SQream DB Version 2.24 - SQL Reference Guide

SQream Technologies

2019-01-22 | Version 2.24

Table of Contents

Overview

SQream SQL is based on a subset of ANSI SQL, with a little inspiration from well known SQL DBMSs.

All keywords in this guide are **case insensitive**.

1. Concepts

This section describes SQream's database concepts

1.1. SQream DB Daemon

In SQream DB, the `sqreamd` or SQream daemon is the server that deals with most of the operations against a certain GPU. Certain installations may have several `sqreamd` running, each with their own GPU ID.

A `sqreamd` may run with a specific storage cluster, or be teamed up with others, sharing a shared cluster.

1.2. Storage Cluster

A SQream DB Storage Cluster is a collection of all stored objects:

- Databases
- Schemas
- Tables
- Columns
- External Tables
- User Defined Functions
- Roles

A server instance can only run against a single storage cluster at one time. Clusters *can be changed*, but require a restart of the daemon.

A cluster will be created on installation, and shouldn't require any intervention during normal operation

1.3. Databases

A storage cluster may have many databases residing in it.

When you create an applicative connection (from a client or JDBC/ODBC) – you connect to a single database.

A database can have many Schemas and Tables.

Create different databases for different use-cases

To view existing databases, query the Catalog (information schema).

1.4. Schemas

Schemas are part of the effective table name. The default schema is called `public`.

Use schemas to split up the data tables logically

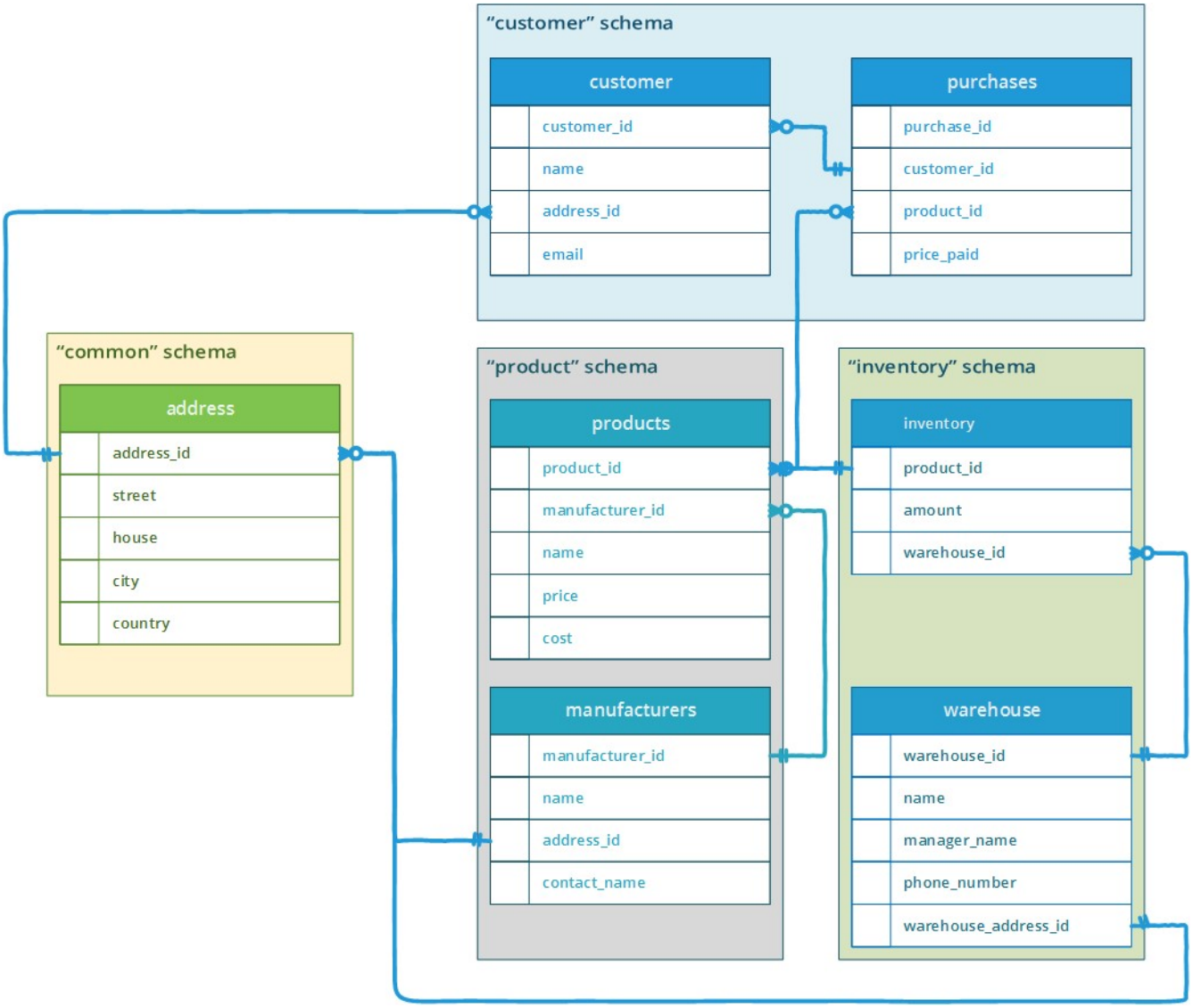

Figure 1. Example of schema usage

1.5. Tables

In SQream DB, a table is a set of data elements organized into horizontal rows and vertical columns. A table has a specified number of columns but can have any number of rows.

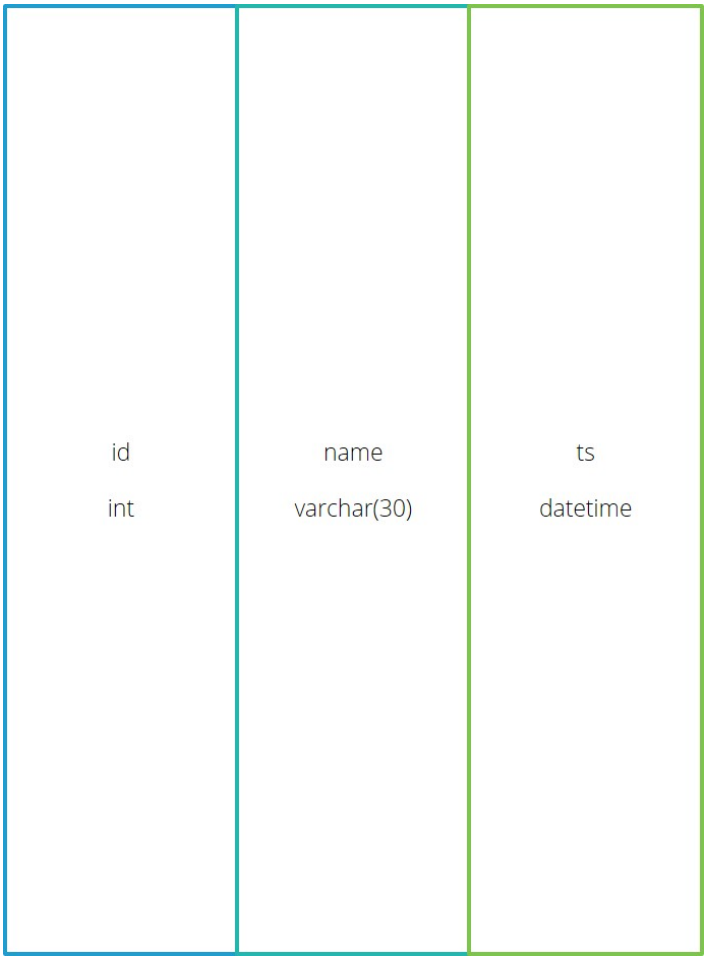

Figure 2. Logical table 'customers'

In a row-oriented database, would be stored as:
`1,John,1997-01-01 15:30:30` ; `2,Jane,1998-05-06 00:00:00;` ...

In a columnar database, would be stored as:
`1,2,3,4,5` ;
`John,Jane,Sam,David,Susan` ;
`1997-01-01 15:30:30,1998-05-06 00:00:00,1956-06-02 14:31:00,2006-12-25 14:40:00,1975-10-21 02:20:00`

Instead of pulling the entire row (all columns) every time, only select the columns you need.
SQream DB automatically removes columns not necessary for the calculations.

Example

```
SELECT name, MAX(ts) FROM customers GROUP BY name;
```

1.6. Columns

A column is the element that makes up a table.

Each column has a specific type and name, that do not change. By physically organizing data by columns, SQream DB is able to scan and aggregate data for individual columns faster, because less

data needs to be read. This makes columns highly suitable for analytical workloads.

1.7. External Tables

External tables are structured DDL that allow SQream to access data that which is stored outside the database in a none-SQream format, and to query it via SQL commands. Upon creation, the user should specify the external files format and location, and the needed table DDL. Once created, SQream will query the tables, as if they were regular tables.

1.8. User Defined Functions

User defined functions (UDF) are used by the DBA/BI/Data scientist to run custom made programs, written in Python, within SQream DB space. By using the UDF, the customers can extent the existing built-in operations in SQream, either as a row-by-row function, or as a utility function.

1.9. Chunks, Compression and Metadata

1.9.1. Chunks

SQream DB splits up columnar data into chunks. The chunk size will be the **minimal** bulk load into the GPU.

For better performance, SQream DB rearranges previously loaded data in new chunks based on the desired **chunk size**. While loading data into SQream DB, each bulk load (either `INSERT INTO` or `COPY`), will generate its own chunks (with sizes up to the chunk size).

The chunk size is a parameter at the cluster level that must be set before the first insert to the cluster. The default chunk size 1 million rows. The parameter can be set at the database level, before any tables are created. Ask your database administrator about setting the chunk size.

- SQream DB, in its own time, will **rechunk** those chunks into the desired chunk size. During the rechunk operation SQream will rearrange and recompress the data using the most appropriate compression type (according to the data type and data distribution).
- The chunk size has an influence on load/query time. Before tuning the parameter, consult your SQream account manager.

1.9.2. Compression specs

When `DEFAULT` compression spec (or no compression spec) is specified, each chunk may be compressed in a variety of different formats, based on the system's understanding. You may override the compression spec, but this is not recommended.

See Compression types in the SQL Manual for more information.

1.9.3. Metadata

SQream DB gathers and saves metadata information regarding the columns data at the chunk level during `COPY`. This information will serve the SQream optimizer while doing Data Skipping and other optimizations. This metadata is gathered automatically and transparently. It requires no user intervention.

1.10. Catalog (information schema)

The SQream DB catalog or information schema consists of views that contain information about all database objects. This provides access to database metadata, column types, tables and their row-counts, etc.

The catalog structure is specific to SQream DB.

1.10.1. Querying the SQream catalog

The catalog is available from any database, under the schema called `sqream_catalog`. You may query the schema as you would any other table in the system.

You can not perform any other operations on the catalog, like `INSERT`, `DELETE`, ...

