Table Management Introduction

Jethro organizes tables and views into schemas. The default schema is called ***def_schema***. Currently, creating new schemas is disabled.

Working with Tables

Creating a Table

To create a table, run a **CREATE TABLE** statement and provide the table name and list of columns.
For example:

```
CREATE TABLE my_table (a INT, b STRING, c TIMESTAMP);
```

For further information on, and examples of, partitioned tables, see Working with Partitioned Tables.

The supported **data types** are:
INT | BIGINT | FLOAT | DOUBLE | STRING | TIMESTAMP
For more details on each data type, see **Data Types** under SQL Language Elements.

Jethro automatically creates indexes for all columns – there is no need to run CREATE INDEX commands.

Renaming a Table

To rename a table, run an ALTER TABLE … RENAME TO statement. For example:

```
ALTER TABLE my_table RENAME TO test_table;
```

Adding and Dropping Columns from a Table

To add a new column to a table:

1. Run an **ALTER TABLE ... ADD COLUMN** statement.
2. Specify a new column name and data type;

for example:

```
ALTER TABLE test_table ADD COLUMN number_of_cats INTEGER;
```

Note: Existing rows will have NULL value in this new column.

To drop an existing column from a table, run an ALTER TABLE ... DROP COLUMN statement. For example:

```
ALTER TABLE test_table DROP COLUMN number_of_cats;
```

Dropping a Table

To drop a table, run a **DROP TABLE** statement. For example:

```
DROP TABLE test_table;
```

Alternatively, run a **TRUNCATE TABLE** statement to delete all rows from a table while keeping the table. For example:

```
TRUNCATE TABLE test_table;
```

Listing All Tables

To list all tables, run the **SHOW TABLES** statement:

```
SHOW TABLES;
TABLE
-----------------------
test1
test2
test3
test4
-----------------------
```

Unlike Hive and Impala, **in Jethro the above command lists only tables, not views. To list views, use SHOW VIEWS command**.
In addition, to get a detailed report of all tables including number of columns, number of rows, number of partitions and size on disk, run the **SHOW TABLES EXTENDED** statement (the output was slightly trimmed to fit into the page width):

```
SHOW TABLES EXTENDED;
```

ID	Schema Name	Table Name	Columns	Rows	Partitions	Columns-MB	Indexes-MB
16	def_Schema	test1	9	1920800	0	4.598	26.078
92	def_Schema	test2	3	20	0	0.003	0.003
299	def_Schema	test3	4	11745000	0	37.338	44.478
453	TOTAL	test4	2	0	0	0.000	0.000
227		test5	23	2880404	63	166.178	510.365

Because this command analyzes the instance current storage usage, its execution can take a while.

Listing Table Columns

To list all columns of a specific table and their data type, run the **DESCRIBE** statement:

```
DESCRIBE my_table;
Column | Type ------------------ id | INTEGER
V | string -------------------
```

In addition, to get a detailed report of a table's columns, including number of distinct values, number of NULLs and size on disk, run the **SHOW TABLE COLUMNS** *table_name* statement (the output was slightly trimmed to fit into the page width):

```
SHOW TABLE COLUMNS my_table;
ID | Column | Type | Nulls |Distinct| Total(MB)| Index (MB)| Column(MB)| Keys (MB)
---------
13 | event_type | INTEGER | 0 | 4 | 1.419 | 0.631 | 0.788 | 0.000
14 | event_ts | TIMESTAMP| 2370142| 930 | 1.026 | 0.105 | 0.921 | 0.000
15 | client_id | string | 0 | 2243288| 108.219 | 31.067 | 6.052 | 71.099
17 | class | INTEGER | 1288168| 181 | 4.184 | 2.009 | 2.174 | 0.000
18 | event_type | BIGINT | 2334246| 6343 | 0.879 | 0.518 | 0.360 | 0.000
-------------------------------------------------------------------------------------------
```

Working with Views

A view is a virtual table, based on the result of an SQL query. While a view contains rows and columns just like an ordinary base table, it does not form part of a physical schema; removing (dropping) a view has no effect on the view's underlying data. Views are very useful for presenting only the requested data; for example, hiding the complexity a query that joins several tables by coding the query's logic into a view, or displaying only the non-confidential rows /columns of table by saving as a view the results of a query that selects theses columns.

Creating a View

To create a view, run a **CREATE VIEW** statement and provide a view name, an optional list of column name for the view, and the view's query text:

```
CREATE VIEW v1 AS
     SELECT region, count event_count FROM events GROUP BY region;
CREATE VIEW v1 (region, event_count) AS
      SELECT region, count FROM events GROUP BY region;
```

Each column in the view must have a unique and valid name. To give proper name to columns that are based on functions and expressions, either use column aliases in the query (as in the first example above) or specify a column list in the CREATE VIEW statement (as in the second example).

