



SQL Reference Guide

SQream Technologies

Version 3.3

Copyright © 2010-2019. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchant- ability or fitness for a particular purpose.

We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document.

This document may not be reproduced in any form, for any purpose, without our prior written permission.

Table of Contents

Table of Contents	3
SQream DB SQL Reference Guide	7
Overview	7
Data Definition Language	8
Databases	8
CREATE DATABASE	8
DROP DATABASE	8
Schemas	8
CREATE SCHEMA	8
DROP SCHEMA	9
ALTER DEFAULT SCHEMA	9
Tables	9
CREATE TABLE	10
Constraints and defaults	10
Compression types	11
Create Or Replace Table	12
ALTER TABLE	12
Rename table	13
Rename column	13
Add column	13
Drop column	14
DROP TABLE	14
Drop Table If Exists	14
External Tables	15
CREATE EXTERNAL TABLE	15
CONVERSION TABLE FOR DATATYPES OF EXTERNAL TABLES OVER PARQUET	17
Views	17
CREATE VIEW	17

DROP VIEW	18
RECOMPILE VIEW	18
Users/Roles	18
CREATE ROLE	18
DROP ROLE	19
ALTER ROLE	19
Database Roles and Permissions	19
Roles at a glance	19
Roles and users	20
Roles and object permissions	20
Roles and schema	20
Superuser	20
PUBLIC Role	21
GRANT Permissions	21
REVOKE Permissions	22
Locks	23
Generating the DDL	23
Data Manipulation Language	25
INSERT	25
COPY FROM (bulk import)	25
COPY TO (bulk export)	29
TRUNCATE TABLE	31
DELETE	31
Operational Commands	34
Saved queries	34
Create saved query	34
Execute saved query	34
Drop saved query	34
Show saved query	34
List saved queries	34
Queries	36

Value expressions	38
JOIN	39
WITH Subqueries	39
Manual Query Tuning	41
WHERE with HIGH_SELECTIVITY	41
Data types	42
Boolean	42
Numeric types	43
Date & datetime	43
String types	44
Value expressions	45
Special operator	47
EXTRACT operator	47
CASE expression	49
Identifier Rules	50
Operator precedences	50
Functions and Operators	52
Operators	52
Logical	52
Comparison	52
Bitwise Operators	53
Mathematical functions and operators	55
Check functions	58
COALESCE	58
Conversion functions	58
CAST	59
UTF8_TO_ASCII	60
CRC64	60
Assignment resolution	60
Expression set type resolution	60
String functions and operators on VARCHAR	61

String functions and operators on NVARCHAR	64
Pattern matching syntax	67
Regular Expression Pattern Matching Syntax	67
Date and Datetime	68
Geospatial	71
Aggregate functions	72
Window functions	73
User Defined Functions	74
Create User Defined Functions	74
Drop User Defined Functions	75
DDL for User Defined Function	75

SQream DB SQL Reference Guide

Version 3.3

Overview

Welcome to the SQream DB SQL Reference. This guide provides an overview of SQream DB's SQL syntax.

This guide contains some basic examples of syntax, limits, usage, and the system catalog.

There is also a reference of the data types, functions, statements, and query syntax.

This document assumes that you are familiar with the some SQL, and tries to relate to common best-practices where possible.

TIP:

You can use the SEARCH bar to find relevant function names and concepts.

NOTE:

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

Data Definition Language

Databases

The following database commands are available:

- CREATE DATABASE
- DROP DATABASE

CREATE DATABASE

Creates 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

Deletes 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;
```

Schemas

The following schema commands are available:

- CREATE SCHEMA
- DROP SCHEMA
- ALTER DEFAULT SCHEMA

CREATE SCHEMA

Creates a new schema in the current database.


```
create_schema_statement ::=  
  
    CREATE SCHEMA schema_name ;  
  
schema_name ::= identifier
```

NOTE:

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

Deletes an empty schema in the current database.

```
drop_schema_statement ::=  
  
    DROP SCHEMA schema_name ;
```

NOTE:

The PUBLIC schema can not be dropped.

Examples:

```
DROP   SCHEMA   my_schema;
```

ALTER DEFAULT SCHEMA

Changes the default schema for a specific role. Use this command 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;
```

Tables

The following table commands are available:

- CREATE TABLE
 - Create Or Replace Table
- ALTER TABLE
 - Rename table
 - Rename column
 - Add column
 - Drop column
- DROP TABLE
 - Drop Table If Exists

CREATE TABLE

Creates a new table in the current database under a specific schema.

NOTE:

- 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.

WARNING:

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.

NOTE:

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

Does one of the following

- If the table does not exist: creates a new table
- If the table does exist: drops and creates the table with the 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;
```

WARNING:

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

ALTER TABLE

Alters the structure of an existing table.

NOTE:

This operation will require exclusive lock on the table.

Rename table

Renames 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

Renames 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

Adds 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

Drops 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

Deletes a table or external table and all its data. Note that this operation requires an exclusive lock on the table.

NOTE:

- 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.

Examples:

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

Drop Table If Exists

Drops a table if it exists, or does nothing (returns an error that the table does not exist).