Example

```
demo_db=> SELECT * from sqream_catalog.tables;
```

Table 1. Example result for a demo database

database_name	table_id	schema_name	table_name	row_count_valid	row_count	rechunker_ignore
demo_db	0	public	nation	1	25	0
demo_db	1	public	region	1	5	0
demo_db	2	public	part	1	20000000	0
demo_db	3	public	supplier	1	1000000	0
demo_db	4	public	partsupp	1	80000000	0
demo_db	5	public	customer	1	15000000	0
demo_db	6	public	orders	1	300000000	0
demo_db	7	public	lineitem	1	600037902	0

Example for identifying delete predicates on tables

```
demo_db=> select t.table_name,d.* from sqream_catalog.delete_predicates d
.> inner join sqream_catalog.tables t on
.> d.table_id=t.table_id;
```

1.10.2. Available catalog views

Database object catalog

Table 2. SQream catalog views

View name	Description
sqream_catalog.databases	All database objects in the current storage cluster

View name	Description
sqream_catalog.schemas	Schema objects in the database
sqream_catalog.tables	Table objects in the database
sqream_catalog.columns	Column objects in the current database
sqream_catalog.views	View objects in the database
sqream_catalog.external_tables	External table objects in the database
sqream_catalog.udf	User defined functions in the current database
sqream_catalog.catalog_tables	All catalog views available

Fine-grain storage catalog

View name	Description
sqream_catalog.extents	Extent storage objects in the database
sqream_catalog.chunks	Chunk storage objects in the database
sqream_catalog.delete_predicates	Logical delete predicates added to the compiler with a `DELETE` command

Role and permission catalog

View name	Description
sqream_catalog.roles	Roles (users) in the current databases
sqream_catalog.role_memberships	Roles membership
sqream_catalog.table_permissions	Tables and their assigned roles
sqream_catalog.database_permissions	Databases and their assigned roles
sqream_catalog.schema_permissions	Schemas and their assigned roles
sqream_catalog.permission_types	Permission types

Database objects

Databases view

Table 3. sqream_catalog.databases

Column name	Type	Description
database_id	varchar	Database ID
database_name	varchar	Database name
default_disk_chunk_size	bigint	Storage chunk size (in number of rows)
default_process_chunk_size	bigint	Process chunk size (in number of rows)
rechunk_size	bigint	Internal use
storage_subchunk_size	bigint	Internal use

Column name	Type	Description
compression_chunk_size_threshold	bigint	Internal use

Schemas view

Table 4. sqream_catalog.schemas

Column name	Type	Description
schema_id	varchar	Schema ID
schema_name	varchar	Schema name
schema_owner	varchar	Role who owns this schema
rechunker_ignore	bigint	Internal use

Tables view

Table 5. sqream_catalog.tables

Column name	Type	Description
database_name	varchar	Owning database name
table_id	varchar	Table ID
schema_name	varchar	Owning schema name
table_name	varchar	Table name
row_count_valid	bool	See warning below
row_count	bigint	Number of rows in the table
rechunker_ignore	int	Internal use

When `row_count_valid` is 0 (after a DELETE operation), the row count may be inaccurate. To get the accurate row-count, run

```
SELECT COUNT(column) FROM table;
```

Columns view

Table 6. sqream_catalog.columns

Column name	Type	Description
database_name	varchar	Owning database name
schema_name	varchar	Owning schema name
table_id	varchar	Owning table ID
table_name	varchar	Owning table name
column_id	int	Column ID
column_name	varchar	The column name
type_name	varchar	Column type

Column name	Type	Description
column_size	bigint	Column data size in bytes
has_default	int	Indicates whether or not the column has a default
default_value	varchar	Indicates the default column value
compression_strategy	varchar	User-overridden compression strategy
created	varchar	Creation date
altered	varchar	Last alter date

Views view

Table 7. sqream_catalog.views

Column name	Type	Description
view_id	varchar	View ID
view_schema	varchar	Owning schema name
view_name	varchar	The view name
view_data	varchar	Internal use
view_query_text	varchar	Full statement test that created this view

External tables view

Table 8. sqream_catalog.external_tables

Column name	Type	Description
database_name	varchar	Owning database name
table_id	varchar	External table ID
schema_name	varchar	Owning schema name
table_name	varchar	External table name
format	int	0=CSV, 1=Parquet
created	varchar	Creation data as a string

User defined functions

Table 9. sqream_catalog.udf

Column name	Type	Description
database_name	varchar	Owning database name
function_id	varchar	The function ID
function_name	varchar	The function name

Fine-grain storage

Extent view

Table 10. sqream_catalog.extents

Column name	Type	Description
`database name`	`varchar`	Owning database name
`table_id`	`varchar`	Owning table ID
`column_id`	`bigint`	Owning column ID
`extent_id`	`bigint`	The extent ID
`size`	`bigint`	Size of the extent in MB
`path`	`varchar`	Full path to the extent file on disk

Chunks view

Table 11. sqream_catalog.chunks

Column name	Type	Description
`database name`	`varchar`	Owning database name
`table_id`	`varchar`	Owning table ID
`chunk_id`	`bigint`	The chunk ID
`rows_num`	`bigint`	The amount of rows in this specific chunk
`deletion_status`	`bigint`	This chunk's deletion mark. 0 means keep, 1 means chunk needs partial deletion, 2 means delete entire chunk.

Delete predicates view

Table 12. sqream_catalog.delete_predicates

Column name	Type	Description
`database name`	`varchar`	Owning database name
`table_id`	`varchar`	Owning table ID
`max_chunk_id`	`bigint`	The highest chunk_id seen during the DELETE time
`delete_predicate`	`varchar`	The predicates added by the compiler (one predicate-statement per row in this view)

Role and permission catalog

Role view

Table 13. sqream_catalog.roles

Column name	Type	Description
role_id	bigint	The role ID
name	varchar	Role name
superuser	bool	1 for superuser, 0 otherwise
login	bool	1 if the role has login permission, 0 otherwise
has_password	bool	Does this role have a password?
can_create_function	bool	Does this role have the permissions to create/revoke user defined functions?

Role membership view

Table 14. sqream_catalog.role_memberships

Column name	Type	Description
role_id	int	The role ID
member_role_id	int	This role is member of role_id
inherit	bool	1 for inherit permission, 0 otherwise.

Table permission view

Table 15. sqream_catalog.table_permissions

Column name	Type	Description
database name	varchar	Owning database name
table_id	bigint	Owning table ID
role_id	bigint	The role ID
permission_type	int	The permission type

Database permission view

Table 16. sqream_catalog.database_permissions

Column name	Type	Description
database name	varchar	Owning database name
role_id	bigint	The role ID
permission_type	int	The permission type

Schema permission view

Table 17. sqream_catalog.schema_permissions

Column name	Type	Description
database name	varchar	Owning database name

Column name	Type	Description
schema_id	bigint	Owning schema ID
role_id	bigint	The role ID
permission_type	int	The permission type

Permission type view

Table 18. sqream_catalog.permission_types

Column name	Type	Description
permission_type_id	bigint	The permission type ID
name	varchar	Permission name

1.11. Locks

SQream DB operates in two modes: **exclusive**, which sends a single operation at a time, and **inclusive** which is a multi operations mode. DDL operations are always exclusive.

DML are separated to **DELETE/TRUNCATE** as exclusive; and **INSERT** as inclusive. This allows multiple inserts into the table, but prevents multiple **DELETE** operations.

Querying (**SELECT** operations) can coexists with both DDL and DML.

1.11.1. Locking

Table 19. Locks by SQream

Operation	Select	DML (Insert)	DML (Delete/Truncate)	DDL
Select	No lock	No lock	No lock	No lock
DML (insert)	No lock	No lock	No lock	Lock
DML (delete/truncate)	No lock	No lock	Lock	Lock
DDL	No lock	Lock	Lock	Lock

By default, when a session is requesting a lock on an object and the object is busy, SQream will wait 3 seconds before it return an error message. This wait time is defined in the configuration JSON. See the `statementLockTimeout` parameter in SQream Administrator Guide for more information.

DDL on an object will prevent other DDL/DML to wait on a lock on the same object.

For specific DDL operations, SQream uses global permissions that requires very short exclusive locks at the cluster level. Global permission will be used on operation such as **CREATE DATABASE/TABLE/ROLE**, **ALTER ROLE/TABLE**, **DROP ROLE/TABLE/DATABASE**, **GRANT/REVOKE**.

1.11.2. Viewing locks

To view all existing locks in the SQream database use the utility function show_locks() :

Example

```
SELECT show_locks();
```

1.11.3. Releasing locks

To release a specific lock in an active SQream instance, use the **stop_statement()** utility function with the relevant statement_id. Use the statement_id returned by the **show_locks()** utility function.

Example

```
SELECT stop_statement(12009);
```

To release all locks in a suspect/inactive SQream instance, use the utility function **release_defunct_locks()** which will remove the instance from the cluster and release all its resources. To see all SQream instances status, use **show_cluster_nodes()**.

The utility function **release_defunct_locks()** works only for a system running the metadata server. In a single instance (no metadata server) it will not work.

Example

```
SELECT show_cluster_nodes();
SELECT release_defunct_locks();
```

1.12. Workload Manager

SQream will distribute work throughout the hardware resources to maximize the hardware utilization. By default, this distribution will be done in an equal manner.
The DBA can change this setting and optimize the utilization to their needs by using SQream workload manager and defining each SQream instance to specific service/s.
The specific service to connect to is defined in the session connection string, with the property 'service'. Default service name is: sqream (for more details, see each driver connection string specification).

Each SQream instance can serve multiple services, and each service can work with multiple SQream instances.

1.12.1. Managing Services

Monitor services subscription

```
select show_subscribed_instances();
```

Instance ID is a unique identifier, defined by SQream at the installation, for each instance in SQream cluster.

Example

```
select  show_subscribed_instances('etl_service');
select  show_subscribed_instances();
```

Add services to an existing instance

```
select subscribe_service('instance_id', 'service_name'); ;
```

Example

```
select subscribe_service('node_11', 'etl_service');
```

Remove services to an existing instance

```
select unsubscribe_service('instance_id', 'service_name'); ;
```

Example

```
select unsubscribe_service('node_11', 'etl_service');
```

You can't unsubscribe the last instance from existing service with working/waiting statements in it's queue.

2. SQL Reference

This is the reference for the SQL statements supported by SQream. For a conceptual overview and recommendations on best practices regarding SQream internal behavior, please refer to SQream Administration Guide.

2.1. Data Definition Language

2.1.1. Databases

CREATE DATABASE

CREATE DATABASE will create a new database in the current cluster storage set. To use the new database, disconnect and reconnect to the new database.

```
create_database_statement ::=

    CREATE DATABASE database_name ;

database_name ::= identifier
```

Examples

```
create database my_database;
```

DROP DATABASE

DROP DATABASE will delete a database and all of its files. During **DROP DATABASE**, no other DDL/DML operations can run on the database, and subsequent connections to it will fail.

```
drop_database_statement ::=

    DROP DATABASE database_name ;
```

Examples

```
drop database my_database;
```

2.1.2. Schemas

CREATE SCHEMA

CREATE SCHEMA will create a new schema in the current database.

```
create_schema_statement ::=

    CREATE SCHEMA schema_name ;

schema_name ::= identifier
```

New tables should be explicitly associated to an existing schema, or implicitly associated to the PUBLIC default schema.

Examples

```
create schema my_schema;
```

DROP SCHEMA will delete an empty schema from the database.

DROP SCHEMA

```
drop_schema_statement ::=

    DROP SCHEMA schema_name ;
```

The PUBLIC schema can not be dropped.

Examples

```
DROP SCHEMA my_schema;
```

ALTER DEFAULT SCHEMA

Change the default schema for a specific role. Alter default schema should be used to change the user/role default schema to a different schema name.

```
alter_default_schema_statement ::=
    ALTER DEFAULT SCHEMA FOR  role_name TO schema_name;
```

Examples

```
ALTER DEFAULT SCHEMA FOR user_a TO schema_a;
```

2.1.3. Tables

CREATE TABLE

CREATE TABLE creates a new table in the current database under a specific schema.

- This operation requires exclusive lock on the table.
- New tables should be explicitly associated to an existing schema. Otherwise, they will be implicitly associated to the PUBLIC default schema.
- Max row length cannot exceed 10239 bytes.

```
create_table_statement ::=

    CREATE TABLE [schema_name].table_name (
        { column_name type_name [ default ]
          [ column_constraint ] }
        [, ... ]
    )
    ;

schema_name ::= identifier

table_name ::= identifier

column_name ::= identifier
```

`identifier` is defined below in the identifier definition section.

`type_name` is defined below in the type name definition section.

Constraints and defaults

```
column_constraint ::=

    { NOT NULL | NULL }

default ::=

    DEFAULT default_value

    | IDENTITY [ ( start_with [ , increment_by ] ) ]
```

The `default_value` can be NULL or a literal.

The common sequence generator options can be comma or whitespace separated.

Identity columns are only supported for columns of type BIGINT.

Identity does not enforce uniqueness of the value. When the value in the identity column reaches the maximum number for the specific column datatype limitation, the next number in the identity will restart as 1.

Compression types

```
compression_type ::=
    CHECK ( 'CS "compression_name"' )
```

SQream recommends using the default compression type by omitting the compression specification, which defaults to automatic compression.

You may override the default compression by specifying the `check` modifier. For example, `check ('CS "p4d"')`.
Please contact SQream support for more information about recommended compressions.

Examples

```
create table t (
    a bigint identity (1,1) CHECK ( 'CS "default"' ),
    b int);

create table my_schema.t (
    a int null CHECK ( 'CS "p4d"' ),
    b int not null CHECK ( 'CS "dict"' ));

create table u (
    a int default 0,
    b int,
    c date);

create table u (
    k bigint not null identity,
    v varchar(10) CHECK ( 'CS "dict"' ));

create table u (
    k bigint not null identity(1,1),
    v varchar(10));

create table t(x int not null,
               y int default 0 not null);
```

Special use case: Create a table from an existing table. The new table will be populated with the records from the existing table (based on the SELECT Statement).

```
create table customers_new as select * from customers;
```

Create Or Replace Table

CREATE OR REPLACE TABLE will either create a new table (if the same table doesn't already exists) or DROP and CREATE the table with its new definition.

Examples

```
create or replace table t (
    a bigint identity (1,1),
    b int
);


CREATE OR REPLACE TABLE t AS select * from sqream_catalog.tables;
```

If the CREATE TABLE operation does not complete successfully, the replaced table (t in this example) will no longer exist.

