JethroData Reference Guide

Version: 2.0.0
Updated: 15-Nov-2016

1 Installation Process

Jethro installation is pretty simple. It runs as root, extracts Jethro software to /opt/jethro, and automatically creates and configures the jethro O/S user and Linux service. To install Jethro:

1) Connect as root and download the Jethro RPM to a local directory:

   ```
   wget http://www.jethro.io/latest-rpm
   ```

2) Install the RPM you downloaded with the following command:

   ```
   rpm -Uvh latest-rpm
   ```

3) Change the owner of the Jethro cache directory to the jethro user (the user was created during the RPM installation):

   ```
   chown jethro:jethro /mnt/jethro_local_cache
   ```

4) If using Kerberos, securely copy the keytab file that was generated in section 3.2 to the jethro home directory and change its permissions accordingly:

   ```
   scp kdc_owner@kdc_server:jethro.hadoop.keytab /home/jethro
   chown jethro:jethro /home/jethro/jethro.hadoop.keytab
   chmod 600 /home/jethro/jethro.hadoop.keytab
   ```

5) Optionally, set the password for the jethro O/S user:

   ```
   passwd jethro
   ```

6) The rest of the setup is done as the jethro O/S user:

   ```
   su - jethro
   ```

To install the RPM under a different user than the default one (jethro), run:

```
JETHRO_INSTALL_USER=<user> JETHRO_INSTALL_GROUP=<group> rpm -i <rpm
file>
```

Notes:

- If a non-default user is to be used, then the user must already exist.
- If a non-default user is specified, the group must also be specified.
- If the rpm is already installed, the same user must be used.
- If a new user is going to be used for an installation over an existing installation, then the rpm package should be removed first (existing instances will need to be attached to, directories or mount points of storage and cache will need to be manually chown'd).

2 Instances

A Jethro instance includes an entire database – all the tables, data, indexes and metadata. The instance is stored on a chosen type of storage, shared (such as HDFS, EFS etc') or local (POSIX). An instance can be attached to a server, but can also be detached and reattached later (by the same server or another).

2.1 Creating and Attaching Instances

Creating an instance

To create a new, empty instance, run the JethroAdmin create-instance command and provide the following parameters:

- An instance name of your choice (demo in the following example).
- An HDFS storage path (An HDFS directory owned by user jethro) or a Local storage path.
- Local caching parameters – A local root path and maximum size for the cache directory.
- When using a local storage path, an extra parameter is required: -Dstorage.type=POSIX.

The basic command:

```
JethroAdmin create-instance < instance-name > -storage-path=<storage-path>
-cache-path=<cache-path> -cache-size=80G [-Dstorage.type=POSIX]
```

For example – Local instance creation:

```
JethroAdmin create-instance demo -storage-path=/user/jethro/instances
-cache-path=/mnt/jethro_local_cache -cache-size=80G -Dstorage.type=POSIX
```

HDFS instance creation:

```
JethroAdmin create-instance demo -storage-path=/jethro/instances
-cache-path=/mnt/jethro_local_cache -cache-size=80G
```

Attaching an instance

Instances can be attached/detached from a server. When using a shared storage, they can (and most probably will) be attached to multiple servers. To attach an existing instance, use the following command:

```
JethroAdmin attach-instance < instance-name > -storage-path=<storage-path>
-cache-path=<cache-path> -cache-size=80G [-Dstorage.type=POSIX]
```

Detaching an instance

Detaching an instance from a server require much less parameters:

```
JethroAdmin datach-instance < instance-name >
```

Delete an instance

To remove an instance and all of its data and metadata, use the following command:

```
 JethroAdmin delete-instance < instance-name >
```

2.2 Managing Instances

By server: JethroData server might be attached with a number of instances. The typical case would be an attachment with one instance. To view the list of locally attached instances, run the following command:

```
JethroAdmin list-instances
```

By storage: Instances in Jethro are based in their storage. They don't have to be attached to any server at any times, and they might be attached to one or more than one server at a time. This means that when a user work on one server of Jethro, the other server might not know about this instance. Therefore, in order to view a list of instances by storage, run the following command:

```
JethroAdmin list-storage-instances -storage-path=[storage-root-path]
-Dstorage.type=[HDFS|POSIX]
```

For example:

JethroAdmin list-storage-instances -storage-path=/home/jethro/instances -Dstorage.type=POSIX

To add/remove an instance from the auto start configurations of a server service, edit:

```
/opt/jethro/instances/services.ini
```

3 Managing Tables and Views

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

3.1 Working with Tables

Creating a Table

To create a regular (unpartitioned) 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 partitioned tables, see section Partitioned Tables for discussion and examples.

The supported **data types** are:

```
INT | BIGINT | FLOAT | DOUBLE | STRING | TIMESTAMP
```

For more details on each data type, see section Data Types.

JethroData indexes all columns automatically – there is no need to run CREATE INDEX commands.

Renaming a Table

To rename a table, run a **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, run a **ALTER TABLE … ADD COLUMN** statement and specify a new column name and data type. Existing rows will have NULL value in this new column. For example:

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

To drop an existing column from a table, run a 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;
```

List All Tables

To list all tables (optionally, limited to a specific schema), run the **SHOW TABLES** statement:

```
SHOW TABLES;
```

```
TABLE
------------------------
test1
test2
test3
test4
test5
------------------------
```

Unlike Hive and Impala, in JethroData the above command list only tables, not views. To list views, use SHOW VIEWS command, as discussed in the next section.

In addition, to get a detailed report of all tables, including number of columns, number of rows, number of partitions and size of 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 | def_schema  | test4      |       2 |        0 |          0 |      0.000 |      0.000
 227 | def_schema  | test5      |      23 |  2880404 |         63 |    166.178 |    510.365
-------------------------------------------------------------------------------------------
```

As this command analyzes the instance current storage usage, it may take up to a few minutes

List a Table's 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 of 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 | subclass   | BIGINT    |  2334246 |     6343 |      0.879 |      0.518 |       0.360 |     0.000
----------------------------------------------------------------------------------------------------
```

3.2 Working with Views

Views are a stored sub-query definition (metadata-only). When used in a query, it is replaced with a sub-query containing its text.

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 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 (optionally, limited to a specific schema), 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 TABLES 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  | v2        |       3 | invalid | create view v3 (a,b,c) as select * from
a_table_that_got_dropped
-------------------------------------------------------------------------------
```

List 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 TABLE statement that was used to create 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, JethroData verifies that the view's query is valid. However, after the view was created, its query may become invalid. For example, a view may refer to a table (or another view) that was dropped or changed after the view's creation.

JethroData automatically maintains the status of each view. When the view's query becomes invalid, the view itself moves to INVALID status and cannot be used. If the view's query becomes valid and matches the view definition, the view will become valid automatically.

The current status of each view can be seen using `SHOW VIEWS EXTENDED` command.

3.3 Partitioned Tables

Concepts

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** (maintenance). For example, if a table is partitioned by a timestamp column – let's say one partition per day – then you can implement DELETE by dropping a daily partition.
  Since JethroData format is append-only, dropping a partition is the only way to delete data.
- In many databases, partitioning is considered a critical **performance** tool as it allows them to scan less data when they full scan a table. **However**, in JethroData,, all queries use the indexes to read only the relevant rows, so a query will access the same number of rows regardless of if it is partitioned or not. So, for JethroData partitioning is not a major performance feature.
- 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 it 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).

Partitioning Types – JethroData only supports range partitioning. Also, it only support a single partitioning key. Additional options are not needed as partitioning is used mostly for rolling window operations, not as a tool to minimize I/O (we use the indexes for that!)

How to 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 etc) and the likely partitioning key is the event timestamp – when did the event happen.

How big 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. So,

- 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.

Partition key and NULL. If the partition key has NULL values, JethroData will create a dedicated partition for those rows. This is part of the normal processing.

Simple and Automatic. In JethroData, partitioning is automatic. You just need to specify a partitioning policy in the CREATE TABLE statement. JethroData creates the actual partitions on the fly during data load. Also, partitions don't have names, so you don't need to manage them. When you drop a partition, you just reference it by value (see below).

Creating a Partitioned Table

To create a partitioned table, you need to provide two additional parameters – the column that will 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 - 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

There is no direct command to add a new partition! Instead, partitions are automatically added on-the-fly by the JethroLoader as it loads new rows. Also, there is no option(or an 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 successfully ended, new queries will start seeing the new partitions and their data.

