions of times, the performance of each UDF is a critical factor in the speed of the overall ETL or ELT pipeline. Tiny optimizations you can make within the function body can pay off in a big way when the function is called over and over when processing a huge result set.

Examples of creating and using UDFs

This section demonstrates how to create and use all kinds of user-defined functions (UDFs).

For downloadable examples that you can experiment with, adapt, and use as templates for your own functions, see the Cloudera sample UDF github. You must have already installed the appropriate header files, as explained in Installing the UDF development package on page 409.

Sample C++ UDFs: HasVowels, CountVowels, StripVowels

This example shows 3 separate UDFs that operate on strings and return different data types. In the C++ code, the functions are HasVowels() (checks if a string contains any vowels), CountVowels() (returns the number of vowels in a string), and StripVowels() (returns a new string with vowels removed).

First, we add the signatures for these functions to udf-sample.h in the demo build environment:

```
BooleanVal HasVowels(FunctionContext* context, const StringVal&
input);
IntVal CountVowels(FunctionContext* context, const StringVal& ar
g1);
StringVal StripVowels(FunctionContext* context, const StringVal&
 arg1);
```

Then, we add the bodies of these functions to udf-sample.cc:

```
BooleanVal HasVowels(FunctionContext* context, const StringVal&
input)
{
        if (input.is_null) return BooleanVal::null();

        int index;
        uint8_t *ptr;

        for (ptr = input.ptr, index = 0; index <= input.len; i
ndex++, ptr++)
        {
                uint8_t c = tolower(*ptr);
                if (c == 'a' || c == 'e' || c == 'i' || c == 'o'
 || c == 'u')
                {
                        return BooleanVal(true);
                }
        }
        return BooleanVal(false);
}

IntVal CountVowels(FunctionContext* context, const StringVal&
arg1)
{
        if (arg1.is_null) return IntVal::null();

        int count;
        int index;
        uint8_t *ptr;

        for (ptr = arg1.ptr, count = 0, index = 0; index <= arg1.
len; index++, ptr++)
        {
                uint8_t c = tolower(*ptr);
                if (c == 'a' || c == 'e' || c == 'i' || c == 'o'
 || c == 'u')
                {
                        count++;
                }
        }
        return IntVal(count);
```

```
}

StringVal StripVowels(FunctionContext* context, const StringVal&
 arg1)
{
        if (arg1.is_null) return StringVal::null();

        int index;
        std::string original((const char *)arg1.ptr,arg1.len);
        std::string shorter("");

        for (index = 0; index < original.length(); index++)
        {
                uint8_t c = original[index];
                uint8_t l = tolower(c);

                if (l == 'a' || l == 'e' || l == 'i' || l == 'o'
 || l == 'u')
                {
                        ;
                }
                else
                {
                    shorter.append(1, (char)c);
                }
        }
// The modified string is stored in 'shorter', which is destroyed
 when this function ends. We need to make a string val
// and copy the contents.
        StringVal result(context, shorter.size()); // Only the ve
rsion of the ctor that takes a context object allocates new memo
ry
        memcpy(result.ptr, shorter.c_str(), shorter.size());
        return result;
}
```

We build a shared library, libudfsample.so, and put the library file into HDFS where Impala can read it:

```
$ make
[  0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
[ 33%] Built target udasample
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
Scanning dependencies of target udfsample
[ 83%] Building CXX object CMakeFiles/udfsample.dir/udf-sample.o
Linking CXX shared library udf_samples/libudfsample.so
[ 83%] Built target udfsample
Linking CXX executable udf_samples/udf-sample-test
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudfsample.so /user/hive/udfs/li
budfsample.so
```

Finally, we go into the `impala-shell` interpreter where we set up some sample data, issue CREATE FUNCTION statements to set up the SQL function names, and call the functions in some queries:

```
[localhost:21000] > create database udf_testing;
[localhost:21000] > use udf_testing;
```

```
[localhost:21000] > create function has_vowels (string) returns b
oolean location '/user/hive/udfs/libudfsample.so' symbol='HasVow
els';
[localhost:21000] > select has_vowels('abc');
+-----------------------+
| udfs.has_vowels('abc') |
+-----------------------+
| true                  |
+-----------------------+
Returned 1 row(s) in 0.13s
[localhost:21000] > select has_vowels('zxcvbnm');
+---------------------------+
| udfs.has_vowels('zxcvbnm') |
+---------------------------+
| false                     |
+---------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select has_vowels(null);
+-----------------------+
| udfs.has_vowels(null) |
+-----------------------+
| NULL                  |
+-----------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > select s, has_vowels(s) from t2;
+-----------+--------------------+
| s         | udfs.has_vowels(s) |
+-----------+--------------------+
| lower     | true               |
| UPPER     | true               |
| Init cap  | true               |
| CamelCase | true               |
+-----------+--------------------+
Returned 4 row(s) in 0.24s
[localhost:21000] > create function count_vowels (string) retur
ns int location '/user/hive/udfs/libudfsample.so' symbol='CountV
owels';
[localhost:21000] > select count_vowels('cat in the hat');
+-------------------------------------+
| udfs.count_vowels('cat in the hat') |
+-------------------------------------+
| 4                                   |
+-------------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select s, count_vowels(s) from t2;
+-----------+----------------------+
| s         | udfs.count_vowels(s) |
+-----------+----------------------+
| lower     | 2                    |
| UPPER     | 2                    |
| Init cap  | 3                    |
| CamelCase | 4                    |
+-----------+----------------------+
Returned 4 row(s) in 0.23s
[localhost:21000] > select count_vowels(null);
+------------------------+
| udfs.count_vowels(null) |
+------------------------+
| NULL                   |
+------------------------+
Returned 1 row(s) in 0.12s
```

```
[localhost:21000] > create function strip_vowels (string) returns
 string location '/user/hive/udfs/libudfsample.so' symbol='Strip
Vowels';
[localhost:21000] > select strip_vowels('abcdefg');
+------------------------------+
| udfs.strip_vowels('abcdefg') |
+------------------------------+
| bcdfg                        |
+------------------------------+
Returned 1 row(s) in 0.11s
[localhost:21000] > select strip_vowels('ABCDEFG');
+------------------------------+
| udfs.strip_vowels('abcdefg') |
+------------------------------+
| BCDFG                        |
+------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select strip_vowels(null);
+-------------------------+
| udfs.strip_vowels(null) |
+-------------------------+
| NULL                    |
+-------------------------+
Returned 1 row(s) in 0.16s
[localhost:21000] > select s, strip_vowels(s) from t2;
+-----------+----------------------+
| s         | udfs.strip_vowels(s) |
+-----------+----------------------+
| lower     | lwr                  |
| UPPER     | PPR                  |
| Init cap  | nt cp                |
| CamelCase | CmlCs                |
+-----------+----------------------+
Returned 4 row(s) in 0.24s
```