ALTER TABLE

ALTER TABLE is used to alter the structure of an existing table.

This operation will require exclusive lock on the table.

Rename table

This form of alter table allows you to rename a table within the same schema.

```
alter_table_statement_rename_table ::=

    ALTER TABLE [schema_name].table_name RENAME TO new_table_name ;
```

Examples

```
ALTER TABLE my_table RENAME TO your_table;
```

Rename column

This form of alter table allows you to rename a column in an existing table.

```
alter_table_statement_rename_column ::=

    ALTER TABLE [schema_name].table_name
        RENAME COLUMN column_name TO new_column_name ;
```

Examples

```
ALTER TABLE my_table RENAME COLUMN col1 to col2;
```

Add column

Add a new column to an existing table.

Known Limitations

1. When adding a new column to an existing table, a default (or nullability) has to be specified, even if the table is empty.
2. The new column can not contain an **IDENTITY** or an **NVARCHAR**.

```
alter_table_statement_add_column ::=

    ALTER TABLE [schema_name].table_name ADD COLUMN column_name type_name default [
column_constraint ]
    ;
```

Examples

```
ALTER TABLE my_table ADD COLUMN new_supercool_column BIGINT default 1;
-- Adds a new column of type nullable BIGINT, with default value of 1.

ALTER TABLE my_table ADD COLUMN new_supercool_column BIGINT default 1 NOT NULL;
-- Adds a new column of type non-null BIGINT, with default value of 1.

ALTER TABLE my_table ADD COLUMN new_date_col date default '2016-01-01';
-- Adds a new column of type nullable date, with default date '2016-01-01'.
```

Drop column

Drop a column from an existing table This form of alter table allows you to drop a column from an existing table.

```
alter_table_statement_rename_table ::=

    ALTER TABLE [schema_name].table_name drop column column_name ;
```

Examples

```
ALTER TABLE my_table DROP dreadful_column_i_never_even_wanted_in_my_table;
```

DROP TABLE

DROP TABLE will delete a table or external table and all its data. Note that this operation will require exclusive lock on the table.

- Dropping a table without explicit `schema_name`, will drop the table under the default PUBLIC schema.
- To be able to drop tables, a role requires the **superuser** permission. See the SQream Administrator Guide for more details.

```
drop_table ::=

    DROP TABLE [IF EXISTS] [schema_name].table_name ;
    DROP TABLE [IF EXISTS] [schema_name].external_table_name ;
```

Examples

```
DROP TABLE my_schema.my_table;
```

DROP TABLE IF EXISTS will either drop the table if it exists, or do nothing (other than returning an error that the table do not exists).

Examples

```
drop table if exists my_schema.my_table;
```

2.1.4. External Tables

CREATE EXTERNAL TABLE

CREATE EXTERNAL TABLE creates a new external table in the current database under a specific schema. External tables allow SQream to access data that is stored outside the database in a none-SQream format, and to query it via SQL commands.
While creating it, the user should specify the files format and location, these will be used by SQream while accessing it at the query execution time.

- SQream accesses the files only at execution time, hence, errors in case of file/DDL mismatch will not be captured during table creation time.
- New external tables should be explicitly associated to an existing schema. Otherwise, they will be implicitly associated to the PUBLIC default schema.
- Max row length cannot exceed 10239 bytes.

```
CREATE [OR REPLACE] EXTERNAL TABLE [schema_name.]table_name
        (column_name column_type [, ...])
   USING FORMAT {PARQUET | CSV}
   WITH PATH '{file_name | directory_path}'
   [FIELD DELIMITER 'delimiter']
   [RECORD DELIMITER 'record delimiter'];
```

- **WITH PATH**:
  Contains either a (1) file name or a (2) path to a directory in which all the CSV or PARQUET files will be read.
  You can use wildcard characters for file name and directory path. Standard wildcard characters for Linux are used.
  Hidden files (files with file names that start with a dot) are skipped.

Wildcard options:

- `*` - Represent zero or more characters.
- `?` - Represent a single character.
- `[a-z]` - Represent a range of characters or numbers.

- **FIELD DELIMITER**:
  Allows to specify a character(s) as delimiter between fields in a row. Default is `,` (comma) the CSV standard field delimiter.
- **RECORD DELIMITER**:
  Allows to specify a character(s) as delimiter between lines. Default is `\n`, the CSV standard record delimiter.

Examples

```
CREATE OR REPLACE EXTERNAL TABLE my_ext_table (column1 int, column2 varchar(10),
column3 date)
USING FORMAT csv
WITH PATH '/home/sqream/my_csv_file.csv';

CREATE OR REPLACE EXTERNAL TABLE my_ext_table (column1 int, column2 varchar(10),
column3 date)
USING FORMAT parquet
WITH PATH '/home/sqream/my_parquet_file.parquet';

CREATE OR REPLACE EXTERNAL TABLE my_ext_table (column1 int, column2 varchar(10),
column3 date)
USING FORMAT csv
WITH PATH '/home/sqream/my_csv_file.csv'
FIELD DELIMITER '|'
RECORD DELIMITER '\n';

-- Use wildcard characters for directory name and file name:
CREATE OR REPLACE EXTERNAL TABLE ext_my_table
  ( id bigint,
    name varchar(100),
    location varchar(25),
    comments nvarchar(10000)
  )
    USING FORMAT csv
    WITH PATH '/home/sqream/csv_?/*.csv';
```

CONVERSION TABLE FOR DATATYPES OF EXTERNAL TABLES OVER PARQUET

Table 20. Conversion table

Data type in Parquet	Matching data type in SQream	Comments
BOOLEAN	BOOL	
INT16	SMALLINT	
INT32	INT	
INT64	BIGINT	
FLOAT	REAL	
DOUBLE	DOUBLE	
BYTE_ARRAY with annotation UTF8	VARCHAR or NVARCHAR	
DATE	DATE	
INT 96 (TIMESTAMP_MILLIS)	DATETIME	INT96 may contain microseconds. As DATETIME in SQream supports milliseconds only, the microseconds are rounded to milliseconds.

2.1.5. Views

CREATE VIEW

CREATE VIEW creates a new view in the current database.

```
create_view_statement ::=

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


view_name ::= identifier
```

Example

```
CREATE VIEW [schema_name].my_view as select * from my_schema.t where x > 5;
```

DROP VIEW

DROP VIEW will delete a view from the current database.

```
drop_view_statement ::=

    DROP VIEW [schema_name].view_name ;
```

Example

```
DROP VIEW my_schema.my_view;
```

RECOMPILE VIEW

Use the utility function **RECOMPILE_VIEW** to recompile an invalid view in the current database.

```
recompile_view_statement ::=

    SELECT RECOMPILE_VIEW([schema_name].view_name) ;
```

Examples

```
SELECT recompile_view('my_view_name');
```

2.1.6. Users/Roles

CREATE ROLE

SQream manages users by *roles*. **CREATE ROLE** adds a new user/role to the cluster.

When the **ROLE** is used as a **USER**, it has to be granted a password, with login and connect privilege to the relevant databases.

For more information regarding **ROLES** and permissions, see Managing Database Roles and Permissions

```
create_role_statement ::=

    CREATE ROLE role_name ;
    GRANT LOGIN to role_name ;
    GRANT PASSWORD 'new_password' to role_name ;
    GRANT CONNECT ON DATABASE database_name to role_name ;
```

Examples

```
CREATE ROLE new_role_name ;
GRANT LOGIN TO new_role_name;
GRANT PASSWORD 'my_password' TO new_role_name;
GRANT CONNECT ON DATABASE master TO new_role_name;
```

DROP ROLE

DROP ROLE deletes a role/user.

```
drop_role_statement ::=

    DROP ROLE role_name ;
```

Example

```
DROP ROLE admin_role;
```

ALTER ROLE

Use **ALTER ROLE** to rename an existing role.

```
alter_role_statement ::=

    ALTER ROLE role_name RENAME TO new_role_name ;
```

Example

```
ALTER ROLE admin_role RENAME TO copy_role;
```

2.1.7. Database Roles and Permissions

SQream manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a set of database permissions, depending on how the role is set up.

Roles can have permissions and access database objects (for example, tables). **SUPERUSER** can grant membership in a role to another role, thus allowing the use of permissions assigned to the role it is a member of.

Roles are global across all the databases.

For a role to be used as a user, it must have **PASSWORD**, **LOGIN** and **CONNECT** permissions to a specified database(s).

For a role to create and manage (read/write/alter) objects, it has to have the **CREATE** and **USAGE** permissions at the respective SCHEMA level.

When creating an object, it will be created in the role's default schema. Typically the 'default schema' is **public**. You can change the default schema with **ALTER DEFAULT SCHEMA**.

The owner is normally the role that executed the creation statement. Once the object was created, only the owner (or a superuser) can perform read/write/alter operations on the object.

To allow other roles perform these operations, permissions must be granted. These permissions can be given in advance for a given schema in a given role with the command **ALTER DEFAULT PERMISSIONS** (as long as they are sharing the same group).

PUBLIC Role

On database creation, SQream automatically generates the **sqream** user as a superuser and the **PUBLIC** schema and role. Each new user will automatically be granted with the PUBLIC role which cannot be revoked.

The PUBLIC role has USAGE and CREATE permissions on PUBLIC schema by default, therefore, new users can create and manage their own objects in the **PUBLIC** schema. To see/manage other users/role objects in PUBLIC schema, use the **GRANT** command while connected as the object owner or as a superuser.

GRANT Permissions

The **GRANT** command assigns specific permissions to existing database objects or at the database level, to one or more roles.

The identifier name can be the role name or CURRENT_ROLE.

Possible permissions at the cluster level: LOGIN, PASSWORD, CREATE FUNCTION

Possible permissions at the database level: SUPERUSER, CONNECT, CREATE, USAGE

Possible permissions at the schema level: USAGE, CREATE

Possible permissions at object level: SELECT, INSERT, DELETE, DDL, EXECUTE, ALL.

- SQream allows passing privilege between roles by granting one role to another, with or without inherit.
- For a role to act as a user, it must be granted with *PASSWORD*, *LOGIN*, and *CONNECT* permission to the database it wishes to connect to.
- For a role to create and manage (read/write/alter) objects, it has to have the CREATE and USAGE permissions in that SCHEMA level.

```
-- Grant permissions at the cluster level:
    GRANT
    { SUPERUSER
    | LOGIN
    | PASSWORD '<password>'
    | CREATE FUNCTION
    }
    TO <role> [, ...]

-- Grant permissions at the database level:
    GRANT { { CREATE | CONNECT | DDL | SUPERUSER } [, ...] | ALL [ PERMISSIONS ] }
    ON DATABASE <database> [, ...]
    TO <role> [, ...]

-- Grant permissions at the schema level:
    GRANT { { CREATE | DDL | USAGE | SUPERUSER } [, ...] | ALL [ PERMISSIONS ] }
    ON SCHEMA <schema> [, ...]
    TO <role> [, ...]

-- Grant permissions at the object level:
    GRANT { { SELECT | INSERT | DELETE | DDL } [, ...] | ALL [ PERMISSIONS ] }
    ON { TABLE <table_name> [, ...]  | ALL TABLES IN SCHEMA <schema_name> [, ...] }
    TO <role> [, ...]

-- Grant execute function permission:
    GRANT {ALL | EXECUTE | DDL} ON FUNCTION function_name TO role;


-- Pass privileges between roles by granting one role to another:
    GRANT <role1> [, ...] TO <role2> [, ...] WITH [ INHERIT | NOINHERIT ]
```

SELECT: Allows to SELECT any column of the specified table.

INSERT: Allows INSERT of a new row into the specified table using INSERT/COPY command.

DELETE: Allows to DELETE a row from the specified table. This privilege effects both DELETE and TRUNCATE commands.

DDL: Allows DROP/ALTER on a specified table.

SUPERUSER: The most privileged role in SQream cluster, allowing full control over the cluster. By default, SQream role is a superuser. Some utility functions can be used by superusers only.

SUPERUSER ON SCHEMA: Having maximum permissions on existing and new objects for a specific schema.

LOGIN: Grants the login permission. Without it, a role cannot function as a USER and login.

PASSWORD: Grant a new password for the role. Without a password the role cannot function as a USER and login.

INHERIT/NOINHERIT: While INHERIT is true (default), permissions belonging to the superclass role, will automatically be transferred to the heir role.

The permissions LOGIN cannot be inherited and must be explicitly granted.