Listing Partitions

To list all the partitions of a table, use the **`SHOW TABLE PARTITIONS`** ***`table_name`*** command. The command lists the existing partitions, their range, row count and disk space. Note: 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
```

```
p7   | 2451706                | 2452070                |  548425830 | 25000.232   | 24208.061
p6   | 2452071                | 2452435                |  550208601 | 25083.587   | 24282.732
p5   | 2452436                | 2452800                |  412451934 | 18876.952   | 18544.290
-------------------------------------------------------------------------------------------
```

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

To drop a partition, use the ALTER TABLE command. To identify which partition to drop, specify any value within the partition range. JethroData will find in which partition does the value reside and drop that partition. For example:

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

Concurrency: Dropping a partition may take up to a few minutes, as it updates some 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 has successfully ended, new queries will stop seeing the dropped partition and its data.

4 Loading Data

The JethroData loader utility `JethroLoader` allows a fast and efficient data loading from text files into a JethroData table. It parses its input into rows and loads them into an existing JethroData table.

In production environments where large data loads are running during user activity, we recommend to run the `JethroLoader` from a separate host, to minimize the impact on running queries.

This chapter will use the following distinction between an input file and a table:

- The input file is made of **records** (usually, each line is a separate record), each record is broken into one or more **fields**.
- A JethroData table is made of **rows,** each row has one or more **columns**.

4.1 Running JethroLoader

The JethroLoader command line tool expect three parameters:

```
JethroLoader instance-name description-file
             STDIN | input-location [input-location]...
```

- *Instance Name* – the name of the local instance to connect.
- *Description File* – a file describing the format of the input file and the way it will be processed.
- *Input location*– either STDIN (to load from a pipe) or a list of one or one locations.
  Each location is a path (local or on HDFS) to a file or directory to be loaded.
  Files can be a mix of uncompressed files and compressed files (.gz or .gzip extension) – compressed files are automatically uncompressed in-memory when they are read.

Examples:

- `JethroLoader dev1 t1.desc t1_input_dir`
- `JethroLoader dev1 t1.desc t1.part1.csv t1.part2.csv`
- `JethroLoader dev1 t1.desc HDFS://home/jethro/files/t1.gz`
- `sed ... | JethroLoader dev1 t1.desc stdin`

NOTE – when loading from a pipe, the program that sends data to `JethroLoader` may fail in the middle. In that case, the loader is not be notified of error - it gets a standard end-of-file (EOF) notification for its input, and therefore will commit the data and exit without error.
It is your responsibility to monitor for such cases and respond accordingly.

The location of the loader report log is placed at:
`/var/log/jethro/{instance-name}/loader`
The loader report file name includes the target table name of each load.

4.2 Loader Description File Format

The loader description file describes the structure of the input file and the way it will be processed. It has three sections:

1. **Table-level section** – describe the format of the input file and some processing options.
2. **Column-level section** – a mapping between the fields in the file and columns in the table.
3. ***(Optional)* Record description section** – special clause for handling variable format files – files with different format per line (discussed in section Input Files With Variable Format)

To get started, you can copy and modify the sample description file that is located at `$JETHRO_HOME/doc/sample_loader_desc_file.desc`. The sample description file also include a summary of the syntax, commented out.

Table-Level Description Section

If using all the default options, the minimal table-level section is just the table name to be loaded into – like *`TABLE my_table`*. You can optionally specify non-default options like the file format and processing options.
The general syntax of the table level section is: *(the full version is at end of the chapter)*

```
TABLE [schema_name.]table_name
   [APPEND | OVERWRITE] [PARTITION partition_list]
   [ROW FORMAT DELIMITED
       [FIELDS [TERMINATED BY char]
               [QUOTED BY char | NONE]
               [ESCAPED BY char]
       ]
       [NULL DEFINED AS string]
   ]
   [OPTIONS [SKIP n] [ROW REJECT LIMIT n]]
```

APPEND / OVERWRITE and the optional PARTITION limit

The default behavior of `JethroLoader` is `APPEND` - inserting the input file data into the existing table. You can also optionally limit the load to just a specific set of partitions – the rest of the records will be skipped. Alternatively, you can specify `OVERWRITE` to replace either an entire table or just a specific set partitions. This allows updating or deleting rows from a table, in cases where dropping a partition is not fine-grained enough.

Overwriting (like appending) is an atomic operation – until the operation is done, queries will see the older version, and once it is done, the queries will see the new version. So, you can safely refresh the contents of a dimension table while users are running queries, for example, without the need to drop the dimension table, create it again and reload it. This avoids the risk of some queries failing (if the dimension table does not exist momentarily) or returning zero results (if joining to the dimension table while it is still empty, during `JethroLoader` run).

Overwriting specific partitions is useful for a data correction process – for example, replacing a daily partition with a corrected data. Limiting the loader to a specific set of partitions allows smooth replacement of specific partitions with new data. Input records outside the list of partitions will be automatically skipped. To limit the loader to a specific set of partitions, provide a comma-separated list of partition values, each quoted by single quotes, for example:

```
TABLE my_partitioned_table
   OVERWRITE PARTITION '2013-06-01', '2013-06-02', '2013-06-03'
...
```

`ROW FORMAT DELIMITED` – a clause describing a non-default file format.

- `FIELDS` sub-clause – specifies non-default field definition; how to break an input line into fields.
  - `TERMINATED BY char` – specifies an alternative field separator.
    The default field separator is tab (\t). For example:
    - TERMINATED BY '|'
    - TERMINATED BY '\001' – uses ASCII 001 (^A), used by Hive TEXTFILE by default.
  - `QUOTED BY char | NONE` – specifies a quote character.
    Default is `none` - surrounding quotes are considered part of a string.
  - `ESCAPED BY char` – specifies an escape character (usually the backslash \ character) for quoted strings (must also specify a quote character). Needs to be explicitly specified. Once specifies, it allows:
    - embedding the quote character and escape characters inside a string, for example \" \\ \'
    - embedding special characters - `\t` for tab, `\n` for newline, `\r` for carriage return.
- `NULL DEFINED AS string` – specifies an alternative NULL string, in addition to the default empty string. For example, Hive and Impala CLIs write NULLs to their output as the string "NULL"

`OPTIONS` – a clause describing non-default processing options.

- `SKIP n` – instruct the JethroLoader to skip *n* header lines (default is 0 – no skipping).
- `ROW REJECT LIMIT n` – allows up to *n* invalid records in the input file before aborting the load. The The default row rehect limit is 100. Rejected rows are stored in a file called `loader_rejects_date_time_pid.out` in the local directory, each with the reject reason.

Column-Level Description Section

Basic Mapping

The column description section maps the file fields to table columns. For each field in the file, it mentions the target column name, by input file position. For example:

```
TABLE my_table (c1,c9,c5)
```

In the above example, the first field of each input record will be mapped to c1 column, the second field will be mapped to c9 column, and the third field will be mapped to c5 column.

- If a specific record has additional fields, they will be ignored.
  In the above example – if a specific record has five fields, the last two will be ignored.
- If a specific record has fewer fields than mentioned in the description file, fields with missing value will have `NULL` value.
  In the above example , if a specific record has only one field, then columns c9,c5 will have NULL

- Any table columns that are not mentioned in the description will have `NULL` value.
  In the above example, if table MY_TABLE also has a column named c4, it will have NULLs.

Column NULL Definition

A specific per-column NULL identifier can be declared, adding NULL defined as 'string' following the specific column's name. This comes as addition to the global NULL definition that can be declared in the table-**level** description.

```
c1 null defined as 'c1-null'

c2

c3 null defined as 'c3-null'
```

*In the above example, where **c1-null** appears in c1 and **c3-null** appears in c3, NULLs will be written.*

Skipping Input File Fields

An input file may include fields that are you may wish not to load. You can ask the loader to skip them by specifying `skip_columns(n)` in the description file. The skipped fields are not mapped to any column. For example:

```
TABLE my_table
  OPTIONS SKIP 2
  (c1,
   skip_columns(3),
   c9)
```

In the above example, the first field of each input record will be mapped to `c1` column and the fifth field will be mapped to `c9` column – skipping three fields from each input record (fields 2,3,4).
In addition, the first two records (lines) in the input file will also be skipped (`OPTIONS SKIP 2`).

Assigning Constants, Internal Variables and Functions

Instead of assigning a column the corresponding field from the input file, you can instead assign it a constant, an internal loader variable or the output of a few function calls.

In that case, the corresponding field from the input file will be skipped.

NOTE - if you don't want to skip any the input file fields, put all the special column assignments at the end of the description file column list.

The **internal loader variables** can help you track the lineage of each row – where did it come from:

- `$LOAD_START` – when did this `JethroLoader` run start (returns TIMESTAMP).
- `$FILE_NAME` – the name of the current file that is being loaded (returns STRING).
- `$FILE_PATH` – the full path of the current file that is being loaded (returns STRING).
- `$LINE_NUMBER` – the line number in the current input file (returns BIGINT)

In addition, there are several **built-in functions** you can use:

- `getenv('env_variable_name')` – get an environment variable value
- `substr(another_column, start_pos[,length])` – returns a substring of another

column from character position `pos`, either until its end or only `length` characters.

- `extract(another_column,'timestamp_element')` – extracts a element from a timestamp column, like day of month. Valid elements are '`year`', '`quarter`', '`month`', '`week`', '`day`', '`hour`', '`minute`', '`second`' and '`microsecond`'.
- `url_extract_domain(another_column)` – extracts domain name from url
- `url_extract_base_url(another_column)` – extracts base url (remove parameters and anchor) from url.

For example:

```
TABLE my_table
( product_code,
  product_code_type = substr(product_code,1,4),
  load_start_ts     = $LOAD_START,
  load_full_path    = $FILE_PATH,
  load_line_number  = $LINE_NUMBER,
  source_region     = 'us-east-1',
  loader_home_dir   = getenv('JETHRO_HOME')
)
```

In the above example, only `product_code` column is read from the input file.
The rest are either constants (`load_start_ts`, `load_full_path`, `source_region`, `load_home_dir`) or computed per row (`product_code_type`, `load_line_number`).

INTEGER/BIGINT Formats

Numeric values loaded into INTEGER or BIGINT column can be formatted as a decimal number or as hexadecimal number using the ‘0x’ prefix. The format is automatically identified by the Loader and converted to INTEGER or BIGINT.

Examples for numeric input for loader and their converted INTEGER/BIGINT values:

Input Value	Value in INTEGER column	Value in BIGINT column
1234	1234	1234
-1234	-1234	-1234
0x4D2	1234	1234
0xFFFFFB2E	-1234	4294966062
0xFFFFFFFFFFFFFB2E	Out of range	-1234

TIMESTAMP Formats and Timezone

The default timestamp format is '`yyyy/mm/dd HH:mm:ss`' (without sub-second component). However, you can specify a FORMAT clause to specify an alternative format per column.

For example:

```
TABLE my_table
( event_id,
  event_ts        format='dd-MM-yyyy HH:mm:ss.SSS',
  processing_date format='yyyy-MM-dd')
```

If the input string of the field is longer than the format string, the rest of field content is ignored. For example: for `format='yyyy/MM/dd'`, the input field '`2014-02-14 15:16:17`' will be stored as the timestamp '`2014-02-14 00:00:00`'. This allows truncating the input record to a lower precision – in the above example, from a second-level to a day-level.

Timestamp format strings are **case sensitive**. The valid format elements are:

Format Element	**Meaning**
`yyyy`	4-digit year (1970-2038)
`M`	1 or 2 digit month (1-12)
`MM`	2-digit month (01-12)
`MMM`	3-character month (Jan-Dec)
`d`	1 or 2 digit day (1-31)
`dd`	2-digit day (01-31)
`H`	1 or 2 digit hour (0-23)
`HH`	2-digit hour (00-23)
`m`	1 or 2 digit minute (0-59)
`mm`	2-digit minute (00-59)
`s`	1 or 2 digit second (0-59)
`ss`	2-digit second (00-59)
`SSS...`	1-6 -digit sub-second element For example: S → 1 digit, SSS → 3 digits, SSSSSS → 6 digits etc.
`unix_timestamp`	The number of seconds and microseconds/milliseconds (optional) since 1/1/1970. The format cannot be mixed with other format elements.

In addition, the `JethroLoader` by default does not perform any **timezone** manipulation – the input is loaded as is. In other words, we store all TIMESTAMP data in UTC timezone and we assume all input is already in UTC.

However, in some cases you may want to adjust the timezone of an input field. For example, you may want to load log files from different data centers – each file has timestamps in the local timezone, so each needs its own adjustment.

You can ask for a timezone adjustment of a specific field by using the `TIMEZONE` keyword for timestamp field. In that case, the `JethroLoader` will compute the current offset of that timezone vs. UTC once (at the beginning of the run) and apply it to all values of that field. The offset is specified at the field-level – different fields may have different offsets.

For example:

```
...
event_ts timezone='America/New_York',
processed_ts format='unix_timestamp' timezone='America/Los_Angeles',
...
```

4.3 Input Files with Variable Format

In some cases, different records (lines) in the same file have different format. For example, a file can include several event types or log message formats, each with a different number and order of fields.

JethroLoader description file supports these type of input files in the following manner:

1. You can define several, different record description blocks – different mapping of the source record to table columns, You must provide each of them a different alias (record_desc_alias).
2. If you defined more than one record description block, you must also add a record description chooser block to the description file– a list of rules for choosing the correct record format per row.

The record description chooser syntax allows defining a list of rules on the source data. Each rule has a condition plus a record description alias. For example:

```
@if (int($1) == 2) type2
```

The rule above says – find the first field in each record, cast it to integer, check if that integer equals to two, and if so, process this record according to record description called type2 (see an example below).

The rules are processed one-by-one, by their order in the description file. Once a rule is matched, the record is processed according to the record description alias and the JethroLoader moves to the next record. If none of the rules match a specific record, it is skipped.

For example, assume an input file that holds multiple event types. The first field of each line holds the event type. In the example, we chose not to load the event type into a column, just use it for record format decision (though you could definitely also map it to a column in the record description blocks):

```
TABLE events
OPTIONS
  ROW REJECT LIMIT 10000
(skip_columns(1),event_id,event_time,url) type1
(skip_columns(1),event_id,event_time,user_id,url) type2
(skip_columns(1),event_id,event_time,host,path,filesize) type3
@if (int($1) == 1) type1
@if (int($1) == 2) type2
@if (int($1) >= 3) AND int($1) <= 7) type3
```

4.4 CSV Output Mode

Sometimes it useful to export a query result set as a CSV file. For example, you may want to store it or load it into another database (or even back to JethroData).

The CSV output mode (`-c` option) allows a simple export to CSV. It writes the query result set into a compact form (without wrapping each column with extra whitespace). It also suppresses all informational message (quiet mode) so there are no extra lines to remove in the beginning or end of the output, lets you control the column separator character (default is comma) and let you optionally remove the header line.

<u>JethroClient Command Line Options</u>

- `-i FILE` – executes a SQL script and exit. It runs all SQLs from a local FILE (each SQL should be terminated with a semicolon).
- `-o FILE` : writes the output to FILE.
- `-q TEXT` : executes the TEXT query and exit.
- `--no_header` : does not print the header lines for queries (the column names lines).
- `--quiet` : does not print informational messages.
- `-c` : switches to CSV output mode
- `-d 'char'` : for CSV mode, changes the column delimiter from comma to `char`. You can also use ASCII annotation like `'\001'` or use `'\t'` for tab delimiter.

4.5 Description File Syntax Diagram

Description File Format

```
table_level_desc
record_desc [record_desc]...
[record_desc_chooser]...
```

table level desc

```
TABLE [schema_name.]table_name
  [APPEND | OVERWRITE] [PARTITION partition_list]
  [ROW FORMAT DELIMITED
     [FIELDS [TERMINATED BY char]
             [QUOTED BY char | NONE]
             [ESCAPED BY char]
     ]
     [LINES TERMINATED BY string]
     [NULL DEFINED AS string]
  ]
  [OPTIONS [SKIP n] [ROW REJECT LIMIT n]
  ]
```

record desc

```
( column_desc [,column_desc]... ) [record_desc_alias]
```

column desc

```
column_name                                                       |
column_name [NULL DEFINED AS ‘string’]                            |
column_name [FORMAT='format_string'] [TIMEZONE='region/location'] |
column_name = const | builtin_var | builtin_function              |
skip_columns(n)
```

builtin var

```
$LOAD_START  |  $FILE_NAME  |  $FILE_PATH  |  $LINE_NUMBER
```

builtin function

```
extract(another_column,'timestamp_element')  |
getenv('env_variable_name')                  |
substr(another_column, start_pos [,length])
```

record desc chooser

```
@if(condition) record_desc_alias;
```

condition

```
simple_condition              |
(condition)                   |
condition AND | OR condition  |
NOT condition
```

simple condition

```
data_type($col_position)   = | != | <> | > | >= | < | <=   constant
```

4.6 Scheduled Loads

Jethro allows scheduling of automatic loads, according to pre-defined time cycles. Input files are polled periodically from a source folder called drop folder and loaded into table.

Scheduled loads are managed by loads scheduler service. The service will initiate periodic check for new files in the drop folder and will start a loader task if one or more new files found in the drop folder. Following a successful or failed load input files will be renamed, deleted or moved as defined in the create scheduled load command.

Starting Scheduled Loads Service

To enable scheduled loads start the Jethro loads scheduler service:

```
service jethro start <Instance-name> loadscheduler
```

To stop the service:

```
service jethro stop <Instance-name> loadscheduler
```

Alternatively this service can be set to start automatically upon service start command. Edit the service.ini file and set the third parameter for this instance to **yes**:

```
vi $JETHRO_HOME/instances/services.ini
--> Instance-name:port:yes:yes:yes
```

Scheduling a Load

To schedule a load, run CREATE SCHEDULED LOAD command from JethroClient.

Example:

Creating scheduled load that start every 15 minutes that loads file from HDFS drop folder /data/drop into table PRODUCTS:

```
CREATE SCHEDULED LOAD products FROM HDFS:/data/drop SCHEDULE PERIODIC
15 MINUTES;
```

For more information see: CREATE SCHEDULED LOAD

5 Administrating JethroData

5.1 Managing Local Cache

`JethroServer` automatically caches some of its HDFS files locally. Caching can accelerate read time for frequently accessed files and also avoid frequent accesses to the NameNode and DataNodes.

The local cache is populated by `JethroServer` process in the background. It is used by all JethroData processes, including `JethroLoader` and `JethroMaint`. The cache transparently handles partially written files – if a file is cached and later appended with new data, all processes will continue work correctly (internally, the older data will be read from the local cache and the newer parts will be read directly from HDFS).

Also, the local cache is automatically checked and updated once a minute. Irrelevant files are removed (after tables are dropped or indexes are optimized) and new data is incrementally added, up to the cache max size limit.

The local cache location and size is controlled by the parameters `local.cache.path` and `local.cache.max.size`. Both are initially derived from the command line parameters of `JethroAdmin create-instance` or `attach-instance`, which provides a directory that is used for both Local Cache and for Adaptive Cache. They can be modified later if needed by editing the `local-conf.ini` file at `$JETHRO_HOME/instances/`*`instance_name`*.

Full Caching of Specific Columns

In addition to the automatic caching, you can explicitly ask that specific columns or tables will be fully cached. For example, frequently accessed columns of large tables (like join keys or common measures) or entire small dimension table.

In the current release, you need to manually specify which columns or tables will be fully cached, using the configuration file `local-cache.ini` in `$JETHRO_HOME/instances/`*`instance_name`*.

Each line in the file adds a caching request for a column or a table in the format of `schema_name.table_name.column_name`. You can specify `*` instead of column name to ask all columns to be cached. For example:

```
def_schema.country_dim.*
def_schema.time_dim.*
def_schema.fact.country_id
def_schema.fact.time_id
```

Processing Order – the local cache is first populated with schema and column metadata. Then, the `local-cache.ini` file is processed line by line – so its order implies priority. In any case, local caching will not exceed the user specified max disk space (`local.cache.max.size`).

Refreshing the file – changes to `local-cache.ini` are automatically identified and applied by the `JethroServer` process during the next cache refresh (once a minute). There is no need for any user action, like restarting the process.

Errors – if there are any errors while parsing the file, they will be noted in the JethroServer log file at `$JETHRO_HOME/instances/`*`instance_name`*`/log/jethroserver.log`

Viewing the Local Cache

You can view the current contents of the local cache with the `SHOW LOCAL CACHE` command. It shows for each table – the disk space it uses in the local cache. In addition, under "Non-metadata cached (MB)" column you can find how much of the total comes from the specific `local-cache.ini` configuration.