Sample C++ UDA: SumOfSquares

```
[localhost:21000] > insert overwrite sos values (1, 1), (2, 0),
(3, 1), (4, 0);
Inserted 4 rows in 1.24s

[localhost:21000] > -- Compute 1 squared + 3 squared, and 2 sq
uared + 4 squared;
[localhost:21000] > select y, sum_of_squares(x) from sos group by
 y;
+---+------------------------+
| y | udfs.sum_of_squares(x) |
+---+------------------------+
| 1 | 10                     |
| 0 | 20                     |
+---+------------------------+
Returned 2 row(s) in 0.43s
```

This example demonstrates a user-defined aggregate function (UDA) that produces the sum of the squares of its input values.

The coding for a UDA is a little more involved than a scalar UDF, because the processing is split into several phases, each implemented by a different function. Each phase is relatively straightforward: the “update” and “merge” phases, where most of the work is done, read an input value and combine it with some accumulated intermediate value.

As in our sample UDF from the previous example, we add function signatures to a header file (in this case, uda-sample.h). Because this is a math-oriented UDA, we make two versions of each function, one accepting an integer value and the other accepting a floating-point value.

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val);
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val);

void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal
& input, BigIntVal* val);
void SumOfSquaresUpdate(FunctionContext* context, const Double
Val& input, DoubleVal* val);

void SumOfSquaresMerge(FunctionContext* context, const BigIntV
al& src, BigIntVal* dst);
void SumOfSquaresMerge(FunctionContext* context, const DoubleV
al& src, DoubleVal* dst);

BigIntVal SumOfSquaresFinalize(FunctionContext* context, const Bi
gIntVal& val);
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const Do
ubleVal& val);
```

We add the function bodies to a C++ source file (in this case, uda-sample.cc):

```
void SumOfSquaresInit(FunctionContext* context, BigIntVal* val) {
  val->is_null = false;
  val->val = 0;
}
void SumOfSquaresInit(FunctionContext* context, DoubleVal* val) {
  val->is_null = false;
  val->val = 0.0;
}

void SumOfSquaresUpdate(FunctionContext* context, const BigIntVal
& input, BigIntVal* val) {
  if (input.is_null) return;
  val->val += input.val * input.val;
}
void SumOfSquaresUpdate(FunctionContext* context, const DoubleVal
& input, DoubleVal* val) {
  if (input.is_null) return;
  val->val += input.val * input.val;
}

void SumOfSquaresMerge(FunctionContext* context, const BigIntVal&
 src, BigIntVal* dst) {
  dst->val += src.val;
}
void SumOfSquaresMerge(FunctionContext* context, const DoubleV
al& src, DoubleVal* dst) {
  dst->val += src.val;
}
BigIntVal SumOfSquaresFinalize(FunctionContext* context, const
BigIntVal& val) {
  return val;
}
DoubleVal SumOfSquaresFinalize(FunctionContext* context, const
 DoubleVal& val) {
  return val;
}
```

As with the sample UDF, we build a shared library and put it into HDFS:

```
$ make
[  0%] Generating udf_samples/uda-sample.ll
[ 16%] Built target uda-sample-ir
Scanning dependencies of target udasample
[ 33%] Building CXX object CMakeFiles/udasample.dir/uda-sample.o
Linking CXX shared library udf_samples/libudasample.so
[ 33%] Built target udasample
Scanning dependencies of target uda-sample-test
[ 50%] Building CXX object CMakeFiles/uda-sample-test.dir/uda-s
ample-test.o
Linking CXX executable udf_samples/uda-sample-test
[ 50%] Built target uda-sample-test
[ 50%] Generating udf_samples/udf-sample.ll
[ 66%] Built target udf-sample-ir
[ 83%] Built target udfsample
[100%] Built target udf-sample-test
$ hdfs dfs -put ./udf_samples/libudasample.so /user/hive/udfs/li
budasample.so
```

To create the SQL function, we issue a CREATE AGGREGATE FUNCTION statement and specify the underlying C++ function names for the different phases:

```
[localhost:21000] > use udf_testing;

[localhost:21000] > create table sos (x bigint, y double);
[localhost:21000] > insert into sos values (1, 1.1), (2, 2.2),
(3, 3.3), (4, 4.4);
Inserted 4 rows in 1.10s