Dropping a View

To drop a view, run a **DROP VIEW** statement. For example:

```
DROP VIEW v1;
```

Listing All Views

To list all views, run the **SHOW VIEWS** statement:

```
SHOW VIEWS;
View
----------------
V
V1
V2
----------------
```

In addition, to get a detailed report of all views, including number of columns, status and definition, run the **SHOW VIEWS EXTENDED** statement:

```
SHOW VIEWS EXTENDED;
ID |Schema Name|View Name| Columns |Status | View Statement ----------------------------------------
182| def_schema| V | 19 | valid | create view v1 as select * from events
196| def_schema| V1 | 2 | valid | create view v2 as SELECT region,count event_count FROM events GROUP BY
region; 261 | def_schema| V1 | 3 | invalid | create view v3 (a,b,c) as select * from a_table_that_got_dropped
----------------------------------------------------------------------------------------------------
```

Listing a View's Columns and Definition

To list all columns of a specific view and their data type, run the **DESCRIBE** statement. The command also prints the CREATE VIEW statement that was used for creating the view:

```
DESCRIBE my_view;
Column | Type ------------------ id | INTEGER V | string -------------------
View query: | CREATE VIEW v1 AS
SELECT region, count event_count FROM events GROUP BY region;
```

View Invalidation

When you create a view, Jethro verifies that the view's query is valid. However, after the view is created, its query may become invalid. For example, a view may refer to a table (or another view) that was dropped or modified after the view's creation.
Jethro automatically maintains the status of each view. When the view's query becomes invalid, the view itself changes its status to INVALID and cannot be used. If the view's query becomes valid and matches the view's definition, the view will automatically become valid again.
The current status of each view can be displayed by running the command:

```
SHOW VIEWS EXTENDED
```

Working with Partitioned Tables

Partitioning physically splits a very large table ("fact table") into smaller pieces, based on user-specified criteria.

Why Use Partitioning?

- Partitioning enables **rolling window operations** **Rolling window operation refers to the archiving and removal (purging) of old data, when new data is added to the data warehouse. This practice is commonly used when deciding to keep only a few years of data, thereby preventing the data in the data warehouse from growing indefinitely.**(maintenance). For example, if a table is partitioned by a timestamp column, thereby adding one partition per day, you can implement **DELETE** by dropping a daily partition.

  Because Jethro format is append-only, dropping a partition is the only way to delete data.

- In many databases, partitioning is considered a critical **performance boosting** tool, because it allows scanning less data during a full-table scan. **However**, because in Jethro all queries use the indexes to read only the relevant rows' a query will access the same number of rows regardless of whether the table is partitioned. As a result, partitioning is not a major performance feature for Jethro.
- Partitioning also enables better **scalability**. If a query needs to access a specific subset of the table, based on the partitioning key (for example, a single day), partitioning helps to isolate the requested subset from the actual size of the table – the query will consider a similarly sized subset of the table regardless of the table's retention (one month / year / decade of data).

Which Partitioning Types are Supported?

Jethro supports only range partitioning. In addition, only a single partitioning key is supported. Additional options are not needed, as partitioning is used mostly for rolling window operations and not as a tool for minimizing I/O (we use the indexes for that!)

How do I Choose a Partition Key?
The partitioning key should be the main timestamp column that is used both for maintenance (keeping data for *n* days) and in queries. That column data type is typically TIMESTAMP, but occasionally it is a generated key (usually INT) from a date dimension.
In most cases, the large tables hold events (calls, messages, page views, clicks and so on) and the likely partitioning key is the event timestamp, which indicates when the event took place.

How Large Should Each Partition Be?
Generally, you should align the partition range to the retention policy.
For example:

- If you plan to keep data for 12 months, purging once a month, start with monthly partitions.
- If you plan to keep data for 60 days, purging once a day, start with daily partitions.

However, it is also recommended to aim for a typical partition size of a few billion rows. Many small partitions are inefficient and may overwhelm the HDFS NameNode with too many small files. A few extra-large partitions of many billions of rows each are harder to maintain – for example, harder to correct the data after loading one bad file. Therefore:

- If you have a few billion rows per month, partition by month.
- If you have a few billion rows per day, partition by day.
- If you have a few billion rows per hour, partition by hour.

How does Jethro Handle a Partition Key with NULL Values?
If the partition key has NULL values, Jethro will create a dedicated partition for those rows. This is part of the normal processing.