Examples

```
GRANT LOGIN,superuser TO admin;
GRANT CREATE FUNCTION TO admin;
GRANT SELECT ON TABLE admin.table1 TO userA;
GRANT EXECUTE ON FUNCTION my_function TO userA;
GRANT ALL ON FUNCTION my_function TO userA;
GRANT DDL ON admin.main_table TO userB;
GRANT ALL ON all tables IN schema public TO userB;
GRANT SELECT ON all views IN schema admin TO userA;
GRANT ROLE admin TO userC noinherit;
GRANT superuser ON schema demo TO userA;
GRANT admin_role TO userB;
```

REVOKE Permissions

The **REVOKE** command removes permissions from one or more roles.

The identifier name can be the role name or CURRENT_ROLE.

```
-- Revoke permissions at the cluster level:
    REVOKE
    { SUPERUSER
    | LOGIN
    | PASSWORD
    | CREATE FUNCTION
    }
    FROM <role> [, ...]

-- Revoke permissions at the database level:
    REVOKE { { CREATE | CONNECT | DDL |  SUPERUSER } [, ...] | ALL [ PERMISSIONS ] }
    ON DATABASE <database> [, ...]
    FROM <role> [, ...]

-- Revoke permissions at the schema level:
    REVOKE { { CREATE | DDL | USAGE |  SUPERUSER } [, ...] | ALL [ PERMISSIONS ] }
    ON SCHEMA <schema> [, ...]
    FROM <role> [, ...]

-- Revoke permissions at the object level:
    REVOKE { { SELECT | INSERT | DELETE | DDL } [, ...] | ALL }
    ON { [ TABLE ] <table_name> [, ...] | ALL TABLES IN SCHEMA <schema_name> [, ...] }
    FROM <role> [, ...]

-- Pass privileges between roles by granting one role to another:
    REVOKE <role1> [, ...] FROM <role2> [, ...]
```

Examples

```
REVOKE superuser on schema demo from userA;
REVOKE delete on admin.table1 from userB;
REVOKE login from role_test;
REVOKE CREATE FUNCTION FROM admin;
```

Alter Default Permissions

ALTER DEFAULT PERMISSIONS allow setting/removing permissions to objects created in the future as long as they are sharing the same group. By using this command, the role who run it controls which grants will be given on each object to each roles.

```
    ALTER DEFAULT PERMISSIONS FOR role, ...
    ( [IN schema, ...] FOR TABLES | FOR SCHEMAS )
    ( GRANT | DROP GRANT ) ... TO xrole, ...;

xrole := role | CREATOR_ROLE
```

Examples

```
-- Anytime role DEMO creates a new schema, roles U1,U2 will get USAGE and CREATE
permissions in the new schema:
ALTER DEFAULT PERMISSIONS FOR demo FOR SCHEMAS GRANT USAGE, CREATE TO u1,u2;

-- Anytime role DEMO create a new table in schema S1, roles U1,U2 will be granted with
SELECT on it:
ALTER DEFAULT PERMISSIONS FOR demo IN s1 FOR TABLES GRANT SELECT TO u1,u2;

-- To grant all schemas so they can all create and use all future schemas in all
possible operations, use the following:
ALTER DEFAULT PERMISSIONS FOR public for schemas GRANT USAGE, CREATE TO public;
ALTER DEFAULT PERMISSIONS FOR public FOR TABLES GRANT SELECT,DDL,INSERT,DELETE TO
public;

-- To remove future grant, add the keyword DROP before the given grant:
ALTER DEFAULT PERMISSIONS FOR public FOR TABLES DROP GRANT SELECT,DDL,INSERT,DELETE TO
public;
```

2.1.8. Locks

See Locks in the Concepts section.

2.1.9. DDL for a whole database or single database object

Generate the DDL for all objects of a database

Use the utility function **dump_database_ddl()** to generate the DDL for all tables and views of the database you are currently connected to.

```
-- Just view
SELECT dump_database_ddl();

-- Output to file
COPY (SELECT dump_database_ddl()) TO '/path/file_name';
```

Generate the DDL of a single table, external table or view

Use the utility function **get_ddl('table_name')** or **get_view_ddl('view_name')** to generate the DDL for a specified table, external table or view.

If the table name has a numeric prefix (e.g. "2018_my_table"), the name must be wrapped with single and double quotes. If the schema name or table/view name is case sensitive, they must be wrapped with single and double quotes. See examples below.

```
-- Just view
SELECT get_ddl('my_table');
SELECT get_ddl('"2018_my_table"');
SELECT get_ddl('my_external_table');
SELECT get_view_ddl('my_view_name');
SELECT get_view_ddl('"My_schema"."My_View"');

-- Output to file
COPY (SELECT get_ddl('my_table')) TO '/path/file_name';
```

Generate the DDL of a single user defined function (UDF)

Use the utility function **get_function_ddl()** to generate the DDL for a specified UDF.

```
select get_function_ddl('user_function_name');
```

2.2. Data Manipulation Language

This section covers updates to the data in tables. Queries are in their own section which follows this one.

2.2.1. INSERT

INSERT is used to add rows to a table.

```
insert_statement ::=

    INSERT INTO [schema_name].table_name
        [ ( column_name [, ... ] ) ]
    query ;
```

Examples

```
INSERT INTO my_schema.dst1 SELECT a,b,c from src;

INSERT INTO dst2(d1, d3) SELECT d1,d3 from src;

INSERT INTO t(cint,cint_2) VALUES(1,3);

INSERT INTO t VALUES(1,3);
```

When the insert statement does not include all the columns in the table, columns which aren't explicitly mentioned will get their default values (string/number, NULL or identity)

2.2.2. COPY FROM (bulk import)

COPY FROM is used to quickly insert data from CSV files into a table. It is the recommended way to ingest data.

The copy command will always insert data to one table. When using the directory option, all the files in that directory will be loaded into the same table.

```
copy_from_statement ::=

    COPY [schema_name].table_name [ ( column_name [, ... ] ) ]
        FROM 'file_name | directory_path'
        [ [ WITH ] ( option [ ...] ) ]
        ;

    with option can be one of:
        OFFSET N
    LIMIT N
    DELIMITER 'delimiter'
    RECORD DELIMITER 'record delimiter'
        ERROR_LOG 'error_log_filename'
        ERROR_VERBOSITY { 0 | 1 }
    STOP AFTER N ERRORS
        PARSERS { '[column_name=parser_format, ...]' }
```

- **Files/Directory**:
  Copy command can either load a specific file or load the entire directory. In both cases, the directory and the file should be available to the server process on the host machine.
  You can use wildcard characters for file name and directory path. Standard wildcard characters for Linux are used. Hidden files (files with file names that start with a dot) are skipped.

Wildcard options:

- `*` - Represent zero or more characters.
- `?` - Represent a single character.
- `[a-z]` - Represent a range of characters or numbers.

- **Offset**:
  The load will start with the offset requested row number. When being used in copy from directory, the offset number will affect each file that is being loaded.
- **Limit**:
  The load will stop with the requested limit row number. When being used in copy from directory, the limit will affect each file that is being loaded.
- **Delimiter**:
  The default field delimiter is: `,` (comma). The field delimiter can be any single printable ascii character (32-127), or ascii characters between 1 and 126 enclosed by E'\' (for example: E'\001') excluding the following: '\', 'N', '-', ':', "", ascii code 10, ascii code 13, all lower case characters, all digits.

For loading string that contains the column delimiter in them (like a comma or tab), surround the whole string with double quotes ("string").

For loading string that contains double quotes in them, enclose each of the double quotes (") with another double quotes (csv data: my""string will be loaded as my"string).
This is similar to the string literal escaping method

- **Record delimiter**:
  The record delimiter must be one of Windows (`\r\n`) / Linux (`\n`) / Mac (`\r`) new line characters.

The copy command will always insert data to one table. When using the directory option, all the files in that directory will be loaded into the same table.

- **Error_log**:
  When not using error log, SQream will stop the load at the first error message and do rollback to the entire copy. For allowing SQream to load valid rows despite the errors, use the ERROR_LOG with the following options:
- **Stop after N errors**:+ Stop after N errors allow to load valid rows and ignore errors up until a certain amount of errors (N). If the number of errors in the load reaches the given N, all will be rolled back.
- **Error verbosity**:
  1. 0 - only the bad line is printed into the error log file (without the actual error message) - for replaying the error log back to the server.
  2. 1 - both bad line and error message are printed at the error log file - for debugging.

Using the COPY command without the ERROR_VERBOSITY option will fail the entire load upon the first error message.

- **Parsers**:
  Parses allows specifying a different date-format than the default (ISO8601) for DATE or DATETIME columns.

Examples

```
-- Copy from row 2 (ignore header), and with Windows newline format
COPY table_name from 'filename.csv' with DELIMITER ','
                                        RECORD DELIMITER '\r\n'
                     OFFSET 2
                                        error_log 'error.log'
                                        error_verbosity 0;

COPY table_name from 'filename.csv' with delimiter '|'
                                        error_log '/temp/load_err.log'
                                        offset 10
                                        limit 100
                                        stop after 5 errors;

COPY table_name from '/full_path_directory/' with delimiter '|'
                                                parsers 'date_column=iso8601';

-- Use wildcard characters to load one level of CSV files:
COPY [schema_name].table_name [ ( column_name [, ... ] ) ]
        FROM '/home/sqream/csv/*.csv';

-- Load all files within the specified directory and its sub-directories:
COPY [schema_name].table_name [ ( column_name [, ... ] ) ]
        FROM '/home/sqream/csv/';

-- Use wildcard characters to load all files in  a directory beginning with a, of
format CSV:
COPY [schema_name].table_name [ ( column_name [, ... ] ) ]
        FROM '/home/sqream/a*/*.csv';

-- Use wildcard characters to load all files, in CSV format, in a directory beginning
with "Jan_" and followed by two characters. for example: "Jan_31:
COPY [schema_name].table_name [ ( column_name [, ... ] ) ]
        FROM '/home/sqream/Jan_??/*.csv';
```

Table 21. Supported date formats and their parsing

Format Name	Format Pattern	Example	Note
ISO8601 or DEFAULT	YYYY-MM-DD [hh:mm:ss[.SSS]]	2017-12-31 11:12:13.456	
ISO8601C	YYYY-MM-DD [hh:mm:ss[:SSS]]	2017-12-31 11:12:13:567	Milliseconds are separated by a colon (:).
DMY	DD/MM/YYYY [hh:mm:ss[.SSS]]	31/12/2017 11:12:13.000	In versions prior to V2.11 this format was called "British".
YMD	YYYY/MM/DD [hh:mm:ss[.SSS]]	2017/12/31 11:12:13.678	
MDY	MM/DD/YYYY [hh:mm:ss[.SSS]]	12/31/2017 11:12:13.456	

Time parts in brackets [] are optional. If the time part is missing, it will be set to 00:00:00.000. If milliseconds are missing, they are set to 000.

Milliseconds are stored as 3 digits. If milliseconds are ingested with more than 3 digits, the system will round them to 3 digits.

Table 22. Date Format Pattern Syntax

Pattern	Explanation
YYYY	four digit year representation (0000-9999)
MM	two digit month representation (01-12)
DD	two digit day of month representation (01-31)
m	short month representation (Jan-Dec)
a	short day of week representation (Sun-Sat).
hh	two digit 24 hour representation (00-23)
h	two digit 12 hour representation (00-12)
P	uppercase AM/PM representation
mm	two digit minute representation (00-59)
ss	two digit seconds representation (00-59)
SSS	3 digits fraction representation for milliseconds (000-999)

2.2.3. COPY TO (bulk export)

COPY TO is used to save query results to a file in CSV format.

```
copy_to_statement ::=

    COPY ( query ) TO 'path/file_name'
        [ [ WITH ] ( option [...] ) ]
    ;

    with option can be:
        DELIMITER 'delimiter'
                HEADER
```

- The target path must be accessible by the server process.
- If the specified file already exists, it will be overwritten.

- **Delimiter**:
  The default field delimiter is: , (comma). The field delimiter can be any single printable ascii character (32-127), or ascii characters between 1 and 126 enclosed by E'\' (for example: E'\001') excluding the following: '\', 'N', '-', ':', "", ascii code 10, ascii code 13, all lower case characters, all digits.

- **Header**:
  This option adds the column names as the first line (header line) of the output CSV file, using specified field and line delimiters. If COPY is done from a table it adds the natural column names as they appear in the metadata. If COPY is done from a query the system creates an alias name for calculated fields (e.g., SUM(amount)). For calculated fields it is therefore highly recommended to specify a column alias name. Column alias names follow the identifier rules (see identifier definition section).