[localhost:21000] > create aggregate function sum_of_squares(b
igint) returns bigint
  > location '/user/hive/udfs/libudasample.so'
  > init_fn='SumOfSquaresInit'
  > update_fn='SumOfSquaresUpdate'
  > merge_fn='SumOfSquaresMerge'
  > finalize_fn='SumOfSquaresFinalize';
[localhost:21000] > -- Compute the same value using literals or
 the UDA;
[localhost:21000] > select 1*1 + 2*2 + 3*3 + 4*4;
+-------------------------------+
| 1 * 1 + 2 * 2 + 3 * 3 + 4 * 4 |
+-------------------------------+
| 30                            |
+-------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select sum_of_squares(x) from sos;
+------------------------+
| udfs.sum_of_squares(x) |
+------------------------+
| 30                     |
+------------------------+
Returned 1 row(s) in 0.35s
```

Until we create the overloaded version of the UDA, it can only handle a single data type. To allow it to handle DOUBLE as well as BIGINT, we issue another CREATE AGGREGATE FUNCTION statement:

```
[localhost:21000] > select sum_of_squares(y) from sos;
ERROR: AnalysisException: No matching function with signature: ud
fs.sum_of_squares(DOUBLE).
```

```
[localhost:21000] > create aggregate function sum_of_squares(dou
ble) returns double
  > location '/user/hive/udfs/libudasample.so'
  > init_fn='SumOfSquaresInit'
  > update_fn='SumOfSquaresUpdate'
  > merge_fn='SumOfSquaresMerge'
  > finalize_fn='SumOfSquaresFinalize';