```
SHOW LOCAL CACHE;
```

```
Schema Name          | Table Name            | Total Cached (MB) | Non-metadata cached (MB)
------------------------------------------------------------------------------------------
def_schema           | test1                 | 1,713.31          | 1,008.73
def_schema           | test2                 | 6.035             | 0
def_schema           | test3                 | 20,800.9          | 18,634
TOTAL                | TOTAL                 | 22,520.245        | 19,642.73
------------------------------------------------------------------------------------------
```

If you make changes to the local cache configuration, you can use the above command to monitor the progress of populating the cache and the size cached per table.

5.2 Managing Adaptive Cache

As users work with their favorite BI tools, the sequence of SQLs that these tools generate and send to JethroData has some predictable patterns. For example, most users start from a dashboard (that sends a predictable list of queries), than typically start adding filter conditions and aggregate conditions one at a time.

Adaptive cache is a unique JethroData feature that leverage those patterns. It is a cache of re-usable, intermediate bitmaps and query results that is automatically maintained and used by the `JethroServer` engine on its local storage. The cache is also **incremental** – after new rows are loaded, the next time a cached query will be executed it will in most cases automatically combine previously cached results with computation over only the newly loaded rows.

The adaptive cache is managed locally by each `JethroServer`. It is guaranteed to be consistent – having the same transactional visibility as a regular query – and will not return stale results. The adaptive cache is self-managed – when it fills up, it removes less useful entries while taking into account how frequently were they used, how recently were they used and how much cache space they consume.

Cache Size and Location

The adaptive cache location and size is controlled by the parameters `adaptive.cache.location` and `adaptive.cache.quota`. Both are initially derived from the command line parameters of `JethroAdmin create-instance` or `attach-instance`, which provides a directory that is used for both Local Cache and for Adaptive Cache. They can be modified later if needed by editing the `local-conf.ini` file at `$JETHRO_HOME/instances/`*`instance_name`*.

Adaptive Query Cache

Adaptive query cache is the part of the adaptive cache that holds **intermediate query result sets** (for SELECT statements only). It is enabled by default, but can optionally be turned off by setting `adaptive.cache.query.enable` to zero (this could be useful when performing static performance benchmarks).

A query is considered for inclusion in the adaptive query cache if it finished without errors and took more than `adaptive.cache.query.min.response.time`, which defaults to one second. Also, the cached result set must be lower than `adaptive.cache.query.max.results` rows, which defaults to 100,000 rows. When possible, the result set will be stored in incremental format, otherwise it will be stored in full format.

Adaptive Query Cache – Incremental

JethroData can incrementally refresh several common types of queries after new data is loaded – for example, aggregative queries on a star schema. In those cases, when a query result set was stored in the query cache and new data is loaded into one of the tables, the next time this query will be executed JethroData will automatically combine the previously cached result set with a computation only on the newly inserted rows.

A query must meet some criteria to be cached in incremental result set mode. The main ones are:

1. It may include unlimited joins and sub queries.

2. It should include an aggregation in its top-level query block.
3. It should not include DISTINCT inside aggregate functions, like `count(distinct column)`.
4. It should only include simple columns in its GROUP BY clause and inside the aggregate functions, not function calls or complex expressions.
5. The number of rows returned from the top-level GROUP BY, before the final HAVING clause or LIMIT clause, should be below the `adaptive.cache.query.max.results` limit.

Adaptive Query Cache – Full mode

When a query does not meet the criteria to be cached in incremental result set format, it is considered for being fully stored. Any query can have its result set fully stored if its number of rows is below the threshold, except queries that call non-deterministic functions, such as `rand()` or `now()`.

In this mode the final query result set is stored. So, repeated query executions will just read the cached result and send it to the client without additional processing. However, a full result set is not incremental – if any of the tables involved in the query change (new rows inserted, old partitions dropped etc), the cached result set will be discarded.

Adaptive Index Cache

Adaptive index cache is the part of the adaptive cache that holds **intermediate bitmap index entries** that were computed on-the-fly during query execution. It is enabled by default, but can optionally be turned off by setting `adaptive.cache.index.enable` to zero (this could be useful when performing static performance benchmarks).

An intermediate bitmap is considered for inclusion in the adaptive index cache if creating it took more than `adaptive.cache.index.min.response.time`, which defaults to 0.5 second. Also, its size must be lower than `adaptive.cache.index.max.size`, which defaults to 100MB.

The intermediate bitmaps are stored separately per partition. So, dropping a partition does not invalidate cached bitmaps from other partitions. In addition, intermediate bitmaps are currently cached only for IN conditions, either with a list of values or those generated as part of a star transformation optimization.

Viewing the Adaptive Cache

You can view all the queries and indexes that are currently cached with the `SHOW ADAPTIVE CACHE` command. The command shows various attributes of the cached elements, including the number of times they were used, the last time they were used and their rank - low rank means less valuable, more likely to be deleted when cache becomes full.

The command output is still work-in-process and will be cleaned up and fully documented in the coming releases.

5.3 Managing Query Queuing

Query queuing is a feature that limits the number of SQL queries that can run concurrently. By transparently limiting the number of concurrent queries, it reduces JethroData's peak memory utilization, which helps avoiding out-of-memory issues under heavy usage.

Query Queuing Behavior and Parameters

- **Concurrent Queries** - the maximum number of concurrently running queries is controlled by the parameter `query.resource.max.concurrent.queries`.
  The default value is 0, which means that JethroData automatically picks a value based on the host memory size and number of cores. Alternatively, you can set it to any positive number, to choose the number of concurrently running queries.

The exact formula is internal and may change from version to version.
As of version 0.9, it is set to allow one concurrent query per 20GB of host RAM or one concurrent query per four cores – the lower of the two.

- **Additional Memory Limit** – JethroData also adapts the number of concurrent queries based on the current memory utilization of the host, as an additional protection against memory spikes.
  If available host memory drops below a certain threshold, new queries will be queued, regardless of the number of queries currently running. The default threshold is 15% - controlled by the parameter `query.resource.min.memory.available.percentage`.
- **Query Queue Size** – the number of queries that can be queued at the same time is controlled by the parameter `query.resource.max.waiting.queries`. The default queue size is 20 queries. When a query needs to be queued and the queue is full, it will fail immediately.
- **Queue Timeout** – if a query is queued for execution for a long period of time, it will eventually time out and an error will be returned to the client. The timeout is controlled by the parameter `query.resource.max.waiting.timeout`, which defaults to 60 seconds.
  This parameter can also be set at the session level.
- **Adaptive Cache Bypass** – when a query is received from the client, the `JethroServer` checks if the results for the query are already available in the adaptive query cache. If so, the query will skip the queue and will return the cached results immediately.
- **Non-Query Statements** are never queued. They are typically not resource-intensive.

Viewing The Queries Queue

You can view all the queries that are in progress with the `SHOW ACTIVE QUERIES` command. The command shows all queries that are in progress (running or queued), including their position, duration, the client IP address and the query text.

For example:

```
SHOW ACTIVE QUERIES;
```

```
Host         | Position | Query ID        | Duration    | Status   | Client        | Query
----------------------------------------------------------------------------------------------------
jethro-dev1  |        1 | 6@jethro-dev1   | 20.5861981 | RUNNING | 62.0.78.21    | SELECT ...
jethro-dev1  |        2 | 7@jethro-dev1   | 18.5715730 | RUNNING | 62.0.78.195   | SELECT ...
jethro-dev1  |        3 | 8@jethro-dev1   | 18.5382791 | RUNNING | 62.0.74.2     | SELECT ...
```

```
jethro-dev1 |        4 | 9@jethro-dev1  | 14.5227342 | QUEUED  | 127.0.0.1   | SELECT ...
jethro-dev1 |        5 | 10@jethro-dev1 |  6.4988081 | QUEUED  | 62.0.74.203 | SELECT ...
jethro-dev1 |        6 | 14@jethro-dev1 |  0.0000522 | RUNNING | 62.0.78.62  | SHOW ACTIVE
QUERIES
-----------------------------------------------------------------------------------------------
```

Note – when using Client-Side Load Balancing, each SQL command will be directed to a different JethroData host. So, take it into account and either:

- *Connect to a single JethroServer to investigate it.*
- *Prepare to have each execution return from a different host (check the Host column).*
- *Use JethroClient - it has special handling for this command. It automatically iterates over all the JethroServers in its connect string and concatenates the results.*

5.4 Viewing Background Maintenance Status

The `JethroMaint` service is in charge of running maintenance tasks in the background. Its main responsibilities are:

1. **Optimizing indexes** – performing a background **merge** of column indexes after data loads. The merge writes a new version of an column index that replaces two or more column index files. New queries will automatically start using the new version of the column index.
2. **Deleting unneeded files** – after an index was optimized, the files of its old version can be deleted. However, there could be queries already in progress that access the older version of the index, so the older versions are deleted after a delay. Also, dropped and truncated tables are also physically deleted in the background after the same delay.

If the `JethroMaint` service was not running for a long time (for example, it was manually stopped or it ran into an unexpected issue), it could affect query performance, as each column may have unoptimized index with many small index files.

`SHOW TABLES MAINT` Command and Output

To verify that all maintenance operations ran successfully, use the **`SHOW TABLES MAINT`** command:

```
SHOW TABLES MAINT;
```

```
ID     | Schema Name | Table Name | Merge Status                 | Pending Delete (MB)
-----------------------------------------------------------------------------------------
  806 | def_schema   | test1      | FULLY MERGED (145/145)      |       311.1
 1208 | def_schema   | test2      | MOSTLY MERGED (140/28)      |      2177.4
 1295 | def_schema   | test3      | FULLY MERGED (32/32)        |         0.0
 1832 | def_schema   | test4      | FRAGMENTED (370/37)         |      5436.6
-----------------------------------------------------------------------------------------
```

The **`Merge Status`** column shows a status indication of the merge status:

- **FULLY MERGED** means that all the indexes of the table are fully optimized and there is no additional optimization work pending.
- **MOSTLY MERGED** means that there is only a minor amount of work of index optimization pending. Query performance should not be significantly affected, though if you run a benchmark, you should probably wait a few minutes to let `JethroMaint` finish the background optimization.
- **FRAGMENTED** means that there are many index files per column. That may negatively impact query performance. This should not happen under normal circumstances and typically indicates that `JethroMaint` was not running for an extended period (see troubleshooting below).

The Merge Status also includes a more technical status - the actual number of index files and optimal number of index files (*actual / optimal*). In most cases, the optimal number of index files are one per column (and for partitioned table, one per column per partition), though if a table (or a partition) is very large (many billions of rows), the optimal number could be bigger.

The **`Pending Delete`** column shows how many MBs will be freed in HDFS when old index files will be deleted by JethroMaint.

Troubleshooting JethroMaint issues

1. Verify that JethroMaint service is running:

```
service jethro status
```

2. If the service is not running, start it by running:
   ```
   service jethro restart
   service jethro status
   ```
   You can also run it only for specific single instance:
   ```
   service jethro start instance-name maint
   ```

If the service does not start, check its log at:
`$JETHRO_HOME/instance/`*`instance_name`*`/log/services.log`

3. Check the log for errors. The `JethroMaint` log is located at
   `$JETHRO_HOME/instance/`*`instance_name`*`/log/jethro_maint.log`

5.5 Changing The Password of `jethro` User

Each JethroData instance has a pre-defined database user called `jethro`, with a default password of `jethro`. To change the default password, run the following command from any JethroData host:

```
JethroAdmin change-pass instance_name
```

You will be asked to provide a new password for the `jethro` user.

The current release uses this user/password combination to control access to the instance. A more comprehensive authentication and authorization functionality will be available in a future release.

5.6 Local Directory Structure

JethroData software is installed under `/opt/jethro`.

The directory structure supports multiple installed binary versions – for example, for easier upgrade / downgrade. It also supports multiple instances – for example, running several dev and test instances from the same host.

`/opt/jethro` – the top JethroData directory
`/opt/jethro/jethro-`*`version`* – one directory per installed version of JethroData
`/opt/jethro/jethro-`*`version`*`/bin` – binaries and scripts
`/opt/jethro/jethro-`*`version`*`/conf` – templates for new instance configuration
`/opt/jethro/jethro-`*`version`*`/doc` – documentation and samples
`/opt/jethro/jethro-`*`version`*`/instances` – symbolic link to instances directory
`/opt/jethro/jethro-`*`version`*`/jdbc` – JDBC driver
`/opt/jethro/jethro-`*`version`*`/lib` – shared libraries
`/opt/jethro/current` – a symbolic link to the current used version of JethroData
`/opt/jethro/instances` – metadata for all instances
`/opt/jethro/instances/services.ini` – a config file controlling autostart of services
`/opt/jethro/instances/`*`instance_name`* – metadata of a specific instance
`/opt/jethro/instances/`*`instance_name`*`/local-conf.ini` –
a config file with the instance HDFS location, local cache attributes and overriding parameters
`/opt/jethro/instances/`*`instance_name`*`/local-cache.ini` –
a config file for additional control of local caching (see section Managing Local Cache)
`/opt/jethro/instances/`*`instance_name`*`/jethrolog.properties`–
a config file controlling instance debug logging (used by Jethro support)
`/opt/jethro/instances/`*`instance_name`*`/log` – symbolic link to the instance log dir
`/var/log/jethro` – top directory for JethroData logs
`/var/log/jethro/`*`instance_name`* – log directory for a specific instance

6 Auto Micro Cubes Generation

Jethro auto micro cubes brings the great performance value of pre-aggregated cubes, combined with the ability to automatically generate and transparently maintain those cube.
Cubes can be useful to accelerate performance for certain set of queries, mainly queries that are not highly filter and would normally scan large parts of the data set. Jethro optimizer combines cubes with indexing and caching technology to ensure interactive queries response time on wide range of use cases.

Cubes can be generated in two modes:

1. Auto generation based on queries sent to the server
2. Manual cube generation for a given query using the GENERATE CUBES command

Auto cubes generation process is transparent to user. Jethro intercept incoming and automatically generate the appropriate cubes. The initial cubes generation phase for BI dashboard is called **training phase**. Training phase for a new dashboard usually take anywhere between few minutes to an hour.

6.1 Requirements and Initial Setup

Since cube generation can be resource consuming process it is recommended to allocate a designated Jethro node for cubes generation. If a designated node can't be allocated it is recommended turn on cube auto generation only when business user are not accessing the system.

Jethro maint service is responsible for intercepting incoming queries and sending cube generation request to the designated cube generation execution server. By default cube generation host is *localhost* and the default port is *9111*. To change server address or port set the parameter **dynamic.aggregation.auto.generate.execution.hosts**. For example, if cubes generation node dns is cubes.gen and the JethroServer running on port 9112 run the following set global command from SQL client connect to the instance:

```
set global
dynamic.aggregation.auto.generate.execution.hosts=cubes.gen:9112;
```

6.2 Turn on Automatic Cubes Generation

By default auto cubes generation is turned off. The turn cube generation state (on/off) is controlled by the parameter **dynamic.aggregation.auto.generate.enable.** Use set global to enable/disable auto cubes generation.

To enable auto cubes generation set the value to 1:

```
set global dynamic.aggregation.auto.generate.enable=1;
```

To disable auto cubes generation set the value to 0:

```
set global dynamic.aggregation.auto.generate.enable=0;
```

6.3 Manual Cubes Generation

Cubes can be manually create from a query via the GENERATE CUBES command. The command will generate one a more cubes from a given query.

The generate cube command allow also generation of cubes with WHERE. For more info see: GENERATE CUBES.

6.4 Cubes Management

Related commands:

Show all generated cubes use SHOW CUBES;
To remove all cubes from cube repository: DROP CUBES;

Cubes location

Cubes are persistently stored at the instance storage location (either HDFS, NFS or local file system). When total cubes size executed max storage size allocated for cube unused cubes are automatically dropped.

Cubes update

When new data is loaded cubes are automatically updated in the background. The maint service is responsible for updating cubes by sending UPDATE CUBE command to the cube generation server.

Cubes can also be updated manually by issues UPDATE CUBE BY KEY command. The key can be found in the SHOW CUBES command output.

7 SQL Language Elements

7.1 Data Types

The supported column data types in JethroData are:

- **INTEGER** – 32-bit integers
- **BIGINT** – 64-bit integers
- **FLOAT** – 32-bit floating point values.
- **DOUBLE** – 64-bit floating point values.

  When specifying any numeric literal (constant) in SQL, you can choose between standard and scientific notation. For example, either `0.011` or `11e-3`
- **STRING** – store string values of up to 4KB. Replaces char() and varchar() data types in other databases.

  To specify a string literal in SQL, use a single quote. For example, `'abc'.`

  NOTE: string processing assumes data is in ASCII format.
  While Unicode data can be stored inside string columns, no special Unicde handling is currently implemented.
- **TIMESTAMP** – a timestamp data type includes both a date part (year+month+day) and a time part (hour+minute+second +optional sub-second, up to 6 digits).

  Valid values are between '1970-01-01 00:00:00' and '2038-01-19 03:14:07'.

  To specify a timestamp literal (constant) in SQL, there is implicit casting from three string formats to timestamp:

  ```
  'yyyy-MM-dd'
  'yyyy-MM-dd HH:mm:ss'
  'yyyy-MM-dd HH:mm:ss.SSS...' (1 to 6 digits)
  ```

  for example: `'2014-02-25 13:14:15.250'`

To convert between data types, you can use the explicit casting operator `::` or the cast() function.

7.2 Valid Identifiers

In JethroData, you can use **identifiers** as schema names, table names, view names, column names, table and column aliases (in SELECT statement) etc.

A valid identifier:

- Is made from a sequence of alphanumeric characters (English letters and digits) and underscore character.
- Starts with a letter.
- Has maximum length of 64 characters,
- Is case-insensitive (and stored internally in lower case).

7.3 Operators

Numeric Operators

+	addition
-	subtraction
*	multiplication
/	division

Timestamp Operators

ts + *i*	adds *i* seconds to *ts*
ts - *i*	subtracts *i* seconds from *ts*
ts - *ts*	computes difference in seconds, treated as unix_timestamp, converted to timestamp

Relational Operators (returns TRUE/FALSE/NULL)

=	equal
!= *or* <>	not equal
>	greater than
>=	greater or equal to
<	less than
<=	less than or equal to

Other Operators

::	Explicit cast *expression::data_type*

7.4 Aggregate Functions

COUNT()

Returns the number of rows that matches a specified criteria.

count(*) - Returns the number of records in a table

count(*expression*) – Returns number of rows where *expression* is not NULL

count(**distinct** *expression*) – Returns number of distinct values of *expression*

AVG()

avg(*expression*) - Average (arithmetic) of *expression*

MIN()

min(*expression*) - Minimum value for *expression*

MAX()

max(*expression*) - Maximum value for *expression*

SUM()

sum(*expression*) - Sum of the values of *expression*

STDDEV_POP()

stddev_pop(*expression*) - Population standard deviation

STDDEV_SAMP()

stddev_samp(*expression*) - Unbiased sample standard deviation

VAR_POP()

var_pop(*expression*) - Population variance

VAR_SAMP()

var_samp(*expression*) - Unbiased sample variance

7.5 String Functions

ASCII()

ascii(*s*) - Returns ASCII code of the first byte of the argument *s*.

Example: ascii('x') → 120

CHAR()

char(*i*) - Converts an **int** ASCII code to a character.

Example: *char(65)* → *'A'*

CONCAT()

concat(*s1,s2...*) – Concatenate up to 5 strings.

Example: *concat('a','bc','d')* → *'abcd'*

INSTR()

instr(*s1,s2*) – Locate the first position of *s2* inside *s1*.

Example: *instr('abcd','cd')* → *3*

LENGTH()

length(*s*) – Returns the length of string *s*

Example: *length('abcd')* → *4*

LOCATE()

locate(*s1,s2*) - Locate the first position of *s1* inside *s2*.
Example: *locate('b','abcabc')* → *2*
locate(*s1,s2, pos*) - Locate the first position of s1 inside s2 from position pos.
Example: *locate('b','abcabc',4)* → *5*

LOWER()

lower(*s*) - Converts *s* to lowercase.
Example: *lower('AbCd')* → *'abcd'*

REPLACE()

replace(*s,sp, sr*) - Replace occurrences of string pattern *sp* in string *s* with replacement string *sr*. String patten *sp* can't be empty string.
Example: *replace('aCBef', 'CB', 'bcd')* → *'abcdef'*

RIGHT()

right(*s,n*) - Returns the right part of a string *s* with the specified number of characters *n*. When *n* is negative, it will return all string *s* but first |*n*| characters.

Example: *right('abcde', 2)* → *'de'*
right('abcde', -2) → *'cde'*

SPACE()

space*(a)* - Returns a string that is composed of the specified number of repeated spaces.

Example: *space(1)* → *' '*

SUBSTR()/SUBSTRING()

substr(*s*,*start)* - Remove portion of the string *s* from position *start*
Example: *substr('abcd',2)* → *'bcd'*
substr(*s*,*start*,*length*)
Example: *substr('abcd',2,1)* → *'b'*
The function **substring** is alias for **substr**.

TRIM()/LTRIM()/RTRIM()

trim(*s*) – Remove left and right whitespaces from string *s*.
Example: *trim ('   a   ')* → *'a'*
ltrim(*s*) – Remove left whitespaces from string *s*.
Example: *ltrim ('   a   ')* → *'a   '*
rtrim(*s*) – Remove right whitespaces from string *s*.
Example: *rtrim (' a ')* → *' a'*

UPPER()

upper(*s*) – Converts *s* to uppercase.
Example: *upper('AbCd')* → *'ABCD'*

7.6 Timestamp Functions

Current date/time

NOW()

now() – Returns current server date and time as timestamp.

CURDATE()

curdate() – Returns current server date as timestamp.

Date/time parts

YEAR()

year*(ts)* – Extracts year-part from *ts*.
Example: *year('2014-02-25 13:14:15')* → *2014*

QUARTER()

quarter*(ts)* – Extracts quarter-part from *ts*.
Example: *quarter('2014-02-25 13:14:15')* → *1*

MONTH()

month*(ts)* – Extracts month-part from *ts*.
Example: *month('2014-02-25 13:14:15')* → *2*

WEEKOFYEAR()

weekofyear*(ts)* – Extracts week number from the beginning of year of *ts*.
Example: *weekofyear('2014-02-25 13:14:15')* → *9*

DAY() / DAYOFMONTH()

day*(ts)* – Extracts day-part from *ts*.
Example: *day('2014-02-25 13:14:15')* → *25*
The function **dayofmonth** is alias for **day**.

DAYOFWEEK()

dayofweek*(ts)* – Extracts day of week of *ts* as number.
Example: *dayofweek('2014-02-25 13:14:15')* → *3*

HOUR()

hour*(ts)* – Extracts hour-part from *ts* as number.
Example: *hour('2014-02-25 13:14:15')* → *13*

MINUTE()

minute*(ts)* – Extracts minutes-part from *ts* as number.
Example: *minute('2014-02-25 13:14:15') → 14*

SECOND()

second*(ts)* – Extracts seconds-part from *ts* as number.
Example: *second('2014-02-25 13:14:15') → 15*

MICROSECONDS()

microseconds*(ts)* – Extracts microsecond-part from *ts* as number.
Example: *microsecond('2014-02-25 13:14:15.250') → 250000*

Date/time Manipulation and Conversion

DATEDIFF()

datediff*(ts1, ts2)* – Return the difference in days (*ts1*-*ts2*).
Example: *datediff('2014-02-25','2014-02-22') → 3*

DATE_ADD()

date_add*(ts, i)* – Adds *i* days to *ts*. Returns timestamp.
Example: *date_add('2014-02-25 13:14:15', 2) → '2014-02-27 13:14:15'*

DATE_SUB()

date_sub*(ts, i)* – Subtracts *i* days to *ts*. Returns timestamp.
Example: *date_sub('2014-02-25 13:14:15', 2) → '2014-02-23 13:14:15'*

TIMESTAMPADD()

timestampadd*(unit, i, ts)* – Add interval *i* to *ts*. Returns timestamp. The unit for interval is given by the *unit*. Interval units are: *year / quarter / month / week / day / hour / minute / second / microsecond / frac_second.* Unit may include SQL_TSI_ prefix.
Examples:
timestampadd(MINUTE,1, '2010-01-02') → '2010-01-02 00:01:00'
timestampadd(SQL_TSI_YEAR,1,'2010-10-02') → '2011-10-02'

TIMESTAMPDIFF()

timestampdiff*(unit,ts1, ts2)* – Calculate *ts1-ts2*. Return integer. The unit for the difference interval is given by *unit*. Interval units are: *year / quarter / month / week / day / hour / minute / second / microsecond / frac_second.* Unit may include SQL_TSI_ prefix.
Examples:
timestampdiff(MONTH, '2010-02-01', '2010-04-01') → 2

TRUNC()

trunc*(ts, se)* – Truncate *ts* to date/time element level *se*. Return result as timestamp. Date/time elements: *year / quarter / month / week / day / hour / minute / second*
Example: *trunc('2014-02-25 13:14:15', 'month')* → *'2014-02-01 00:00:00'*

TO_TIMESTAMP()

to_timestamp*(s)* – Explicitly converts string *s* to timestamp.
Converted string s format is: 'yyyy-MM-dd HH:mm:ss.SSS...'
Example: *to_timestamp('2014-02-25 13:14:15', 'month')* → *{ts}*

UNIX_TIMESTAMP()

unix_timestamp*()* – Returns current server time as number of seconds since the epoch: 1970/1/1.
unix_timestamp*(ts)* – Convert timestamp *ts* to integer values representing the number of seconds since the epoch: 1970/1/1.
Example: *unix_timestamp('2014-02-25 13:14:15', 'month')* → *1393334055*

7.7 Math Functions

ACOS()

acos(*r*) - Returns invited cosinus of *a*

Example: a*cos(0)* → *1.5707963*

ASIN()

asin(*a*) – Returns invested sinus of *a*

Example: a*sin(1)* → *1.5707963*

ATAN()

atan(*a*) - Returns inverse tangent of *a*

Example: a*tan(tan(1))* → *1*

ATAN2()

atan2(*a,b*) - Returns inverse tangent of *a/b*

Example: a*tan2(4,-3)* → *2.2143*

CEIL()

ceil(*n*) - Returns the smallest BIGINT that is greater than or equal to *n*.

Example: *ceil(1.7)* → *2*

COS()

cos(*r*) - Returns cosinus of radians *r*

Example: *cos(pi()/2)* → *0*

DEGREES()

degrees(*r*) - Returns degrees of radians *r* (convert radians to degrees)

Example: *degrees(pi()/2)* → *90*

EXP()

exp(*a*) - Returns exponential of argument *a*

Example: *exp(1)* → *2.7182818*

FLOOR()

floor(*n*) - Returns the largest BIGINT that is less than or equal to *n*.

Example: *floor(1.7)* → *1*

LN()

ln(*a*) - Returns the natural logarithm of number *a*. Natural logarithms are based on the constant e

Example: *ln(95)* → *4.5538769*

LOG()

log*(a)* - Returns the natural logarithm of number *a*. Natural logarithms are based on the constant e

Example: *log(95)* → *4.5538769*

LOG10()

Log10*(a)* - Returns the natural logarithm of number *a* based on the constant 10.

Example: *log10(100)* → *2*

RADIANS()

radians*(d)* - Returns radians of degrees *d* (convert degrees to radians)

Example: *radians(90)* → *1.5705963*

ROUND()

round*(n,d)* – Rounds decimal number *n* to the number of decimals *d* specified. If *d* is omitted, the function will round the number to 0 decimal places.

Example: *round(125.315)* → *125*

round(125.315, 2) → *125.32*

round(125.315, -2) → *100*

PI()

pi*()* - Returns value of PI

Example: *pi()* → *3.1415927*

POW()

pow*(a,b)* - Returns *a* raised to the power of *b* (a^b)

Example: *pow(2,3)* → *8*

SIGN()

sign*(a)* - Returns the sign of number *a*. The possible return values are -1 if the number is negative, 0 if the number is zero, or 1 if the number is positive.

Example: *sign(-2)* → *-1*

SIN()

sin*(r)* - Returns sinus of radians *r*

Example: *sin(pi()/2)* → *1*

SQRT()

sqrt*(a)* - Returns square root of *a*

Example: *sqrt(4)* → *2*

<u>TAN()</u>

tan*(r)* - Returns tangent of radians *r*

Example: *tan(pi()/4)* → *1*

7.8 Control Flow Functions

CASE

The CASE function implements a complex conditional construct.
There are 2 variants for the CASE syntax:

Simple case:

```
CASE value
      WHEN [compare_value] THEN result
      [WHEN [compare_value] THEN result]  ...
      [ELSE result]
END
```

Searched case:

```
CASE
      WHEN  [condition] THEN  result
      [WHEN  [condition] THEN  result]...
      [ELSE result]
END
```

The first version returns the result where `value=compare_value`. The second version returns the result for the first condition that is true. If there was no matching result value, the result after `ELSE` is returned, or `NULL` if there is no `ELSE` part.

IF

if(*expr1,expr2,expr3)* – If *expr1* is `TRUE` (*expr1*<> 0 and *expr1* <> `NULL`) then return *expr2*; otherwise it returns *expr3*.
This function returns a numeric or string value, depending on the context in which it is used.

IFNULL

ifnull(*expr1,expr2)*

If *expr1* is not `NULL` returns *expr1*; otherwise it returns *expr2*.
The function returns a numeric or string value, depending on the context in which it is used

NULLIF

nullif(*expr1,expr2)*

Returns `NULL` if *expr1= expr2* is true, otherwise returns *expr1*.

7.9 Best Practices

- Always store data in the appropriate data type. For example:
  - Don't store numbers as strings.
  - Don't store timestamps as strings or numbers.
- In a WHERE clause, prefer using "`column operation expression`" – applying a function or a computation on the right side expression, not on the column.
  It allows JethroServer to use optimized execution. For example:
  - Instead of `WHERE a*2 > 10` use `WHERE a > 5` or `WHERE a > 10/2`
  - Instead of:
    ```
    WHERE datediff(now(),ts) >= 4
    ```
    move the function calls to the right side:
    ```
    WHERE ts >= datesub(now(),4)
    ```
    or a literal instead:
    ```
    WHERE ts >= '2014-03-15 12:00:00'
    ```
  - Instead of using multiple function calls on a timestamp column:
    ```
    WHERE year(ts)=2014 and month(ts)=2 and day(ts)=25 and hour(ts)=16
    ```
    use a simple range scan:
    ```
    WHERE ts >= '2014-02-25 16:00:00' and ts < '2014-02-25 17:00:00'
    ```

8 SQL Language Reference

8.1 ALTER TABLE

```
ALTER TABLE [schema_name.]table_name
  RENAME TO table_name                   |
  ADD COLUMN column_name data_type       |
  ADD PRIMARY KEY (column_name)          |
  DROP COLUMN column_name                |
  DROP PRIMARY KEY (column_name)         |
  DROP PARTITION FOR [(value[,value]…)] | [BETWEEN value AND value]
```

DESCRIPTION

Changes an existing table.

- **ADD / DROP COLUMN** – adds or drops a column.
  When adding a column, all existing rows will have NULL value for the new column.
- **ADD / DROP PRIMARY KEY** – adds or drops a primary key column.
  A table can have one primary key column. Primary keys must contain unique values and cannot contain NULL values.
- **DROP PARTITION** - to drop partitions, specify one or more column value for the partition key or a range of value using BETWEEN clause. The partition in which the value exists/that are within the given range will be dropped.
  Examples:
  ```
  ALTER TABLE web_events DROP PARTITION FOR ('2014-05-27');
  ALTER TABLE web_events DROP PARTITION FOR (BETWEEN '2014-01-01'
  AND ‘2014-03-01’);
  ```
- **RENAME** – renames an existing table.

8.2 CREATE JOIN INDEX

```
CREATE JOIN INDEX {join-index-name}
   ON {fact-table-name}({dim-table-name}.{dim-column-name})
   FROM {join-clause}

{join-clause} :
   {fact-table_name} JOIN {dim-table-name} on {fact-join-key-column}
= {dim-join-key-column}
```

DESCRIPTION

Creates a new join index for a given dimension column and a fact table.

Join index is an index on a one table, based on the values of another table column and on a specific join criteria. Typically it is an index on a large fact table based on the values of a dimension attribute. Join index accelerates queries by eliminating both the fetch of the join key from the fact table and the join implementation (hash join or IN - merging indexes).

Join indexes are relevant we have relatively large dimension (few K values or more) and the attribute (the column in the dimension) is low cardinality so that each value in the attribute represent large number of join key values.

Typically a join index is define if the average ratio between unique dimension attribute and the related join keys value is 1000 or more, but if fact table is large (more than few billion) it is recommended to define join index for attribute with smaller number of related join keys per value.

Example:

```
CREATE JOIN INDEX store_sales_by_item_color_idx
   ON store_sales(item.item_color)
   FROM store_sales JOIN item ON ss_item_sk = i_item_sk;
```

8.3 CREATE RANGE INDEX

```
CREATE RANGE INDEX {range-index-name}
   ON [schema-name.]{table-name}.{dim-column-name})
```

DESCRIPTION

Creates a new range index for a given column. The range index is used to optimize performance for range based queries.

Best practice: use when queries where clause contains wide ranges (ranges that contain hundreds or more values)

Limitations: currently range index can be created only for INT and BIGINT columns

8.4 CREATE SCHEDULED LOAD

```
CREATE SCHEDULED LOAD {table-identifier}
FROM {drop-folder}
SCHEDULE [CONTINUES] | [PERIODIC x MINUTES|HOURS]
DESCFILE {desc-file-name}
[optional-parameter [argument] [,...] ]

{drop-folder}: [HDFS://[{ip}]]drop-folder-path[/file-pattern]

{desc-file-name}: [HDFS://[{ip}]]description-file-full-path
```

DESCRIPTION

Create a scheduled load for a specified table from specified drop folder. Loader scheduler will initiate a check for new files in the drop folder according to defined schedule and will start a loader task if one or more new files found in the drop folder. Following a successful or failed load input files will be renamed, deleted or moved as defined in the create scheduled load command.

Mandatory parameters

- **Table Identifier**
  The name of the target table.

- **Drop Folder**
  Location of input file to load with optional files pattern. Drop can be a folder on locally mounted file system or on HDFS. The drop folder path can be defined using four alternative paths:

  Local path
  Full path of a folder on a locally mounted file system.
  The following example create scheduled load of all .csv files under local folder */home/jethro/salesdata/* into sales table:

  ```
  CREATE SCHEDULED LOAD sales FROM /home/salesdata/*.csv SCHEDULE PERIODIC 1
  HOURS;
  ```

  HDFS full path
  Full path of HDFS folder.
  The following example create scheduled load for all HDFS files under */data/sales* folder. HDFS name node IP taken from local core-site.xml file:

  ```
  CREATE SCHEDULED LOAD sales FROM hdfs:/data/sales/ SCHEDULE PERIODIC 15 MINUTES;
  ```

The following example create scheduled load for all HDFS files under */data/sales* folder using standard HDFS URI:

```
CREATE SCHEDULED LOAD sales FROM hdfs://my.cluster:8020/data/sales/ SCHEDULE
PERIODIC 15 MINUTES;
```

- **Schedule**
  The load schedule.
  `CONTINUES` - continually check if new files are available in drop folder and start load at least one new file found in drop folder
  `PERIODIC x MINUTES|HOURS` – check for new files in drop folder every X minutes/hours. Start load if at last one new files found in the drop folder.

- **Description file**
  Path to load direction file.
  DESCFILE {file-name} - full path to load description file. If file is located on HDFS use *HDFS:* prefix notation.

<u>Optional Parameters:</u>

- **Pre load file action**
  `PRELOAD_FILE_ACTION RENAME|NONE`
  `RENAME` (default) - Rename file before load to *.loading
  `NONE` - Don't rename file before load

- **Post load file action**
  `POSTLOAD_FILE_ACTION RENAME|DELETE|MOVE`
  `RENAME` (default) - Rename to *.done
  `DELETE` - Delete loaded file
  `MOVE [HDFS://[{ip}]]loader-files-folder` - Move file to a loaded file folder

- **Failed file action**
  `FAILED_FILE_ACTION RENAME|DELETE|MOVE`
  `RENAME` (default) - Rename to *.failed
  `DELETE` - Delete failed file
  `MOVE [HDFS://[{ip}]]failed-files-folder` - Move failed file to a failed file folder

<u>Comments and limitations:</u>

- All paths must not include spaces
- Move folders must be of same path type as the drop folders
- Scheduled load service must be started to allow scheduled loads to run. To start scheduled load service run:
  `service jethro start <Instance-name> loadscheduler`

Related commands:

`SHOW SCHEDULED LOADS` – Print all scheduled loads

`DROP SCHEDULED LOAD <table-name>` – Remove scheduled load of specific table

`DROP SCHEMA SCHEDULED LOADS <schema-name>` – Remove scheduled loads of specific schema (or default schema if parameter is empty).

8.5 CREATE TABLE

```
CREATE TABLE [schema_name.]table_name
(
  column_name data_type [PRIMARY KEY]
  [,...]
)
[ PARTITION BY RANGE(column_name) EVERY (part_interval) ]
```

Data type

```
INT | BIGINT | FLOAT | DOUBLE | STRING | TIMESTAMP
```

part interval

```
n      |
VALUE |
INTERVAL 'n' month | day | hour
```

DESCRIPTION

Creates a new table.

Table and column names must be a valid identifier. JethroData data types attributes can be found here. A table can have one primary key column. Primary keys must contain unique values and cannot contain NULL values.

The optional **PARTITION BY** clause defines range partitioning.

NOTE – unlike Hive and Impala, in JethroData the partition column name must be one of the columns previously defined in the table column list.

Partitioning interval specification depends on the data type of the partitioning column:

- For numeric partition key – use EVERY (n).
  For example:
  `PARTITION BY RANGE(store_sk) EVERY (10)`
- For string partition key – you must specify EVERY (VALUE). Each distinct string will have its own partition.
  For example:
  `PARTITION BY RANGE(store_name) EVERY (VALUE)`
- For timestamp partition key – use EVERY (INTERVAL …)
  For example:
  `PARTITION BY RANGE(sale_date) EVERY (INTERVAL '3' MONTH)`

See also the examples section of the range partitioning documentation.

8.6 CREATE VIEW