Examples

```
COPY my_table TO '/path/file_name';
COPY my_table TO '/path/file_name' with HEADER;
COPY my_table TO '/path/file_name' with DELIMITER '|' HEADER;
COPY (select column_a, column_b from my_table where column_a>'2016/01/01')  TO
'/path/file_name';
COPY (select customer_id, sum(sales) as sum_sales from sales group by 1) TO
'/path/file_name' with HEADER;
```

2.2.4. TRUNCATE TABLE

The **TRUNCATE TABLE** command deletes all the rows from a table. It has the same effect as a <<delete,**DELETE**> operation on the table without any conditions, but since it does not actually scan the structures, it is much faster.

```
truncate_statement ::=

    TRUNCATE TABLE [schema_name].table_name [ RESTART IDENTITY | CONTINUE IDENTITY ] ;
```

Using **RESTART IDENTITY** will reset the identity columns to their starting values. **CONTINUE IDENTITY** is the default.

See also **DELETE** which provides the ability to delete rows that satisfy a predicate.

Examples

```
truncate table my_schema.t;
```

2.2.5. DELETE

Delete rows from a table.

The **DELETE** command performs a *logical* deletion of rows that satisfy the WHERE predicate from the specified table. As the delete command is only a *logical delete*, it retains data on disk until a clean-up process is performed.
To complete the logical delete with physical removal from disk, use the cleanup utilities (see below).

Only roles with the **DELETE** permission granted may delete from tables.

Logical Delete

The projected result set for queries will not contain the deleted data. Data is marked for deletion, but not physically deleted from disk.

Physical Delete (Cleanup)

Files marked for deletion during the *logical deletion* stage are removed from disk. This is achieved by calling both utility function commands: **CLEANUP_CHUNKS** and **CLEANUP_EXTENTS** sequentially.

During physical delete some files might be rebuilt based on how the data was distributed on disk. This may use up some additional disk space.

```
delete_statement ::=

    DELETE FROM [schema_name].table_name
         [ WHERE condition ] ;

cleanup_utilities ::=

    SELECT CLEANUP_CHUNKS ( schema_name, table_name ) ;

    SELECT CLEANUP_EXTENTS ( schema_name, table_name ) ;
```

TRUNCATE provides a much faster alternative to remove all rows from a table.

Best Practices

1. Apply the **WHERE** condition to a sorted column where possible.
2. To clear an entire table, use TRUNCATE.
3. To optimize performance, after the logical DELETE, run the cleanup (physical) delete.

Known Limitations

1. Unlike some other databases, this command does not return the number of rows affected or deleted.
2. It is not possible to ALTER a table that has not been cleaned up.
3. It is currently not possible to delete rows in a table using information contained in other tables in the database (subqueries or JOINs).
4. During the logical deletion process, the table is locked for TRUNCATE, ALTER, DROP and other DELETE commands.
5. A long delete operation will not execute if it exceeds SQream configuration setting.

Following the recommended best practices, the logical delete operation will first analyze and estimated time the delete should take based on the amount of I/O to delete and the data distribution. If SQream DB finds that the estimated time is beyond the best practices delete time, an error message will return and the user will have to do manual setting to overcome this and continue with the delete (see more information in the relevant error message).

Examples

```
-- Delete all rows from 'books' table for books introduced before 2012
DELETE FROM books WHERE date_introduced < '2012-01-01';

-- Clear the 'books' table completely:
DELETE FROM books;

-- Rearrange data on disk prior to physical deletion (SWEEP)
SELECT CLEANUP_CHUNKS('public','books');

-- Delete leftover files (VACUUM)
SELECT CLEANUP_EXTENTS('public','books');
```

2.3. Operational Commands

2.3.1. Saved queries

Saved queries allow SQream DB to save the query plan for a query. Saved query will save the compiler time on each execution, and therefore can help optimize the total query execution time.

Examples

- Save a query:

```
select save_query('q1', $$select * from t where xint > ? AND xdatetime < ? AND
xvarchar6 <> 'something'$$)
```

- Execute the saved query

```
select execute_saved_query('q1', 1, '2013-12-02 12:01:22')
```

The result: SQream DB will execute the query:

```
select * from t where xint > 1 AND xdatetime < '2013-12-02 12:01:22' AND xvarchar6 <>
'something';
```

The saved query names must be unique in the database and should be defined in lower case.

2.3.2. Create saved query

```
SELECT save_query ( saved_query_name , parameterized_query_string ) ;

saved_query_name ::= string_literal

parameterized_query_string ::= string_literal
```

Execute saved query

```
SELECT execute_saved_query ( saved_query_name [ , argument [ , ... ] ] ) ;

argument ::= string_literal | number_literal
```

Drop saved query

```
SELECT drop_saved_query ( 'saved_query_name' ) ;
```

Show saved query

Show the query for the saved query name.

```
SELECT show_saved_query ( 'saved_query_name' ) ;
```

list_saved_queries

Show all the saved queries in the database.

```
SELECT list_saved_queries ( ) ;
```

2.4. Queries

Queries are used to retrieve data from the current database.

```
query_term ::=

    SELECT
        [ TOP num_rows ]
        [ DISTINCT ]
        select_list
        [ FROM table_ref [, ... ]
            [ WHERE value_expr
        | WHERE HIGH_SELECTIVITY( value_expr ) ]
            [ GROUP BY value_expr [, ... ]
                [ HAVING value_expr ]
            ]
        ]
        [ query_hint ]
    |
    (VALUES ( value_expr [, ... ] ) [, ... ])

select_list ::=

    value_expr [  AS  column_alias ] [, ... ]

column_alias ::= identifier

table_ref ::=

    table_name [  AS  alias [ ( column_alias [, ... ] ) ] ]
    | ( query ) [  AS  alias [ ( column_alias [, ... ] ) ] ]
    | table_ref join_type table_ref
        [ ON value_expr | USING ( join_column [, ... ] ) ]

alias ::= identifier

join_type ::=

    [ INNER ] [ join_hint ] JOIN
    | LEFT [ OUTER ] [ join_hint ] JOIN
    | RIGHT [ OUTER ] [ join_hint ] JOIN
    | CROSS [ join_hint ] JOIN

join_hint ::=

    MERGE | LOOP

order ::=

    value_expr [ ASC | DESC ] [, ...]  [NULLS FIRST | LAST ]
```

```
query_hint ::=

    OPTION ( query_hint_option [, ... ] )
query_hint_option ::=
   SET DisableJoinOptTableA = [ TRUE | FALSE ]
   | SET DisableJoinOptTableB = [ TRUE | FALSE ]
```

See also **WHERE with HIGH_SELECTIVITY**

2.4.1. SELECT lists

TOP is used to retrieve only the first rows from a query.

TOP will be the last operation on the query execution. This means that SQream will limit the results to the end-user after executing the entire statement.

DISTINCT removes duplicate rows.

Value expressions in select lists support aggregate and window functions as well as normal value expressions (see below).

Examples

```
select * from t;

select 1 + a from t;

select a as b from t;

select a+b, c+d from t;

select top 10 col from tbl;

select col from tbl limit 10;

select distinct a,b from t;
```

Column at the **SELECT** list are separated with commas. Columns not separated will be considered as alias: (select a as a1, b as b1 from) can be written as (select a a1, b b1 from)

2.4.2. FROM

FROM is used to specify which tables to read in a query. FROM can either contain table/view names or subqueries.

Examples

```
select * from t;

SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name=table2.column_name;

SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name=table2.column_name;

SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name=table2.column_name;

SELECT *
FROM table1,table2
WHERE table1.column_name=table2.column_name;
```

Join hints can be used to override the query compiler and choose a particular join algorithm. The available algorithms are **LOOP** (corresponding to non-indexed nested loop join algorithm), and **MERGE** (corresponding to sort merge join algorithm).

```
SELECT *
FROM table1
INNER MERGE JOIN table2
ON table1.column_name=table2.column_name;

SELECT *
FROM table1
INNER LOOP JOIN table2
ON table1.column_name=table2.column_name;
```

2.4.3. WHERE

WHERE is used to filter out rows.

Examples

```
SELECT Column1
FROM table1
WHERE column2 <= 1;
```

See also **WHERE with HIGH_SELECTIVITY**

2.4.4. GROUP BY

GROUP BY is used to partition a table so that aggregates can be applied separately to each partition.

Examples

```
select a,sum(b) from t group by a;
```

2.4.5. HAVING

HAVING is used to filter out rows after GROUP BY processing.

Examples

```
select a,sum(b) from t group by a having sum(b) > 5;
```

2.4.6. ORDER BY

ORDER BY is used to order the results.

Examples

```
select * from t order by a asc, b desc;
```

2.4.7. VALUES

VALUES is a way to create a 'literal table value'.

```
values (1,'a'), (2,'b');
```

2.4.8. Set operators

UNION is used to concatenate two queries together. SQream currently supports **UNION ALL**, which doesn't remove duplicate rows.

Examples

```
select * from t
union all
select * from u;
```

2.4.9. WITH Subqueries

The WITH query_name clause allow assigning names to subquery blocks for repeated use in the query.

```
WITH alias_1 AS (query_term)
     [, ...]
SELECT select_list
FROM alias_1
   [ JOIN alias_2 ON join_condition ]
   [ WHERE where_condition ]
```

Examples

```
WITH
    alias_a as (select * from sqream_catalog.databases),
    alias_b as (select * from sqream_catalog.tables)
SELECT a.database_name, b.table_name
FROM alias_a a inner join alias_b b
ON a.database_name=b.database_name;
```

WITH can not refer to a recursive alias (not self-referencing), that contains no 'order by' in its subquery.

2.5. Manual Query Tuning

2.5.1. WHERE with HIGH_SELECTIVITY

WHERE HIGH_SELECTIVITY is used to filter out rows, with a hint optimization. This is best used when the column being filtered out is **not sorted**, and the amount of **rows returned is expected to be small** (good rule of thumb would be less than 40%).

This hint tells the compiler that this WHERE condition is going to filter out more than 60% (for example) of the table rows. It does not affect the query results, but when used correctly can improve query performance.

- This feature is less effective when the condition is on a sorted column, since it will overlap with other optimizations, thereby making it redundant.
- If there's no reason to believe that the WHERE clause is going to filter a majority
of the records, this optimization can be omitted

Examples

```
-- We know LOG_ID=5 is a small amount of values, so we will
-- instruct the compiler about it:
select * from logger where high_selectivity(log_id = 5);
-- We can also add other values:
select * from logger where high_selectivity(log_id = 5) and
high_selectivity(IP='192.168.0.192');
-- Or (alternate syntax):
select * from logger where high_selectivity(log_id = 5 and IP='192.168.0.192');
```

From V2.5 the hint is called HIGH_SELECTIVITY. In lower versions this hint was called LOW_SELECTIVITY.

2.6. Data types

SQream data types to be used in **CREATE TABLE** and **ALTER TABLE**, and in `value_expr`.

```
type_name ::=
    BOOL
    | TINYINT
    | SMALLINT
    | INT / INTEGER
    | BIGINT
    | FLOAT / DOUBLE
    | REAL
    | DATE
    | DATETIME / TIMESTAMP
    | VARCHAR / CHARACTER VARYING
    | NVARCHAR
```

2.6.1. Boolean

Table 23. Boolean data type

Type	Description	Size (not null)	Synonym
BOOL	Boolean type	1 byte	BOOLEAN

Boolean literals can be written as TRUE and FALSE *or* 1 and 0 respectively, but are always displayed as 1 and 0 in the native client.

Examples

```
CREATE TABLE boolean_values (col1 bool);
INSERT INTO boolean_values VALUES ((true),(false));
```

Table 24. Boolean values

Values
1
0

2.6.2. Numeric types

Table 25. Numeric data types

Type	Description	Size (not null)	Synonym	Minimum	Maximum
TINYINT	unsigned integer	1 byte		0	255

Type	Description	Size (not null)	Synonym	Minimum	Maximum
SMALLINT	signed integer	2 bytes		-32,768	32,767
INT	signed integer	4 bytes	INTEGER	-2,147,483,648	2,147,483,647
BIGINT	signed integer	8 bytes		-9,223,372,036,854,775,808	9,223,372,036,854,775,807
REAL	floating point number	4 bytes		-3.40e+38	3.40e+38
FLOAT	floating point number	8 bytes	DOUBLE	-1.79e+308	1.79e+308

To avoid overflow on numeric data types during mathematical operations, it is recommended to cast to a larger data type like BIGINT explicitly.
For example, `SELECT SUM(int_column :: BIGINT) from table;`

2.6.3. Date & datetime

Table 26. Date and datetime data types

Type	Description	Size (not null)	Synonym	Example
DATETIME	Date and time, January 1, 1 CE to December 31, 9999 CE, 1 millisecond precision	8 bytes	TIMESTAMP	'2015-12-31 08:08:00.000'
DATE	Date only, January 1, 1 CE to December 31, 9999 CE	4 bytes		'2015-12-31'

Time zones are not supported.

Milliseconds are stored as 3 digits. If milliseconds are ingested with more than 3 digits, the system will round to 3 digits.