[localhost:21000] > -- Compute the same value using literals or t
he UDA;
[localhost:21000] > select 1.1*1.1 + 2.2*2.2 + 3.3*3.3 + 4.4*4.4;
+-----------------------------------------------+
| 1.1 * 1.1 + 2.2 * 2.2 + 3.3 * 3.3 + 4.4 * 4.4 |
+-----------------------------------------------+
| 36.3                                          |
+-----------------------------------------------+
Returned 1 row(s) in 0.12s
[localhost:21000] > select sum_of_squares(y) from sos;
+------------------------+
| udfs.sum_of_squares(y) |
+------------------------+
| 36.3                   |
+------------------------+
Returned 1 row(s) in 0.35s
```

Typically, you use a UDA in queries with GROUP BY clauses, to produce a result set with a separate aggregate value for each combination of values from the GROUP BY clause. Let's change our sample table to use 0 to indicate rows containing even values, and 1 to flag rows containing odd values. Then the GROUP BY query can return two values, the sum of the squares for the even values, and the sum of the squares for the odd values:

Security considerations for UDFs

When the Impala authorization feature is enabled:

- To call a UDF in a query, you must have the required read privilege for any databases and tables used in the query.
- The CREATE FUNCTION statement requires:
  - The CREATE privilege on the database.
  - The ALL privilege on two URIs where the URIs are:
    - The JAR file on the file system. For example:

      ```
      GRANT ALL ON URI 'file:///path_to_my.jar' TO ROLE my_role;
      ```

    - The JAR on HDFS. For example:

      ```
      GRANT ALL ON URI 'hdfs:///path/to/jar' TO ROLE my_role
      ```

Limitations and restrictions for Impala UDFs

The following limitations and restrictions apply to Impala UDFs in the current release.

- Impala does not support Hive UDFs that accept or return composite or nested types, or other types not available in Impala tables.
- The Hive current_user() function cannot be called from a Java UDF through Impala.

- All Impala UDFs must be deterministic, that is, produce the same output each time when passed the same argument values. For example, an Impala UDF must not call functions such as rand() to produce different values for each invocation. It must not retrieve data from external sources, such as from disk or over the network.
- An Impala UDF must not spawn other threads or processes.
- Prior to Impala 2.5 when the `catalogd` process is restarted, all UDFs become undefined and must be reloaded. In Impala 2.5 and higher, this limitation only applies to older Java UDFs. Re-create those UDFs using the new CREATE FUNCTION syntax for Java UDFs, which excludes the function signature, to remove the limitation entirely.
- Impala currently does not support user-defined table functions (UDTFs).
- The CHAR and VARCHAR types cannot be used as input arguments or return values for UDFs.

SQL transactions in Impala

A transaction is a single logical operation on the data. Impala supports transactions that satisfy a level of consistency that improves the integrity and reliability of the data before and after a transaction.

Specifically, Impala provides atomicity and isolation of insert operations on transactional tables. A single table insert is either committed in full or not committed, and the results of the insert operation are not visible to other query operations until the operation is committed.

For single table, the inserts are ordered, so if Impala doesn't see a committed insert, it won't see any insert committed after it.

For insert-only transactional tables, you can perform the following statements: CREATE TABLE, DROP TABLE, TRUNCATE, INSERT, SELECT

All transactions in Impala automatically commit at the end of the statement. Currently, Impala does not support multi-statement transactions.

Insert-only tables must be the managed and file-format based tables, such as Parquet, Avro, and text.

Note: Impala does not support changing transactional properties of tables. For example, you cannot alter a transactional table to a non-transactional table.

Related Information

CREATE TABLE statement

Impala reserved words

Impala reserves a set of keywords that cannot be used directly as identifiers in Impala SQL.

If you need to use it as an identifier, you must quote it with backticks. For example:

- CREATE TABLE select (x INT): fails
- CREATE TABLE `select` (x INT): succeeds

Because different database systems have different sets of reserved words, and the reserved words change from release to release, carefully consider database, table, and column names to ensure maximum compatibility between products and versions.

Also consider whether your object names are the same as any Hive keywords, and rename or quote any that conflict as you might switch between Impala and Hive when doing analytics and ETL.

To future-proof your code, you should avoid additional words in case they become reserved words if Impala adds features in later releases. This kind of planning can also help to avoid name conflicts in case you port SQL from other systems that have different sets of reserved words. The Future Keyword column in the table below indicates those additional words that you should avoid for table, column, or other object names, even though they are not currently reserved by Impala.

The following is a summary of the process for deciding whether a particular SQL 2016 word is to be reserved in Impala.

- By default, Impala targets to have the same list of reserved words as SQL 2016.
- At the same time, to be compatible with earlier versions of Impala and to avoid breaking existing tables/ workloads, Impala built-in function names are removed from the reserved words list, e.g. COUNT, AVG, as Impala generally does not need to reserve the names of built-in functions for parsing to work.
- For those remaining SQL 2016 reserved words, if a word is likely to be in-use by users of older Impala versions and if there is a low chance of Impala needing to reserve that word in the future, then the word is not reserved.
- Otherwise, the word is reserved in Impala.

The following table lists the reserved words in Impala.

Keyword	Reserved in SQL:2016	Reserved in Impala 3.0 and higher	Future Keyword
abs	X		
acos	X		
add		X	
aggregate		X	
all	X	X	
allocate	X	X	
alter	X	X	
analytic		X	
and	X	X	
anti		X	
any	X	X	
api_version		X	
are	X	X	
array	X	X	
array_agg	X	X	
array_max_cardinality	X	X	
as	X	X	
asc		X	
asensitive	X	X	
asin	X		
asymmetric	X	X	
at	X	X	
atan	X		
atomic	X	X	
authorization	X	X	
avg	X		
avro		X	
backup			X
begin	X		X

begin_frame	X	X	
begin_partition	X	X	
between	X	X	
bigint	X	X	
binary	X	X	
blob	X	X	
block_size		X	
boolean	X	X	
both	X	X	
break			X
browse			X
bulk			X
by	X	X	
cache			
cached		X	
call	X		
called	X	X	
cardinality	X	X	
cascade		X	
cascaded	X	X	
case	X	X	
cast	X	X	
ceil	X		
ceiling	X		
change		X	
char	X	X	
char_length	X		
character	X	X	
character_length	X		
check	X		X
checkpoint			X
class		X	
classifier	X		
clob	X	X	
close	X		X
close_fn		X	
clustered			X
coalesce	X		X
collate	X	X	
collect	X	X	

column	X	X	
columns		X	
comment		X	
commit	X	X	
compression		X	
compute		X	
condition	X	X	
conf			
connect	X	X	
constraint	X	X	
contains	X	X	
continue			X
convert	X	X	
copy	X	X	
corr	X	X	
corresponding	X	X	
cos	X		
cosh	X		
count	X		
covar_pop	X	X	