How Complex is the Partitioning Process?
In Jethro, partitioning is automatic. You just need to specify a partitioning policy in the CREATE TABLE statement; Jethro creates the actual partitions on the fly, during data load. In addition, partitions do not have names, so there is need to manage them. When you drop a partition, you just reference it by value. For details, see "Dropping a Partition" on page .

Creating a Partitioned Table

Creating a partitioned table requires providing two additional parameters – the column to be used as a range partitioning key (in the **PARTITION BY RANGE** clause), and the interval that defines the boundaries of each partition (in the **EVERY** clause).
The way you specify the interval depends on the data type of the partitioning key:

- Examples with **timestamp** partition key:

CREATE TABLE events (id BIGINT, event_ts TIMESTAMP, value INT) PARTITION BY RANGE (event_ts) EVERY (INTERVAL '1' day);
CREATE TABLE events (id BIGINT, event_ts TIMESTAMP, value INT) PARTITION BY RANGE (event_ts) EVERY (INTERVAL '3' month);

- Example with numeric partition key:

CREATE TABLE events (id BIGINT, event_day_id INT, value INT) PARTITION BY RANGE (event_day_id) EVERY (1);

- Examples with **string** partition key:

Note: This option is not recommended - you should always keep timestamps in a timestamp data type.
CREATE TABLE events (id BIGINT, event_date STRING, value INT) PARTITION BY RANGE (event_date) EVERY (VALUE);

Adding a Partition

Adding new partitions is performed automatically, on-the- fly, by JethroLoader as it loads new rows. As a result, there is no direct command to add a new partition, and no option (or need) to create empty partitions in advance.

Concurrency: Adding new partitions by JethroLoader is transparent to concurrently running queries. Loading new rows does not block queries, even if it creates new partitions. Once the JethroLoader has completed successfully, new queries will recognize the new partitions and their data.

Listing Partitions

To list all partitions of a table, run:

```
SHOW TABLE PARTITIONS table_name
```

This command lists the existing partitions, their range, row count, and disk space.

If the partition column is INT or BIGINT, the partition End Value column is inclusive – the end value is included in the partition range.

```
SHOW TABLE PARTITIONS my_table;
ID | Start Value| End Value | Rows |Column(MB) | Index(MB) ---------
p1 | NULL | NULL | 129600843 | 5902.987 | 5807.455
p2 | 2450611 | 2450975 | 138421504 | 6336.308 | 6752.216
p3 | 2450976 | 2451340 | 550095759 | 25077.495 | 24262.893
p4 | 2451341 | 2451705 | 550783528 | 25110.166 | 24306.149
p5 | 2452436 | 2452800 | 412451934 | 25083.587 | 24282.732
p6 | 2452071 | 2452435 | 550208601 | 25083.587 | 24282.732
p7 | 2451706 | 2452070 | 548425830 | 25000.232 | 24208.061
----------------------------------------------------------------------
```

If the partitioning column is TIMESTAMP, FLOAT or DOUBLE, the partition **End Value** column is exclusive – the end value is not included in the partition range. For example, with yearly partitions:

```
SHOW TABLE PARTITIONS my_table2;
ID | Start Value | End Value | Rows |Column(MB)| Index(MB)
-------------------------------------------------------------------------------
p1 | 1998-01-01 00:00:00 | 1999-01-01 00:00:00 | 19791072 | 809.944 | 556.415
p2 | 1999-01-01 00:00:00 | 2000-01-01 00:00:00 | 20024903 | 819.405 | 562.247
p3 | 2000-01-01 00:00:00 | 2001-01-01 00:00:00 | 20086331 | 822.694 | 564.032
p4 | 2001-01-01 00:00:00 | 2002-01-01 00:00:00 | 19891053 | 815.420 | 558.946
p5 | 2002-01-01 00:00:00 | 2003-01-01 00:00:00 | 19989221 | 821.732 | 561.340
p6 | 2003-01-01 00:00:00 | 2004-01-01 00:00:00 | 217420 | 9.483 | 11.599
-------------------------------------------------------------------------------
```

Dropping a Partition

Dropping a partition is carried out by using the ALTER TABLE command. To identify which partition to drop, specify any value within the partition range. Jethro will find in which partition the value resides and will drop that partition. For example:

```
ALTER TABLE my_table DROP PARTITION FOR ('2014-01-01 00:00:00');
```

Concurrency: Dropping a partition can take up to few minutes, as it updates several table-level data structures. However, just like when partitions are added, this is transparent to concurrently running queries – dropping a partition does not block queries. Once the DROP PARTITION completes successfully, new queries will stop seeing the dropped partition and its data.