2.6.4. String types

Table 27. String data type

Type	Description	Maximum size (not null)	Synonym
VARCHAR(*n*)	String of ASCII characters at the length of n	*n* bytes	CHARACTER VARYING
NVARCHAR(*n*)	String of UNICODE characters at the length of n	*4*n* bytes	

- NVARCHAR data type supports multiple languages with UTF8 encoding.
- Restriction: Both VARCHAR and NVARCHAR do not support the ASCII character 0 (=NULL).

- VARCHAR is right-padded with spaces. These trailing spaces are ignored when used in functions.
- **NVARCHAR** column **can not be aggregated** or used as a join key between data sets.
- VARCHAR/NVARCHAR column size can not exceed the SQream max row length (10239 bytes).

2.7. Value expressions

Value expressions are used in select lists, **ON** conditions, **WHERE** conditions, **GROUP BY** expressions, **HAVING** conditions and **ORDER BY** expressions.

```
value_expr ::=
    string_literal
    | number_literal
    | NULL | TRUE | FALSE
    | typed_literal
    | value_expr binary_operator value_expr
    | unary_operator value_expr
    | value_expr postfix_unary_operator
    | special_operator
    | extract_operator
    | case_expression
    | conditional_expression
    | ( value_expr )
    | identifier
    | star
    | function_app
    | aggregate_function_app
    | window_function_app
    | cast_operator
```

2.7.1. String literal

`string_literal` is delimited by single quotes ('), and can contain any printable character other than single quote.

1. To include a single quote within a string literal, write two adjacent single quotes, e.g., 'Database''s features'. Note that this is not the same as a double-quote character (")
2. Similarly, to avoid escaping the single quote, use the *dollar quoting* notation (see examples below)

Examples

```
SELECT 'string literal';

SELECT 'string literal ''with something'' quoted' ;
-- this produces the string "string literal 'with something' quoted"

SELECT $$string literal 'with something' quoted$$ ;
-- Same as above
```

2.7.2. Number literal

```
number_literal ::=

    digits
    | digits . [ digits ] [ e [+-] digits ]
    | [ digits ] . digits [ e [+-] digits ]
    | digits e [+-] digits
```

Examples

```
1234

1234.56

12.

.34

123.56e-45
```

2.7.3. Typed literal

```
typed_literal ::=
    type_name string_literal
```

`type_name` is defined above in the type name definition section.

2.7.4. Binary operator

```
binary_operator ::=
    . | + | ^ | * | / | % | + | - | >= | <= | != | <> | ||
    | LIKE | NOT LIKE | RLIKE | NOT RLIKE | < | > | = | OR | AND
```

2.7.5. Unary operator

```
unary_operator ::=
    + | - | NOT
```

2.7.6. Postfix unary operator

```
postfix_unary_operator ::=

    IS NULL | IS NOT NULL
```

2.7.7. Special operator

```
special_operator ::=

    value_expr IN ( value_expr [, ... ] )
    | value_expr NOT IN ( value_expr [, ... ] )
    | value_expr BETWEEN value_expr AND value_expr
    | value_expr NOT BETWEEN value_expr AND value_expr
```

SQream limits the number of 'in list' values to 500. It is highly recommended to avoid a large 'in list' (more than 20) and to use JOIN operation instead:

```
INSERT INTO temp_table VALUES (val1, val2, val3 .... valN);
SELECT .. FROM my_table JOIN temp_table ON
my_table_col=temp_table_in_list_column;
```

To use AND in the middle of a `value_expr` with a BETWEEN operator, enclose the expression in parentheses:

```
expr BETWEEN ( min_expr_with_and ) AND max_expr
```

2.7.8. EXTRACT operator

The extract operator can be used to extract parts of dates/times from date or datetime values.

See also DATEPART in Date/time functions

This operator always returns a float.

```
extract_operator ::=

    EXTRACT ( extract_field FROM value_expr )

extract_field ::=
    YEAR
  | MONTH
  | WEEK
  | DAY
  | DOY
  | HOUR
  | MINUTE
  | SECOND
  | MILLISECONDS
```

Example

```
SELECT EXTRACT(hour FROM '1997-06-02 15:30:00.000');
-- Returns 15.00

SELECT EXTRACT(year FROM '1997-06-02 15:30:00.000');
-- Returns 1997.00
```

Table 28. Example results

extract_field	Syntax	Result
YEAR	EXTRACT(YEAR FROM '1986-06-02 15:31:22.124')	1986.00
MONTH	EXTRACT(MONTH FROM '1986-06-02 15:31:22.124')	6.00
WEEK	EXTRACT(WEEK FROM '1986-06-02 15:31:22.124')	23.00
DAY	EXTRACT(DAY FROM '1986-06-02 15:31:22.124')	2.00
DOY	EXTRACT(DOY FROM '1986-06-02 15:31:22.124')	153.00
HOUR	EXTRACT(HOUR FROM '1986-06-02 15:31:22.124')	15.00
MINUTE	EXTRACT(MINUTE FROM '1986-06-02 15:31:22.124')	31.00
SECOND	EXTRACT(SECOND FROM '1986-06-02 15:31:22.124')	22.124
MILLISECONDS	EXTRACT(MILLISECONDS FROM '1986-06-02 15:31:22.124')	22124.00

2.7.9. CASE expression

```
case_expression ::=

    searched_case | simple_case

searched_case ::=

    CASE WHEN value_expr THEN value_expr
        [WHEN ...]
        [ELSE value_expr]
    END

simple_case ::=

    CASE value_expr
        WHEN value_expr THEN value_expr
        [WHEN ...]
        [ELSE value_expr]
    END
```

_searched_case_ works as follows:

- Each WHEN _value_expr_ is checked in order, the value of the CASE expression is the value of the THEN `_value_expr_` then for the first WHEN branch which evaluates to true;
- If no WHEN branches evaluate to true, then the value is the value of the ELSE expression, or if there is no ELSE, then the value is NULL.

The **`simple_case`** style is shorthand:

```
CASE v0
    WHEN v1 THEN r1
    WHEN v2 THEN r2
    ...
    ELSE e
END

-- Is identical to:

CASE
    WHEN v0 = v1 THEN r1
    WHEN v0 = v2 THEN r2
    ...
    ELSE e
END
```

2.7.10. Identifier Rules

Identifiers are typically used as database objects names, such as databases, tables, views or columns. In addition, identifiers can be used to change the resulting column name (column alias)

with SELECT.

identifier is

- Unquoted identifier:
  - Length can be up to 128 chars.
  - Must begin with any ASCII (A-Z) character (uppercase or lowercase) or underscore (_).
  - Subsequent characters can be letters, underscores or digits.
  - Uppercase characters in unquoted identifiers are converted to lowercase, and kept in the SQream catalog as **lowercase**.
- Quoted identifier:
  - Length can be up to 128 chars.
  - Wrapped with double quotes (").
  - May contain any printable character, except for **@**, **$** or **"** (double quotes).
  - Quoted identifiers are kept in the SQream catalog as **case sensitive**.

Examples

```
CREATE TABLE "Customers" (
    ID int,
    "Name" varchar(50)
);

SELECT * from "Customers";

CREATE TABLE customers (
    ID int,
    name varchar(50)
);

SELECT col1 AS "My favourite column", col2 as "I'm not really sure I like this column"
FROM t;
```

2.7.11. Aggregate function app

```
aggregate_function_app ::=

    agg_name ( [ value_expr [, ... ] ] )

    | agg_name ( [ DISTINCT ] [ value_expr [, ... ] ] )

agg_name ::= identifier
```

2.7.12. Window function app

```
window_function_app ::=

    window_fn_name ( [ value_expr [, ... ] ] )
        OVER ( [ value_expr [, ... ] ]
               [ PARTITION BY value_expr [, ... ] ]
               [ ORDER BY value_expr [ ASC | DESC] [, ... ] ] )

window_fn_name ::= identifier
```

Examples

See the Window functions segment for examples.

2.7.13. Operator precedences

This table lists the operators in decreasing order of precedence. We recommend using parentheses rather than relying on precedences in anything other than trivial expressions.

Table 29. Operator precedences

Operator	Associativity
.	left
+ - (unary)	
^	left
* / %	left
+ - (binary)	left
||	right
BETWEEN, IN, LIKE, RLIKE	
< > = <= >= <> !=	
IS NULL, IS NOT NULL	
NOT	
AND	left
OR	left

The **NOT** variations: **NOT BETWEEN**, **NOT IN**, **NOT LIKE**, **NOT RLIKE** have the same precedence as their non-**NOT** variations.

2.8. Functions and Operators

2.8.1. Operators

Logical

Table 30. Logical operators

Name	Type	Description
`and`	**(bool, bool) returns bool**	logical and
`or`	**(bool, bool) returns bool**	logical or
`not`	**(bool) returns bool**	logical not

AND

```
and (bool, bool) returns bool
```

Logical and.

Examples

```
TRUE AND FALSE
```

OR

```
or (bool, bool) returns bool
```

Logical or.

Examples

```
a OR b
```

NOT

```
not (bool) returns bool
```

Logical not.

Examples

```
NOT TRUE
```

Comparison

Table 31. Comparison operators

Name	Type	Description
`< > <= >= == !=`	(***any***, ***any***) **returns bool**	regular binary comparison operations
`between`	(*exp* ***any***, *min* ***any***, *max* ***any***) **returns bool**	is *exp* between *min* and *max* inclusive
`not between`	(*exp* ***any***, *min* ***any***, *max* ***any***) **returns bool**	inverse of between
`is null`	(***any***) **returns bool**	argument is null
`is not null`	(***any***) **returns bool**	argument isn't null
`in`	(***any*** [, ...]) **returns bool**	list membership

any is any type.

Binary comparison operators

```
binary_comparison_operator (any, any ) returns bool

binary_comparison_operator is one of < > <= >= == !=
```

Regular binary comparison operators. The two input types should be the same, but the system will insert valid implicit casts in many cases (see the cast section).

BETWEEN, NOT BETWEEN

`exp between min and max` is shorthand for `exp >= min and exp <= max`.

`exp not between min and max` is shorthand for `not (exp between min and max)`.

```
between (exp any, min any, max any) returns bool
```

Examples

```
a between b and c
```

IS NULL, IS NOT NULL

IS NULL checks if the argument is null.

IS NOT NULL checks if the argument isn't null.

Testing for null using `exp = NULL` will not work to check if a value is null, and testing for not null using `exp <> NULL` will not work to check if a value is not null. You have to use the **IS NULL** and **IS NOT NULL** operators.

Examples

```
(1 + null) is null

(a * b) is not null
```

IN

IN tests for membership in a list.

IN subqueries are not supported.

Examples

```
a in (1,3,5,7,11)
```

2.8.2. Bitwise Operators

A bitwise operation operates on one or more bit patterns or binary numerals at the level of their individual bits. It is a fast, simple action directly supported by the processor, and is used to manipulate values for comparisons and calculations.

Table 32. Bitwise Operators

Symbol	Operator	Short Description	Example	Returns
`&`	bitwise AND	Result is true only if both operands are true.	`101 & 110`	`100`
			`5 & 6`	`4`
`|`	bitwise inclusive OR	Result is true if any of the operands is true.	`101 | 110`	`111`
			`5 & 6`	`7`
`xor`	bitwise XOR (eXclusive OR)	Result is true only if one of its operands is true.	`xor(101,011)`	`110`
			`xor(5,3)`	`6`
`~`	bitwise NOT	Provides the bitwise complement of an operand by inverting its value such that all zeros are turned into ones and all ones are turned to zeros.	`~0100`	`-101`
			`~4`	`-5`

Symbol	Operator	Short Description	Example	Returns
>>	Shift right	Moves the bits the number of positions specified by the second operand in the right direction.	8 >>2	2
<<	Shift left	Moves the bits the number of positions specified by the second operand in the left direction.	3 <<2	12

Bitwise operators are supported for all Integer data types: TINYINT, SMALLINT, INT, and BIGINT.

2.8.3. Mathematical functions and operators

When performing mathematical operations on an integer, SQream will round up the results and return an integer. In order to return a decimal number, make sure to use real or float with the operation itself or cast the integer to real/float. For example: (100/14) will result in 7 while (100.0/14) or (100/14.0) will result in 7.1429.