covar_samp	X	X	
create	X	X	
cross	X	X	
cube	X	X	
cume_dist	X		
current	X	X	
current_catalog	X		
current_date	X	X	
current_default_transform_group	X	X	
current_path	X	X	
current_role	X	X	
current_row	X	X	
current_schema	X	X	
current_time	X	X	
current_timestamp	X		X
current_transform_group_for_type	X	X	
current_user	X		X
cursor	X	X	
cycle	X	X	
data		X	

database		X	
databases		X	
date	X	X	
datetime		X	
day	X		
dayofweek			
dbcc			X
deallocate	X	X	
dec	X	X	
decfloat	X	X	
decimal	X	X	
declare	X	X	
default	X	X	
define	X	X	
delete	X	X	
delimited		X	
dense_rank	X		
deny			X
deref	X	X	
desc		X	
describe	X	X	
deterministic	X	X	
disconnect	X	X	
disk			X
distinct	X	X	
distributed			X
div		X	
double	X	X	
drop	X	X	
dump			X
dynamic	X	X	
each	X	X	
element	X	X	
else	X	X	
empty	X	X	
encoding		X	
end	X	X	
end-exec	X	X	
end_frame	X	X	
end_partition	X	X	

equals	X	X	
errlvl			X
escape	X	X	
escaped		X	
every	X	X	
except	X	X	
exchange			
exec	X	X	
execute	X	X	
exists	X	X	
exit			X
exp	X		
explain		X	
extended		X	
external	X	X	
extract	X		
false	X	X	
fetch	X	X	
fields		X	
file			X
filefactor			X
fileformat		X	
files		X	
filter	X	X	
finalize_fn		X	
first		X	
first_value	X		
float	X	X	
floor	X		
following		X	
for	X	X	
foreign	X	X	
format		X	
formatted		X	
frame_row	X	X	
free	X	X	
freetext			X
from	X	X	
full	X	X	
function	X	X	

functions		X	
fusion	X	X	
get	X	X	
global	X	X	
goto			X
grant	X	X	
group	X	X	
grouping	X	X	
groups	X	X	
hash		X	
having	X	X	
hold	X	X	
holdlock			X
hour	X		
identity	X		X
if		X	
ignore		X	
ilike		X	
import			
in	X	X	
incremental		X	
index			X
indicator	X	X	
init_fn		X	
initial	X	X	
inner	X	X	
inout	X	X	
inpath		X	
insensitive	X	X	
insert	X	X	
int	X	X	
integer	X	X	
intermediate		X	
intersect	X	X	
intersection	X	X	
interval	X	X	
into	X	X	
invalidate		X	
iregexp		X	
is	X	X	

join	X	X	
json_array	X	X	
json_arrayagg	X	X	
json_exists	X	X	
json_object	X	X	
json_objectagg	X	X	
json_query	X	X	
json_table	X	X	
json_table_primitive	X	X	
json_value	X	X	
key			X
kill			X
kudu		X	
lag	X		
language	X		
large	X	X	
last		X	
last_value	X		
lateral	X	X	
lead	X		
leading	X	X	
left	X	X	
less			
like	X	X	
like_regex	X	X	
limit		X	
lineno			X
lines		X	
listagg	X	X	
ln	X		
load		X	
local	X	X	
localtime	X		
localtimestamp	X	X	
location		X	
log	X		
log10	X	X	
lower	X		
macro			
map		X	

match	X	X	
match_number	X	X	
match_recognize	X	X	
matches	X	X	
max	X		
member	X		
merge	X	X	
merge_fn		X	
metadata		X	
method	X	X	
min	X		
minute	X		
mod	X		
modifies	X	X	
module	X		
month	X		
more			
multiset	X	X	
national	X	X	
natural	X	X	
nchar	X	X	
nclob	X	X	
new	X		
no	X	X	
nocheck			X
nonclustered			X
none	X	X	
normalize	X	X	
not	X	X	
nth_value	X	X	
ntile	X		
null	X	X	
nullif	X		X
nulls		X	
numeric	X	X	
occurrences_regex	X	X	
octet_length	X	X	
of	X	X	
off			X
offset	X	X	

offsets			X
old	X		
omit	X	X	
on	X	X	
one	X	X	
only	X	X	
open	X		X
option			X
or	X	X	
order	X	X	
out	X	X	
outer	X	X	
over	X	X	
overlaps	X	X	
overlay	X	X	
overwrite		X	
parameter	X		
parquet		X	
parquetfile		X	
partialscan			
partition	X	X	
partitioned		X	
partitions		X	
pattern	X	X	
per	X	X	
percent	X	X	
percent_rank	X		
percentile_cont	X	X	
percentile_disc	X	X	
period	X		
pivot			X
plan			X
portion	X	X	
position	X	X	
position_regex	X	X	
power	X		
precedes	X	X	
preceding		X	
precision	X		X
prepare	X	X	

prepare_fn		X	
preserve			
primary	X	X	
print			X
proc			X
procedure	X	X	
produced		X	
ptf	X	X	
public			X
purge		X	
raiseerror			X
range	X	X	
rank	X		
rcfile		X	
read			X
reads	X	X	
readtext			X
real	X	X	
reconfigure			X
recover		X	
recursive	X	X	
reduce			
ref	X	X	
references	X	X	
referencing	X	X	
refresh		X	
regexp		X	
regr_avgx	X	X	
regr_avgy	X	X	
regr_count	X	X	
regr_intercept	X	X	
regr_r2	X	X	
regr_slope	X	X	
regr_sxx	X	X	
regr_sxy	X	X	
regr_syy	X	X	
release	X	X	
rename		X	
repeatable		X	
replace		X	

replication		X	
restore			X
restrict		X	
result	X		
return	X		X
returns	X	X	
revert			X
revoke	X	X	
right	X	X	
rlike		X	
role		X	
roles		X	
rollback	X	X	
rollup	X	X	
row	X	X	
row_number	X		
rowcount			X
rows	X	X	
rule			X
running	X	X	
save			X
savepoint	X	X	
schema		X	
schemas		X	
scope	X	X	
scroll	X	X	
search	X	X	
second	X		
securityaudit			X
seek	X	X	
select	X	X	
semi		X	
sensitive	X	X	
sequencefile		X	
serdeproperties		X	
serialize_fn		X	
session_user	X		X
set	X	X	
setuser			X
show	X	X	

shutdown			X
similar	X	X	
sin	X		
sinh	X		
skip	X	X	
smallint	X	X	
some	X	X	
sort		X	
specific	X	X	
specifictype	X	X	
sql	X		
sqlexception	X	X	
sqlstate	X	X	
sqlwarning	X	X	
sqrt	X		
start	X		
static	X	X	
statistics			X
stats		X	
stddev_pop	X		
stddev_samp	X		
stored		X	
straight_join		X	
string		X	
struct		X	
submultiset	X	X	
subset	X	X	
substring	X		
substring_regex	X	X	
succeeds	X	X	
sum	X		
symbol		X	
symmetric	X	X	
system	X		
system_time	X	X	
system_user	X	X	
table	X	X	
tables		X	
tablesample	X	X	
tan	X		

tanh	X		
tblproperties		X	
terminated		X	
textfile		X	
textsize			X
then	X	X	
time	X		
timestamp	X	X	
timezone_hour	X	X	
timezone_minute	X	X	
tinyint		X	
to	X	X	
top			X
trailing	X	X	
tran			X
transform			
translate	X		
translate_regex	X	X	
translation	X	X	
treat	X	X	
trigger	X	X	
trim	X		
trim_array	X	X	
true	X	X	
truncate	X	X	
try_convert			X
uescape	X	X	
unbounded		X	
uncached		X	
union	X	X	
unique	X	X	
uniquejoin			
unknown	X	X	
unnest	X	X	
unpivot			X
update	X	X	
update_fn		X	
updatetext			X
upper	X		
upsert		X	

use		X	
user	X		X
using	X	X	
utc_tmestamp			
value	X		
value_of	X	X	
values	X	X	
var_pop	X		
var_samp	X		
varbinary	X	X	
varchar	X	X	
varying	X	X	
versioning	X	X	
view		X	
views			
waitfor			X
when	X	X	
whenever	X	X	
where	X	X	
while			X
width_bucket	X	X	
window	X	X	
with	X	X	
within	X	X	
without	X	X	
writetext			X
year	X		

SQL differences between Impala and Hive

Impala's SQL syntax follows the SQL-92 standard, and includes extensions, such as built-in functions. Because Impala and Hive share the same Metastore database and their tables are often used interchangeably, this topic covers differences between Impala and Hive in detail.

The current release of Impala does not support the following SQL features that available in Hive SQL:

- Extensibility mechanisms such as TRANSFORM, custom file formats, or custom SerDes.
- XML functions.
- BINARY data type.
- Certain aggregate functions from Hive SQL: covar_pop, covar_samp, corr, percentile, percentile_approx, histogram_numeric, collect_set. Impala supports the set of aggregate functions and analytic functions.
- Lateral views. In Impala 2.3 and higher, Impala supports queries on complex types (STRUCT, ARRAY, or MAP), using join notation rather than the EXPLODE() keyword.

User-defined functions (UDFs) are supported from Impala 1.2 onward.

- Impala supports high-performance UDFs written in C++, as well as reusing some Java-based Hive UDFs.
- Impala supports scalar UDFs and user-defined aggregate functions (UDAFs). Impala does not currently support user-defined table generating functions (UDTFs).
- Only Impala-supported column types are supported in Java-based UDFs.
- The Hive current_user() function cannot be called from a Java UDF through Impala.

Impala does not currently support these Hive SQL statements:

- ANALYZE TABLE (the Impala equivalent is COMPUTE STATS)
- DESCRIBE COLUMN
- DESCRIBE DATABASE
- EXPORT TABLE
- IMPORT TABLE
- SHOW TABLE EXTENDED
- SHOW TBLPROPERTIES
- SHOW INDEXES
- SHOW COLUMNS