```
CREATE VIEW [schema_name.]view_name
   [(column_name[,column_name]...)]
   AS query
```

DESCRIPTION

Creates a new view.

The list of column names for the view can be explicitly stated before the view's query, otherwise it is derived from the query.

8.7 DROP ADAPTIVE CACHE

```
DROP ADAPTIVE CACHE;
```

DESCRIPTION

Drops all adaptive cache results. Once dropped adaptive cached results cannot be recovered.

8.8 DROP CUBES

```
DROP CUBES;
```

DESCRIPTION

Drops all existing cubes. Once dropped cubes cannot be recovered and must be recreated.

8.9 DROP JOIN INDEX

```
DROP JOIN INDEX join-index-name;
```

DESCRIPTION

Drops a join index.

Internally, the command only marks the join index for deletion. The dropped join index will be physically deleted later by the background maintenance service. New queries will not see the dropped join index, but in-flight queries will finish executing successfully, as if they started execution right before the join index was dropped.

8.10 DROP RANGE INDEX

```
DROP RANGE INDEX range-index-name;
```

DESCRIPTION

Drops a range index.

Internally, the command only marks the range index for deletion. The dropped range index will be physically deleted later by the background maintenance service. New queries will not see the dropped range index, but in-flight queries will finish executing successfully, as if they started execution right before the range index was dropped.

8.11 DROP SCHEDULED LOAD

```
DROP SCHEDULED LOAD [schema_name.]table_name;

DROP SCHEMA SCHEDULED LOADS [schema_name];
```

DESCRIPTION

Remove scheduled load of specific table

OR

Remove scheduled loads of specific schema (or default schema if parameter is empty).

8.12 DROP TABLE

```
DROP TABLE [schema_name.]table_name;
```

DESCRIPTION

Drops a table.

Internally, the command only marks a table for deletion. The dropped table will be physically deleted later by the background maintenance service. New queries will not see the dropped table, but in-flight queries will finish executing successfully, as if they finished execution right before the table was dropped.

8.13 DROP VIEW

```
DROP VIEW [schema_name.]view_name;
```

DESCRIPTION

Drops a view.

8.14 GENERATE CUBES

```
GENERATE CUBE {cubes-tag}
    [WITH WHERE where-expression]
    FROM select-statement
```

DESCRIPTION

Generate one or more cubes from a query. Cube tag and query (select statement) must be provided.

Cubes with Where

To create cube with where use the WITH WHERE clause. Cubes with where are generated on filtered data based on the given where expression and can be applied only to queries that include identical where expression in their where clause.

Generate cube example:

```
GENERATE CUBES mycube
FROM
SELECT PRODUCT, SUM(PRICE)
FROM T_PRODUCT
WHERE COUNTRY='US'
GROUP BY PRODUCT;
```

```
cube tag | status                    | rows | candidate query
----------------------------------------------------------------------
cube1    | finished successfully |    13 | SELECT
                                         | sales_demo.store_country,
                                         | sales_demo.store_name,
                                         | sum(sales_demo.net_profit)
                                         | FROM
                                         | sales_demo
                                         | GROUP BY
                                         | sales_demo.store_country,
                                         | sales_demo.store_name
----------------------------------------------------------------------
```

8.15 KILL QUERY

```
KILL QUERY {query-id};
```

DESCRIPTION

Kill a query by query id.

The get active query id run SHOW ACTIVE QUERIES.

8.16 SELECT

query block
```
SELECT [DISTINCT] select_item [,select_item]...
FROM table_element [join element]...
[WHERE condition]
[GROUP BY expression [,expression]... [HAVING condition] ]
[ORDER BY orderby_item [,orderby_item]... ]
[LIMIT row_count OFFSET row_offet]
```

select item
```
[[schema_name.]table_name.]*  |  expression [ [as] alias]
```

expression (returns a value)
```
simple_expression    |  (expression)      |
function(expression) |  case_expression   |
expression [+ | - | * | /] expression
```

simple_expression (returns a value)
```
column_identifier  |  literal  |  NULL
```

column identifier
```
[[schema_name.]table_name.]column_name
```

table element
```
[schema_name.]table_name [ [as] alias]  |
(query_block) [as] alias
```

join element
```
[INNER | LEFT OUTER | RIGHT OUTER | CROSS] JOIN table_element ON
join_condition
```

join_condition (returns TRUE/FALSE/NULL)
```
expression <=> expression |
condition
```

Condition (returns TRUE/FALSE/NULL)
```
simple_condition              |
(condition)                   |
condition AND | OR condition  |
NOT condition
```

simple_condition (returns TRUE/FALSE/NULL)
```
expression = | != | <> | > | >= | < | <= expression  |
expression BETWEEN expression AND expression         |
expression LIKE 'string_pattern'                     |
expression IS [NOT] NULL                             |
column [NOT] IN ( query_block | list_of_value )
```

orderby item

expression [ASC | DESC]

DESCRIPTION

Runs a query.

Notes:

- **Joins** – there is no limit to the number of tables or sub-queries in a join.
  Only join conditions that use "equals to" conditions (equi-joins) are supported.
  Only ANSI syntax is supported (`a join b on condition`).
  In the ON clause, only join conditions are supported, not single-table filter conditions.
  Self join are not supported in the same FROM clause (workaround: you can however use the same table in different sub-queries)
  Join support null safe join compare condition operator (<=>). The expression: *`T1 JOIN T2 ON T1.C<=>T2.C`* is equivalent to the expression: *`T1 JOIN T2 ON T1.C=T2.C OR (T1.C IS NULL AND T2.C IS NULL`*)
- **IN** – IN accepts a list of values or an uncorrelated sub-query
  Only a single left column is supported. Correlated sub-queries are not supported.

8.17 SHOW

```
SHOW PARAM parameter_name | ALL
SHOW TABLES [EXTENDED | MAINT]  [schema_name]
SHOW TABLE COLUMNS | PARTITIONS [schema_name.]table_name
SHOW VIEWS [EXTENDED] [schema_name]
SHOW LOCAL CACHE
SHOW ACTIVE QUERIES
SHOW SCHEMAS
SHOW JOIN INDEXES
SHOW RANGE INDEXES
SHOW SCHEDULED LOADS
SHOW VERSION
SHOW CUBES
SHOW ADAPTIVE CACHE ACTIVE
```

DESCRIPTION

Displays various aspects of the system.
Some of the command option display detailed storage usage data, for example at the table-level, partition-level or cache-level. This may take a while as those commands analyze the current storage usage.

- **SHOW PARAM** – displays the current value of a specific system parameter, or all parameters. Output specifies set value and set level using "*" sign.
- **SHOW TABLES** – lists all tables in the instance or in a specific schema.
    - **SHOW TABLES EXTENDED** – lists all tables with more detailed information, including number of columns, number of rows, and number of partitions and size of disk.
    - **SHOW TABLES MAINT** – lists the background maintenance status of all tables.
- **SHOW TABLE COLUMNS** – lists all the columns of a table, including number of distinct of values, number of NULLs and size on disk.
- **SHOW TABLE PARTITIONS** – lists all the partitions of a partitioned table, including partition boundaries, number of rows and size on disk.
- **SHOW VIEWS [EXTENDED] –** lists all views in the instance or in a specific schema. When specifying EXTENDED, also lists the view status (valid/invalid) and underling SQL statement.
- **SHOW LOCAL CACHE** – lists the content of the local cache – a summary per table.
- **SHOW ACTIVE QUERIES –** lists all in-progress queries (running or queued).
- **SHOW SCHEMAS** – lists all the schemas defined in the instance.
  Currently adding new schemas is not available.
- **SHOW JOIN INDEXES** – list all the existing join indexes and their attributes.
- **SHOW RANGE INDEXES** – list all the existing range indexes and their attributes.
- **SHOW SCHEDULED LOADS** - show all scheduled loads
- **SHOW VERSION** – show server version number.

- **SHOW CUBES** – show all generated cubes.
- **SHOW ADAPTIVE CACHE ACTIVE** – show all adaptive cache results.

8.18 UNION ALL

```
SELECT ...
UNION ALL SELECT …
[UNION ALL SELECT …]
```

DESCRIPTION

UNION All is used to combine the result from multiple SELECT statements into a single result set. The result includes all matching rows from all the SELECT statements.

8.19 SET

```
SET [SESSION|GLOBAL] parameter-name=value;
```

DESCRIPTION

Set a value to a parameter. The value can be set at session level or at global level. Setting at session level is valid only during the current session lifetime. Setting at global level is persistent and it effects the entire system and become the default value for all new sessions. If set level is not specified session level is assumed.

Parameters Values Levels Hierarchy (top override lower levels):

1. Session level setting
2. Local setting (via local-conf.ini)
3. Global level setting
4. Default value
5.

Examples:

```
set adaptive.cache.query.enable=1;
set global dynamic.aggregation.auto.generate.enable=1;
```

The rest parameter value to its default values use UNSET command.

8.20 TRUNCATE TABLE

```
TRUNCATE TABLE [schema_name.]table_name;
```

DESCRIPTION

Truncates a table (deletes all the rows from a table).

Internally, the command duplicates the table definition and marks the old copy for deletion. The old copy will be physically deleted later by the background maintenance service. New queries will see the new (empty) copy, but in-flight queries will finish executing successfully on the old copy, as if they finished execution right before the table was truncated.

8.21 UNSET

```
UNSET [SESSION|GLOBAL] parameter-name | ALL
```

DESCRIPTION

Unset set value of a parameter. The value can be unset at session level or at global level. Unset can only be used over a parameter that is values was set on the same level. If unset level is not specified session level is assumed.