SQRT

SQRT - Square root of the argument to the function

```
SELECT SQRT (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

ABS

ABS - |x|- Absolute (positive) value of the argument

```
SELECT ABS (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting) Returns Float/Double

ROUND

ROUND - Rounds the number to the nearest precision

```
SELECT ROUND (cfloat,2) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting) Int32T Precision (number of places after the decimal point)

Returns Float/Double

ASIN

ASIN sin-1 (x) – Arcsine (angle in radians whose sine is the argument of the function)

```
SELECT ASIN (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting) Returns Float/Double

ATAN

ATAN tg-1 (x)- Arctangent (angle in radians whose tangent is the argument of the function)

```
SELECT atan(cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

ATN2 – Arctangent

ATN2 (angle in radians between positive X-axis and the ray from the origin where x and y are the first and second arguments)

```
SELECT ATN2 (cfloat,cfloat2) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting) Float/Double (All other numbers available via implicit casting)

Returns Float/Double

COS

COS - cos x- - trigonometric cosine of the angle in radians

```
SELECT COS (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

COT

COT - cot x - Cotangent - trigonometric cotangent of the angle in radians

```
SELECT COT (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

CEILING or CEIL

CEILING or **CEIL** - Return the smallest integer value that is greater than or equal to the argument

```
SELECT CEILING (cfloat) FROM table;
SELECT CEIL (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

LOG10

LOG10 - log10 x- base 10 logarithm of the argument

```
SELECT LOG10 (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

LOG

LOG - ln x - Natural base logarithm (ln or loge) of the argument

```
SELECT LOG (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

LOG (base-y)

LOG base=y - Base-y logarithm of the x parameter, where x,y are the arguments

```
SELECT LOG (cfloat,cint) FROM table;
SELECT LOG (cfloat,8) FROM table;
```

Parameters Float/Double argument (all other numbers available via implicit casting) Integer Base

Returns Float/Double

MOD

MOD - returns the remainder from a division of argument#1 by argument #2.

```
SELECT MOD (cint,cint) FROM table;
```

Parameters Integer

Returns Integer

FLOOR

FLOOR - Floor returns the smallest integer to the argument

```
SELECT FLOOR (cfloat) FROM table
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

SIN

SIN - trigonometric sine of the angle in radians

```
SELECT SIN (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

SQUARE

SQUARE - x2 - the square of the argument

```
SELECT SQUARE (cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

TAN

TAN - Tangent of the argument

```
SELECT tan(cfloat) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting)

Returns Float/Double

PI

PI - mathematical constant

```
SELECT PI () FROM table;
```

Parameters none

Returns Double value of Pi to 10 digits after the decimal point

POWER

POWER - perform a power of one value over the other (x raised to the power of y)

```
SELECT POWER (cfloat,cfloat2) FROM table;
```

Parameters Float/Double (All other numbers available via implicit casting) Float/Double (All other numbers available via implicit casting)

Returns Float/Double

TO_HEX

TO_HEX - Converts an integer to its base-16 string representation

```
SELECT TO_HEX (x) FROM table;
```

Parameters Int/Long parameter

Returns VarChar - Base-16 string representation

TRUNC

TRUNC - TRUNC (float/double) returns the argument truncated to a whole number.

```
SELECT TRUNC (cfloat) FROM table;
```

Parameters Float/Double

Returns Float/Double (same data type as parameter)

2.8.4. Check functions

COALESCE

COALESCE - Returns the first non-null value of the list of arguments to the function.

```
SELECT COALESCE (cfloat, cfloat ) FROM table;
SELECT COALESCE (varchar_column, varchar_column, var_char_column ) FROM table;
SELECT COALESCE (date_column, datetime_column, datetime_column, date_column) FROM
table;
```

Parameters can be of any data type, but must be of same one.

Returns a single value of the same data type as parameter.

2.8.5. Conversion functions

These functions support data type casting and conversion from one data type to another. In addition to explicit conversions, the systems performs implicit conversions and casts different data types in an expressions automatically according to the rules (see table below).

CAST

This function converts an expression from one data type to another.

```
cast_operator ::=

    CAST ( value_expr AS typename )
    value_expr :: typename
```

Examples

```
SELECT CAST(1234 as FLOAT);

SELECT (1234 :: FLOAT);

SELECT CAST('1997-06-02 16:53:00.333' as DATE);
-- returns: date '1997-06-02'

SELECT CAST(1234.56789 as VARCHAR(10));
-- returns: varchar 1234.5679
```

Table 33. Supported conversions, both implicit and explicit

From type	To Type	Context	Comments
int	bigint	implicit	
	bool	explicit	
	datetime	none	
	date	none	
	varchar	explicit	
bigint	int	explicit	
	bool	explicit	
	datetime	none	
	date	none	
	varchar	explicit	
float/real	int	implicit	Truncates the value to Integer.
	bigint	implicit	Truncates the value to BIGINT.
	bool	explicit	
	datetime	none	
	date	none	
	varchar	explicit	Rounds the value to 4 digits after the decimal point and then converts it to a character. See example above.
bool	int	implicit	
	bigint	implicit	
	datetime	none	
	date	none	

From type	To Type	Context	Comments
date	int	none	
	bigint	none	
	bool	none	
	datetime	implicit	
	varchar	explicit	
datetime**	int	none	
	bigint	none	
	bool	none	
	date	implicit	Truncates the time part.
	varchar	explicit	
varchar***	int	explicit	
	bigint	explicit	
	datetime	explicit	
	date	explicit	
	varchar	explicit	

** There are conversion functions to convert part of a date or datetime to integer, or to convert the whole date/datetime to unix timestamps but these are not considered casts or available using the cast syntax

*** String literals without an explicit type are considered unknown type (and not varchar) and will implicitly cast to any type.

UTF8_TO_ASCII

UTF8_TO_ASCII - Converts an ascii-only nvarchar column to a varchar. To complete this function, use the function **IS_ASCII** to verify the column data indeed contains only ascii characters.

```
SELECT UTF8_TO_ASCII(x) FROM table where IS_ASCII(x);
SELECT UTF8_TO_ASCII(x) FROM table where IS_ASCII(x) group by 1;
```

Parameters NVarChar

Returns VarChar

IS_ASCII will only work on NOT NULL columns. If your column definition allows NULL, use ISNULL to convert it: is_ascii(isnull(x,''))

CRC64

CRC64 - Converts values of a varchar or nvarchar column to bigint (hash key) using the crc64 hash function.

```
SELECT CRC64(text_column) FROM table;
```

Parameters NVarChar or Varchar

Returns Bigint

Assignment resolution

This is a different kind of implicit cast which applies when you are inserting one type of expression into a column with a different type. The casting rules are essentially the same as the implicit casting in value expressions for the equals operator.

Expression set type resolution

The last system of implicit casts is used to resolve the type of a collection of expressions with different types which should resolve to a single compatible type.

This is used in:

- case then expressions
- in list values

It is based on the implicit casting rules for the equals operator.

2.8.6. String functions and operators on VARCHAR

VARCHAR data type is intended to ASCII character set. All function parameters must be either NVARCHAR or VARCHAR. SQream does not support casting between this two types.

LOWER

LOWER - Converts a string to lowercase

```
SELECT LOWER (varchar_column) FROM table;
```

UPPER

UPPER - Converts a string to uppercase

```
SELECT UPPER (varchar_column) FROM table;
```

LEN

LEN - Returns the length of a varchar.

```
SELECT LEN (varchar_column) FROM table;
```

Remarks Trailing whitespace on the right are ignored: LEN on 'abc' and 'abc ' will both return 3.

LIKE

LIKE - Checks if a string matches a LIKE pattern. Also used as **NOT LIKE**.

```
SELECT * FROM table WHERE varchar_column LIKE '%string%';
SELECT * FROM table WHERE varchar_column NOT LIKE '%string%';
```

Only literal patterns are supported. Column references are not supported as a pattern.

RLIKE

RLIKE - Checks if a string matches a regex pattern

```
SELECT * FROM table WHERE varchar_column RLIKE '[0-9]+$';
```

Does not work on NVARCHAR columns.

SUBSTRING

SUBSTRING - Returns a specific substring of a string

```
SELECT SUBSTRING (varchar_column,start,length) FROM table;
```

Parameters

1. col_ref - (varchar) the string column to substring
2. start_position - (int) the starting point of the substring, while the value 1 represent the first character.
3. length - (int) the length of the substring to take

Returns

- String of the resulting substring operation

If `start ⇐1`, then the substring begins from the first character but the length is reduced.
`substring('abc',1,2) == 'ab'`
`substring('abc',0,2) == substring('abc',1,1) == 'a'`

REGEXP_COUNT

REGEXP_COUNT - Counts regex matches in string. For example, the pattern '[1-9]' appears once in '01' and twice in '12'.

```
SELECT REGEXP_COUNT (col,'[0-9]',2) FROM table;
```

Parameters

1. col_ref - (varchar) the string column to match
2. pattern - (string literal) the regex (literal only)
3. start_pos - starting location (Optional. When unset, default is 1)

Returns

- Amount of matches of the regex pattern in the string (int)

REGEXP_INSTR

REGEXP_INSTR - Matches regex and returns the position in a string of the n-th occurence.

```
SELECT REGEXP_INSTR (varchar_column,'[0-9]') FROM table;
SELECT REGEXP_INSTR (varchar_column,'[0-9]',2) FROM table;
SELECT REGEXP_INSTR (varchar_column,'[0-9]',2,2) FROM table;
SELECT REGEXP_INSTR (col,'[0-9]',2,2,1) FROM table;
```

Parameters

1. col_ref - (varchar) the string column to match
2. pattern - (string literal) the regex (literal only)
3. Start position - (int) - Optional. When unset, default is 1
4. Occurence number - (int) - which occurrence of the pattern - optional. When unset, default is 1 - first occurence
5. Match start/end position - (int) - 0 for match's start position, 1 for its end - optional. When unset, default is 0

Returns

- Position of the first occurence of the regex pattern (int)

REGEXP_SUBSTR

REGEXP_SUBSTR - matches regex and returns it.

```
SELECT REGEXP_SUBSTR (varchar_column,'[0-9]') FROM table;
SELECT REGEXP_SUBSTR (varchar_column,'[0-9]',2) FROM table;
SELECT REGEXP_SUBSTR (varchar_column,'[0-9]',2,2) FROM table;
```

Parameters

1. col_ref - the string column to match
2. VarChar - the regex (literal only)
3. Int - starting location (Optional. When unset, default is 1)
4. Int - which occurrence of the pattern (Optional. When unset, default is 1)

Returns

- String of the matched column

ISPREFIXOF

ISPREFIXOF - Checks if one string is a prefix of the other.

```
SELECT ISPREFIXOF (x,y) FROM table
```

Remarks Internal function. "isprefix(x,y)" is equivalent to "y LIKE x + '%' ", but more efficient

Concatenation (||)

|| - String concatenation - concatenates two string values

```
SELECT fname || '_' || lname FROM customers;
```

CHARINDEX

CHARINDEX - Returns the position of a subexpression in an expression

```
SELECT CHARINDEX (y,x,1) FROM table;
```

Parameters VarChar - the subexpression. Either a scalar or a column VarChar - the expression Int (optional) - starts the search from this index

Returns Int - the position of the subexpression in the expression or 0 if it wasn't found

PATINDEX

PATINDEX - Returns the position of a pattern in an expression

```
SELECT PATINDEX ('%[0-9]%',x) FROM table;
```

Parameters VarChar (literal) - the subexpression VarChar - the expression

Returns Int - the position of the first match of the pattern in the expression or 0 if there's no match

LTRIM

LTRIM - Trims leading whitespace from the left side of the string

```
SELECT LTRIM (x) FROM table;
```

Parameters VarChar

Returns VarChar

REVERSE

REVERSE - Reverses a string

```
SELECT REVERSE (x) FROM table;
```

Parameters VarChar

Returns VarChar

RTRIM

RTRIM - Trims trailing whitespace from the right side of the string

```
SELECT RTRIM (x) FROM table;
```

Parameters VarChar

Returns VarChar

TRIM

TRIM - Removes any whitespace from the beginning and end of a string (leading and trailing)

```
SELECT TRIM (x) FROM table;
```

Parameters VarChar

Returns VarChar

2.8.7. String functions and operators on NVARCHAR

NVARCHAR data type is intended to support multiple languages with UTF8 encoding. All function

parameters must be either NVARCHAR or VARCHAR. SQream does not support casting between this two types.

LOWER

LOWER - Converts ASCII string to lowercase. Note that in non-ascii characters the function will return the original column data.

```
SELECT LOWER (nvarchar_column) FROM table;
```

UPPER

UPPER - Converts ASCII string to uppercase. Note that in non-ascii characters the function will return the original column data.

```
SELECT UPPER (nvarchar_column) FROM table;
```

LEN

LEN - Returns the length of a nvarchar while trimming whitespaces.

```
SELECT LEN (nvarchar_column) FROM table;
```

CHAR_LENGTH

CHAR_LENGTH Returns the length of a nvarchar without trimming whitespaces.

```
SELECT CHAR_LENGTH (nvarchar_column) FROM table;
```

LIKE

LIKE - Checks if a string matches a LIKE pattern. For NVARCHAR columns the following options for like function exists: '%string', 'string%', '%string%'. Can be used also as **NOT LIKE**.