- INSERT OVERWRITE DIRECTORY; use INSERT OVERWRITE *table_name* or CREATE TABLE AS SELECT to materialize query results into the HDFS directory associated with an Impala table.

Impala respects the serialization.null.format table property only for TEXT tables and ignores the property for Parquet and other formats. Hive respects the serialization.null.format property for Parquet and other formats and converts matching values to NULL during the scan. See the *Data files for text tables* section for using the table property in Impala.

Semantic differences between Impala and Hive SQL features

This section covers instances where Impala and Hive have similar functionality, sometimes including the same syntax, but there are differences in the runtime semantics of those features.

SQL statements and clauses:

- Impala uses different syntax and names for query hints, [SHUFFLE] and [NOSHUFFLE] rather than MapJoin or StreamJoin.
- Impala does not expose MapReduce specific features of SORT BY, DISTRIBUTE BY, or CLUSTER BY.
- Impala does not require queries to include a FROM clause.

Data types:

- Impala supports a limited set of implicit casts. This can help avoid undesired results from unexpected casting behavior.
  - Impala does not implicitly cast between string and numeric or Boolean types. Always use CAST() for these conversions.
  - Impala does perform implicit casts among the numeric types, when going from a smaller or less precise type to a larger or more precise one. For example, Impala will implicitly convert a SMALLINT to a BIGINT or FLOAT, but to convert from DOUBLE to FLOAT or INT to TINYINT requires a call to CAST() in the query.
  - Impala does perform implicit casts from STRING to TIMESTAMP. Impala has a restricted set of literal formats for the TIMESTAMP data type and the FROM_UNIXTIME() format string.

  See the topics under Impala SQL data types on page 11 for full details on implicit and explicit casting for all types, and Impala type conversion functions on page 308 for details about the CAST() function.
- Impala does not store or interpret TIMESTAMP values using the local timezone, to avoid undesired results from unexpected time zone issues. Timestamps are stored and interpreted relative to UTC. This difference can produce different results for some calls to similarly named date/time functions between Impala and Hive. See TIMESTAMP data type on page 40 for a discussion of how Impala handles time zones, and configuration options you can use to make Impala match the Hive behavior more closely when dealing with Parquet-encoded TIMESTAMP data or when converting between the local time zone and UTC.

- The Impala TIMESTAMP type can represent dates ranging from 1400-01-01 to 9999-12-31. This is different from the Hive date range, which is 0000-01-01 to 9999-12-31.
- Impala does not return column overflows as NULL, so that customers can distinguish between NULL data and overflow conditions similar to how they do so with traditional database systems. Impala returns the largest or smallest value in the range for the type. For example, valid values for a tinyint range from -128 to 127. In Impala, a tinyint with a value of -200 returns -128 rather than NULL. A tinyint with a value of 200 returns 127.

Miscellaneous features:

- Impala does not provide virtual columns.
- Impala does not expose locking.
- Impala does not expose some configuration properties.

Related Information

Adapting SQL code from a variety of database systems to Impala
Using text files in Impala

Porting SQL from other database systems to Impala

Although Impala uses standard SQL for queries, you might need to modify SQL source when bringing applications to Impala, due to variations in data types, built-in functions, vendor language extensions, and Hadoop-specific syntax. Even when SQL is working correctly, you might make further minor modifications for best performance.

Porting DDL and DML statements

When adapting SQL code from a legacy database system to Impala, expect to find a number of differences in the DDL statements that you use to set up the schema. Clauses related to physical layout of files, tablespaces, and indexes have no equivalent in Impala. You might restructure your schema considerably to account for the Impala partitioning scheme and Hadoop file formats.

Expect SQL queries to have a much higher degree of compatibility. With modest rewriting to address vendor extensions and features not yet supported in Impala, you might be able to run identical or almost-identical query text on both systems.

Therefore, consider separating out the DDL into a separate Impala-specific setup script. Focus your reuse and ongoing tuning efforts on the code for SQL queries.

Porting data types from other database systems

- Change any VARCHAR, VARCHAR2, and CHAR columns to STRING. Remove any length constraints from the column declarations; for example, change VARCHAR(32) or CHAR(1) to STRING. Impala is very flexible about the length of string values; it does not impose any length constraints or do any special processing (such as blank-padding) for STRING columns. (In Impala 2.0 and higher, there are data types VARCHAR and CHAR, with length constraints for both types and blank-padding for CHAR. However, for performance reasons, it is still preferable to use STRING columns where practical.)
- For national language character types such as NCHAR, NVARCHAR, or NCLOB, be aware that while Impala can store and query UTF-8 character data, currently some string manipulation operations only work correctly with ASCII data.
- Change any DATETIME or TIME columns to DATE or TIMESTAMP. Remove any precision constraints. Remove any timezone clauses, and make sure your application logic or ETL process accounts for the fact that Impala expects all TIMESTAMP values to be in the Coordinated Universal Time (UTC). See TIMESTAMP data type on page 40 for information about the TIMESTAMP data type, and Impala date and time functions on page 311 for conversion functions for different date and time formats.

  You might also need to adapt date- and time-related literal values and format strings to use the supported Impala date and time formats. If you have date and time literals with different separators or different numbers of YY, MM, and so on placeholders than Impala expects, consider using calls to regexp_replace() to transform those values to the Impala-compatible format. See TIMESTAMP data type on page 40 for information about the

allowed formats for date and time literals, and Impala string functions on page 335 for string conversion functions such as regexp_replace().

Instead of SYSDATE, call the function NOW().

Instead of adding or subtracting directly from a date value to produce a value *N* days in the past or future, use an INTERVAL expression, for example NOW() + INTERVAL 30 DAYS.

- Although Impala supports INTERVAL expressions for datetime arithmetic, as shown in TIMESTAMP data type on page 40, INTERVAL is not available as a column data type in Impala. For any INTERVAL values stored in tables, convert them to numeric values that you can add or subtract using the functions in Impala date and time functions on page 311. For example, if you had a table DEADLINES with an INT column TIME_PERIOD, you could construct dates N days in the future like so:

```
SELECT NOW() + INTERVAL time_period DAYS from deadlines;
```

- For YEAR columns, change to the smallest Impala integer type that has sufficient range. See Impala SQL data types on page 11 for details about ranges, casting, and so on for the various numeric data types.
- Change any DECIMAL and NUMBER types. If fixed-point precision is not required, you can use FLOAT or DOUBLE on the Impala side depending on the range of values. For applications that require precise decimal values, such as financial data, you might need to make more extensive changes to table structure and application logic, such as using separate integer columns for dollars and cents, or encoding numbers as string values and writing UDFs to manipulate them.
- FLOAT, DOUBLE, and REAL types are supported in Impala. Remove any precision and scale specifications. (In Impala, REAL is just an alias for DOUBLE; columns declared as REAL are turned into DOUBLE behind the scenes.)
- Most integer types from other systems have equivalents in Impala, perhaps under different names such as BIGINT instead of INT8. For any that are unavailable, for example MEDIUMINT, switch to the smallest Impala integer type that has sufficient range. Remove any precision specifications.
- Remove any UNSIGNED constraints. All Impala numeric types are signed.
- For any types holding bitwise values, use an integer type with enough range to hold all the relevant bits within a positive integer.

  For example, TINYINT has a maximum positive value of 127, not 256, so to manipulate 8-bit bitfields as positive numbers switch to the next largest type SMALLINT. For example, CAST(127*2 AS SMALLINT).

  Impala does not support notation such as b'0101' for bit literals.
- For BLOB values, use STRING to represent CLOB or TEXT types (character based large objects) up to 32 KB in size. Binary large objects such as BLOB, RAW BINARY, and VARBINARY do not currently have an equivalent in Impala.
- For Boolean-like types such as BOOL, use the Impala BOOLEAN type.
- Because Impala currently does not support composite or nested types, any spatial data types in other database systems do not have direct equivalents in Impala. You could represent spatial values in string format and write UDFs to process them. Where practical, separate spatial types into separate tables so that Impala can still work with the non-spatial data.
- Take out any DEFAULT clauses. Impala can use data files produced from many different sources, such as Pig, Hive, or MapReduce jobs. The fast import mechanisms of LOAD DATA and external tables mean that Impala is flexible about the format of data files, and Impala does not necessarily validate or cleanse data before querying it. When copying data through Impala INSERT statements, you can use conditional functions such as CASE or NVL to substitute some other value for NULL fields.
- Take out any constraints from your CREATE TABLE and ALTER TABLE statements, for example PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, UNSIGNED, or CHECK constraints. Impala can use data files produced from many different sources, such as Pig, Hive, or MapReduce jobs. Therefore, Impala expects initial data validation to happen earlier during the ETL or ELT cycle. After data is loaded into Impala tables, you can perform queries to test for NULL values. When copying data through Impala INSERT statements, you can use conditional functions such as CASE or NVL to substitute some other value for NULL fields.

  Do as much verification as practical before loading data into Impala. After data is loaded into Impala, you can do further verification using SQL queries to check if values have expected ranges, if values are NULL or not, and so

on. If there is a problem with the data, you will need to re-run earlier stages of the ETL process, or do an INSE RT ... SELECT statement in Impala to copy the faulty data to a new table and transform or filter out the bad values.

- Take out any CREATE INDEX, DROP INDEX, and ALTER INDEX statements, and equivalent ALTER TA BLE statements. Remove any INDEX, KEY, or PRIMARY KEY clauses from CREATE TABLE and ALTER TABLE statements. Impala is optimized for bulk read operations for data warehouse-style queries, and therefore does not support indexes for its tables.
- Calls to built-in functions with out-of-range or otherwise incorrect arguments, return NULL in Impala as opposed to raising exceptions. (This rule applies even when the ABORT_ON_ERROR=true query option is in effect.) Run small-scale queries using representative data to doublecheck that calls to built-in functions are returning expected values rather than NULL. For example, unsupported CAST operations do not raise an error in Impala:
- For any other type not supported in Impala, you could represent their values in string format and write UDFs to process them.
- To detect the presence of unsupported or unconvertable data types in data files, do initial testing with the ABOR T_ON_ERROR=true query option in effect. This option causes queries to fail immediately if they encounter disallowed type conversions.

SQL statements to remove or adapt

The following SQL statements or clauses are not currently supported or supported with limitations in Impala:

- Impala supports the DELETE statement only for Kudu tables.

  Impala is intended for data warehouse-style operations where you do bulk moves and transforms of large quantities of data. When not using Kudu tables, instead of DELETE, use INSERT OVERWRITE to entirely replace the contents of a table or partition, or use INSERT ... SELECT to copy a subset of data (everything but the rows you intended to delete) from one table to another.
- Impala supports the UPDATE statement only for Kudu tables.

  When not using Kudu tables, instead of UPDATE, do all necessary transformations early in the ETL process, such as in the job that generates the original data, or when copying from one table to another to convert to a particular file format or partitioning scheme.
- Impala has no transactional statements, such as COMMIT or ROLLBACK.

  Impala effectively works like the AUTOCOMMIT mode in some database systems, where changes take effect as soon as they are made.
- If your database, table, column, or other names conflict with Impala reserved words, use different names or quote the names with backticks.

  Conversely, if you use a keyword that Impala does not recognize, it might be interpreted as a table or column alias.

  For example, in SELECT * FROM t1 NATURAL JOIN t2, Impala does not recognize the NATURAL keyword and interprets it as an alias for the table t1. If you experience any unexpected behavior with queries, check the list of reserved words to make sure all keywords in join and WHERE clauses are supported keywords in Impala.
- Impala has some restrictions on subquery support.
- Impala supports UNION and UNION ALL set operators, but not INTERSECT.

  Prefer UNION ALL over UNION when you know the data sets are disjoint or duplicate values are not a problem; UNION ALL is more efficient because it avoids materializing and sorting the entire result set to eliminate duplicate values.

- Impala requires query aliases for the subqueries used as inline views in the FROM clause.

  For example, without the alias contents_of_t1 at the end, the following query gives a syntax error:

  ```
  SELECT COUNT(*) FROM (SELECT * FROM t1) contents_of_t1;
  ```

  Aliases are not required for the subqueries used in other parts of queries. For example:

  ```
  SELECT * FROM functional.alltypes WHERE id = (SELECT MIN(id) FROM functi
  onal.alltypes);
  ```
- When an alias is declared for an expression in a query, that alias cannot be referenced again within the same SELE CT list.

  For example, the average alias cannot be referenced twice in the SELECT list as below. You will receive an error:

  ```
  SELECT AVG(x) AS average, average+1 FROM t1 GROUP BY x;
  ```

  An alias can be referenced again in the same query if not in the SELECT list. For example, the average alias can be referenced twice as shown below:

  ```
  SELECT AVG(x) AS average FROM t1 GROUP BY x HAVING average > 3;
  ```
- Impala does not support NATURAL JOIN, and it does not support the USING clause in joins.
- Impala supports a limited choice of partitioning types.

  Partitions are defined based on each distinct combination of values for one or more partition key columns. Impala does not redistribute or check data to create evenly distributed partitions. You must choose partition key columns based on your knowledge of the data volume and distribution. Adapt any tables that use range, list, hash, or key partitioning to use the Impala partition syntax for CREATE TABLE and ALTER TABLE statements.

  Impala partitioning is similar to range partitioning where every range has exactly one value, or key partitioning where the hash function produces a separate bucket for every combination of key values.

  

  Note: Because the number of separate partitions is potentially higher than in other database systems, keep a close eye on the number of partitions and the volume of data in each one; scale back the number of partition key columns if you end up with too many partitions with a small volume of data in each one.

  To distribute work for a query across a cluster, you need at least one HDFS block per node. HDFS blocks are typically multiple megabytes, especially for Parquet files. Therefore, if each partition holds only a few megabytes of data, you are unlikely to see much parallelism in the query because such a small amount of data is typically processed by a single node.
- For the “top-N” queries, Impala uses the LIMIT clause rather than comparing against a pseudo column named ROWNUM or ROW_NUM.

SQL constructs to double-check

Some SQL constructs that are supported have behavior or defaults more oriented towards convenience than optimal performance. Also, sometimes machine-generated SQL, perhaps issued through JDBC or ODBC applications, might have inefficiencies or exceed internal Impala limits. As you port SQL code, examine and possibly update the following where appropriate:

- A CREATE TABLE statement with no STORED AS clause creates data files in plain text format, which is convenient for data interchange but not a good choice for high-volume data with high-performance queries.
- Adapting tables that were already partitioned in a different database system could produce an Impala table with a high number of partitions and not enough data in each one, leading to underutilization of Impala's parallel query features.

- The INSERT ... VALUES syntax is suitable for setting up toy tables with a few rows for functional testing when used with HDFS. Each such statement creates a separate tiny file in HDFS, and it is not a scalable technique for loading megabytes or gigabytes (let alone petabytes) of data.

  Consider revising your data load process to produce raw data files outside of Impala, then setting up Impala external tables or using the LOAD DATA statement to use those data files instantly in Impala tables, with no conversion or indexing stage.

  INSERT works fine for Kudu tables even though not particularly fast.
- If your ETL process is not optimized for Hadoop, you might end up with highly fragmented small data files, or a single giant data file that cannot take advantage of distributed parallel queries or partitioning. In this case, use an INSERT ... SELECT statement to copy the data into a new table and reorganize into a more efficient layout in the same operation.

  You can do INSERT ... SELECT into a table with a more efficient file format or from an unpartitioned table into a partitioned one.
- Complex queries may have high codegen time. As a workaround, set the query option DISABLE_CODEGEN= true if queries fail for this reason.
- If practical, rewrite UNION queries to use the UNION ALL operator instead. Prefer UNION ALL over UNION when you know the data sets are disjoint or duplicate values are not a problem; UNION ALL is more efficient because it avoids materializing and sorting the entire result set to eliminate duplicate values.

Next porting steps after verifying syntax and semantics

Some of the decisions you make during the porting process can have an impact on performance. After your SQL code is ported and working correctly, examine the performance-related aspects of your schema design, physical layout, and queries to make sure that the ported application is taking full advantage of Impala's parallelism, performance-related SQL features, and integration with Hadoop components. The following are a few of the areas you should examine:

- For the optimal performance, we recommend that you run COMPUTE STATS on all tables.
- Use the most efficient file format for your data volumes, table structure, and query characteristics.
- Partition on columns that are often used for filtering in WHERE clauses.
- Your ETL process should produce a relatively small number of multi-megabyte data files rather than a huge number of small files.

Related Information

Impala conditional functions

ABORT_ON_ERROR query option

How to use specific file formats for compact data and high-performance queries

Most heavily optimized file format for large-scale data warehouse queries