```
SELECT * FROM table WHERE nvarchar_column LIKE '%string';
SELECT * FROM table WHERE nvarchar_column NOT LIKE '%string%';
```

Currently SQream only support literal pattern.

SUBSTRING

SUBSTRING - Returns a specific substring of a string

```
SELECT SUBSTRING (nvarchar_column,start,length) FROM table;
```

Parameters start - the starting point of the substring, while the value 1 represent the first character. length - the length of the substring

If `start ⇐1`, then the substring begins from the first character but the length is reduced.
`substring('abc',1,2) == 'ab'`
`substring('abc',0,2) == substring('abc',1,1) == 'a'`

Concatenation (||)

|| - String columns concatenation - concatenates two string column values

```
SELECT fname || lname FROM customers;
```

CHARINDEX

CHARINDEX - Searches an expression in a string nvarchar column and returns its starting position if found.

```
SELECT CHARINDEX ('text to look',col_x) FROM table;
SELECT CHARINDEX ('text to look',col_x,10) FROM table;
```

Parameters NVarChar - the subexpression as a scalar NVarChar - the column name Int (optional) - starts the search from this index

Returns Int - the position of the subexpression in the expression or 0 if it wasn't found

LEFT

LEFT - Returns the left part of a character string with the specified number of characters.

```
SELECT LEFT (x,3) FROM table;
```

Parameters NVarChar

Returns NVarChar

REPLACE

SQream supports a limited version of the **REPLACE()** function on NVARCHAR. Replaces a sub-string with another sub-string of the same size.

Limitations:

- At this stage the REPLACE() function is only supported for replacing a sub-string with another sub-string of the same size. Meaning, the 'from' and 'to' arguments must be string literals of the same length.
- Works for NVARCHAR only.

```
SELECT REPLACE(x,'a','b') from table;
SELECT REPLACE(x,'1','*') from table;
SELECT REPLACE(x,'123','321') from table;
```

Parameters NVarChar

Returns NVarChar

REVERSE

REVERSE - Reverses a string

```
SELECT REVERSE (x) FROM table;
```

Parameters NVarChar

Returns NVarChar

RIGHT

RIGHT - Returns the right part of a character string with the specified number of characters.

```
SELECT RIGHT (x,3) FROM table;
```

Parameters NVarChar

Returns NVarChar

OCTET_LENGTH

OCTET_LENGTH - Returns the length in bytes (octets) of the nvarchar column value (being the number of bytes in binary string).

```
SELECT OCTET_LENGTH (x) FROM table;
```

In some of the NVARCHAR functions, SQream does not support the use of literals. In others, the use in literals will have to be explicitly wrapped in 'cast(... as nvarchar)' function.

2.8.8. Pattern matching syntax

Table 34. Pattern matching syntax

Syntax	Description
`%`	match zero or more characters
`_`	match exactly one character
`[A-Z]`	match any character between A and Z inclusive
`[^A-Z]`	match any character not between A and Z
`[abcde]`	match any one of a b c d and e
`[^abcde]`	match any character that isn't one of a b c d and e
`[abcC-F]`	match a b c or between C and F

2.8.9. Regular Expression Pattern Matching Syntax

Table 35. Regular expression pattern matching syntax

Syntax	Description
`^`	Match the beginning of a string
`$`	Match the end of a string
`.`	Match any character (including carriage return and newline)
`*`	Match the previous pattern zero or more times
`+`	Match the previous pattern zero or more times
`?`	Match the previous pattern zero or one times
`de|abc`	Match either 'de' or 'abc'
`(abc)*`	Match zero or more instances of the sequence abc
`{2}`	Match the previous pattern exactly two times
`{2,4}`	Match the previous pattern between two and four times
`[a-dX], [^a-dX]`	Matches any character that is (or is not, if ^ is used) either a, b, c, d or X. A - character between two other characters forms a range that matches all characters from the first character to the second. For example, [0-9] matches any decimal digit. To include a literal] character, it must immediately follow the opening bracket [. To include a literal - character, it must be written first or last. Any character that does not have a defined special meaning inside a [] pair matches only itself.

2.8.10. Date and Datetime

Table 36. Date and Datetime functions

Name	Syntax	Description	Return data type
getdate	`getdate()`	Returns the current date and time. Same as current_timestamp().	Datetime

Name	Syntax	Description	Return data type
current_timestamp	`current_timestamp()`	Returns the current date and time. Same as getdate().	Datetime
current_date	`current_date()`	Returns the current date. Same as curdate().	Date
curdate	`curdate()`	Returns the current date. same as current_date().	Date
trunc	`trunc(datetime_column)`	Sets to timepart to 00:00:00 (midnight).	Datetime
trunc	`trunc(datetime_column,interval)`	Rounds the specified date to beginning of year, month, day, minute. etc., based on the specified interval (second argument). See examples below.	Datetime

datepart	`datepart(interval,date_column)`
	
	`datepart(interval,datetime_column)`
		Returns the number of a the specified datepart of a date or datetime.	Integer

datediff	`datediff(interval,*startdate*,*enddate*)`	Returns the difference between two dates based on the specified interval. Same as function EXTRACT.	Integer

dateadd	`dateadd(interval,number,date_column)`
	
	`dateadd(interval,number,datetime_column)`
		Adds a time/date interval to a date.	Datetime

to_unixts	`to_unixts(datetime_column)`	Converts to unix timestamp, seconds since epoch.	Bigint
to_unixtsms	`to_unixtsms(datetime_column)`	Converts to unix timestamp, milliseconds since epoch.	Bigint
from_unixts	`from_unixts(bigint_column)`	Converts unix timestamp, seconds since epoch.	Datetime
from_unixtsms	`from_unixtsms(bigint_column)`	Converts unix timestamp, milliseconds since epoch.	Datetime

eomonth	`eomonth(datetime_column)`
	
	`eomonth(date_column)`
		Returns the last day of the month (end of month).	Datetime / Date

See also EXTRACT

Table 37. Interval Options

Interval	Shorthand aliases
`year`	`yyyy, yy`
`quarter`	`qq, q`
`month`	`mm, m`

Interval	Shorthand aliases
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw
hour	hh
minute	n
second	ss, s
millisecond	ms

Examples

```
select datepart(q,date_column_name) from table_a;
select datepart(dd,date_column_name) from table_a;

select dateadd(dd,1,date_column_name) from table_a;
select dateadd(mm,-1,date_column_name) from table_a;

select dateadd(mm,1,getdate());

select dateadd(dd,1,date_column_name) from table_a;
select dateadd(yy,-1,date_column_name) from table_a;

select datediff(day,date_column_a,date_column_b)  from table_a;
select datediff(hour,'2016-01-01',date_column_b)  from table_a;
select datediff(q,'2016-01-01 13:00:00',date_column_b) from table_a;
```

Examples for TRUNC(date/datetime,interval)

```
select xdatetime,trunc(xdatetime, month) from table;
-- Returns the first day of the month:
xdatetime, trunc
2018-11-04 13:21:20.496, 2018-11-01 00:00:00.000

select trx_datetime,trunc(trx_datetime, minute) from table;
-- Sets the seconds and milliseconds to Zero. Returns the first second of the minute.
trx_datetime, trunc
2018-11-04 13:21:20.496, 2018-11-04 13:21:00.000
```

2.8.11. Geospatial

Point

Points are represented as longitude and latitude columns. Example:

```
create table point (
       longitude float not null,
       latitude float not null
);
```

Polygon

Polygons are N number of points:

```
create table polygon (
    long1 float not null,
    lat1 float not null,

    ...

    long5 float not null,
    lat5 float not null,
);
```

Polyline

A polyline is a collection of line segments, and contains up to twenty points. We represent it as twenty points plus a count column which indicates how many points are actually used in the given row.

```
create table polyline (
    num_of_points int not null,
    long1 float not null,
    lat1 float not null,

    ...

    long20 float not null,
    lat20 float not null
);
```

Table 38. Geospatial functions

Name	Type	Description
point_in_polygon	(_point_long_ *float*, _point_lat_ *float*, _poly_long1_ *float*, _poly_lat1_ *float*, + + ... + + _poly_long5_ *float*, _poly_lat5_ *float*) *returns* *bool*	point inside polygon

Name	Type	Description
line_crosses_polygon	(_number_of_points__ *int*, _polyline_long1_ *float*, _polyline_lat1_ *float*, + + ... + + _polyline_long20_ *float*, _polyline_lat20_ *float*, + _poly_long1_ *float*, _poly_lat1_ *float*, + + ... + + _poly_long5_ *float*, _poly_lat5_ *float*) *returns* *bool*	line crosses polygon

POINT_IN_POLYGON

Returns true if the point is inside the polygon.

Limitations: the point arguments cannot be literals. The polygon arguments can either be all columns or all literals.

LINE_CROSSES_POLYGON

Returns true if the line crosses the polygon.

Limitations: the polyline arguments cannot be literals. The polygon arguments can either be all columns or all literals.

2.8.12. Aggregate functions

Table 39. Aggregate functions

Name	Syntax	Return type	Description
`avg`	`avg(anynumber)`	`float`	average
`count`	`count(any)`	`int`	count
`max`	`max(any)`	`any`	maximum
`min`	`min(any)`	`any`	minimum
`sum`	`sum(anynumber)`	`anynumber`	sum
`stddev`	`stddev(anynumber)`	`float`	standard deviation

`any` can be any type as defined in Data types.
`anynumber` is any numeric type, as defined in Numeric Types

Mathematical operations on integer types may perform rounding. For precise results as a decimal number, a cast is recommended:
For example, `SELECT AVG(int_column :: FLOAT) from table;`

2.8.13. Window functions

Table 40. Window functions

Name
`rank()`
`row_number()`
`min()`
`max()`
`sum()`

Window functions restrictions and limitations

- Window functions cannot be used when the select statement contains any nvarchar columns.
- Window functions expressions can be used only in a select list.
- Window functions cannot be nested (i.e. contain other window functions).
- Window functions can be used only on simple queries, with no group by or sort operations. To bypass this limitation, use the needed operation in a subquery, and the window function at the external query. for example, for 'group by', use the following: select sum(col1) over (partition by col2) from (select count(*) as col1,col2 from my_table group by 1,2);

Examples

```
select col_a,col_c, rank() over ( partition by col_c order by col_c)  from my_table;
select sum(col_a) over ( partition by col_c order by col_c)  from my_table;
select sum(col1) over (partition by col2) from (select count(*) as col1,col2 from
my_table group by 1,2);
select col_a,col_c, row_number() over ( partition by col_c order by col_c)  from
my_table;
```

2.9. User Defined Functions

SQream supports user defined functions written in Python. Customers can use this capability to:

- Generate their own functions to run in SQL commands as a row level function.
- Run Python code from within SQream DB as a utility function (for example: send email, update external logs, activate external libraries etc.)

2.9.1. Create User Defined Functions

```
create_user_defined_function_statement ::=

CREATE [OR REPLACE] FUNCTION function_name (argument-list)
RETURNS return-type
AS $$
Python function body
$$ LANGUAGE python;
```

- SQream requires using Python 3.6.7.
- The PYTHONPATH environment parameter in sqreamd owner should be pointing to the location of the imported python scripts. In a multi node cluster, the location should be the shared file system.

Example 1

```
-- Create a function to calculate distance based on existing data:
CREATE OR REPLACE FUNCTION py_distance (x1 float, y1 float, x2 float, y2 float)
RETURNS float as $$
import math
if y1 < x1:
    return 0.0
else:
    return math.sqrt((y2 - y1) ** 2 + (x2 - x1) ** 2)
$$ LANGUAGE PYTHON;

-- Usage:
SELECT city_name, current_location_name, py_distance(x1,y1,x2,y2) from table1;
```

Example 2

```
-- Create a function that activates an external python script (writefile.py):
CREATE or replace function write_file_to_os() RETURNS int  as $$
import sys
sys.path.append("/home/galit/pythonpath")
import writefile as f
f.main()
return 1
$$ LANGUAGE PYTHON;

-- Usage:
select write_file_to_os();
```

2.9.2. Drop User Defined Functions

```
drop_user_defined_function_statement ::=

DROP FUNCTION [IF EXISTS] function_name();
```

Examples

```
-- drop the user defined function 'py_distance' if it exists:
DROP FUNCTION IF EXISTS py_distance();
```

2.9.3. DDL for User Defined Function

Use the utility function **get_function_ddl()** to generate the DDL for a specified UDF.

Syntax

```
select get_function_ddl('user_function_name');
```

Example

```
select get_function_ddl('fn_full_name');
-- returns:
create function "fn_full_name_new" (fn varchar, ln varchar) returns varchar(100) as
$$return fn+" "+ln $$ language python volatile;
```

Copyright