```
drop_table ::=
```

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

Examples:

```
DROP TABLE IF EXISTS my_schema.my_table;
```

External Tables

The following external table commands are available:

- 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 non-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.

NOTE:

- SQream accesses the files only at execution time, hence, errors in case of file/DDDL 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.

NOTE:*Wildcard options:*

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

• FIELD DELIMITER:

Specifies a character(s) as delimiter between fields in a row. Default is , (comma) the CSV standard field delimiter.

• RECORD DELIMITER:

Specifies 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.

Views

The following view commands are available:

- CREATE VIEW
- DROP VIEW
- RECOMPILE 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
```

Examples:

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

DROP VIEW

Deletes a view from the current database.

```
drop_view_statement ::=  
  
    DROP VIEW [schema_name].view_name ;
```

Examples:

```
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');
```

Users/Roles

SQream manages users by *roles*.

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](#)

The following role commands are available:

- CREATE ROLE
- DROP ROLE
- ALTER ROLE
- GRANT Permissions
- REVOKE Permissions

CREATE ROLE

Adds a new user/role to the cluster.

```
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

Deletes a role/user.

```
drop_role_statement ::=  
  
    DROP ROLE role_name ;
```

Examples:

```
DROP ROLE admin_role;
```

ALTER ROLE

Renames an existing role.

```
alter_role_statement ::=  
  
    ALTER ROLE role_name RENAME TO new_role_name ;
```

Examples:

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

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.

NOTE: For an example of how to configure roles in a cluster, see quick guide to roles and permissions.

Roles at a glance

- Roles are global across all databases in the cluster.
- However, for a role to function as a user in a database, it must have **USAGE** permission on the specific database (as well as any other relevant permissions).

- Roles are granted permissions and access to specific objects. The specified object can be any defined object such as a database or table.
- Roles can be granted permissions to other roles, thus creating a hierarchy of role with increasingly specific or limited permissions for lower-level users.

Roles and users

For a role to be used as a user, it must have PASSWORD, LOGIN and CONNECT permissions to the database it wishes to connect to.

When a role is assigned to a user (see GRANT PERMISSIONS), the user then has all the permissions granted to the role.

Roles and object permissions

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.

The role that creates an object is normally the owner of the object. This means:

- By default only that role can perform read/write/alter operations on the object.
- The object will be created in the role's default schema (usually **public**).

Any of the following can grant access permissions for an existing object to other roles.

- object owner
- superuser
- user with relevant admin permission via the **ADMIN OPTION**

Note that granted permissions affect only the specific existing objects. To grant the desired permissions also to objects created in the future, you must modify the role's default schema using the ALTER DEFAULT PERMISSIONS command.

Roles and schema

Roles are created using the **PUBLIC** schema so new users can create and manage their own objects in the PUBLIC schema. Once the role is created you can change the schema by using either:

- **GRANT** command to assign a different schema with its assigned permissions
- **ALTER DEFAULT PERMISSIONS** command to modify the current default schema

Superuser

A SUPERUSER is like an admin user who can grant permissions to other users. The superuser can create a role, granting it the relevant permissions and then give specified roles the ability to assign this role to other users. (This is the function of the **ADMIN OPTION** option of the **GRANT** command.)

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 any of the following:

- object owner
- superuser
- user with relevant admin permission via the **ADMIN OPTION**

GRANT Permissions

Assigns specific permissions to existing database objects, schema, or databases to one or more roles.

The identifier name can be the role name or CURRENT_ROLE.

The following table lists the possible permissions for various levels in the cluster.

Level	Possible permissions
cluster	LOGIN, PASSWORD, CREATE FUNCTION
database	SUPERUSER, CONNECT, CREATE, USAGE
schema	USAGE, CREATE
object	SELECT, INSERT, DELETE, DDL, EXECUTE, ALL

Examples:

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

-- Grant permissions at the database level:
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 ADMIN OPTION]

```

- **INSERT:** Allows INSERT of a new row into the specified table using INSERT/COPY command.
- **DELETE:** Allows DELETE of a row from the specified table. This privilege effects both DELETE and TRUNCATE commands.
- **DDL:** Allows DROP/ALTER.
- **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:** Has 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:** Grants a new password for the role. Without a password the role cannot function as a USER and login.
- **WITH ADMIN OPTION:** Grants the ability to give a certain role permission to others.

REVOKE Permissions

Removes permissions from one or more roles.

The identifier name can be the role name or CURRENT_ROLE.

Examples:

```

-- 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> [, ...]

-- Revoke privileges from other roles by granting one role
to another:
    REVOKE <role1> [, ...] FROM <role2> [, ...] WITH ADMIN
OPTION

```

Locks

See Locks in the Concepts section.

Generating the DDL

You can generate the DDL for any of the following:

- A whole database (See Generating the DDL for all objects of a database)
- A specific database object (See Generating the DDL of a single table, external table or view)
- A specific user defined function (UDF) (See Generating the DDL of a single user defined function (UDF))

Generating 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';
```

Generating 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.

NOTE:

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';
```

Generating 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');
```


Data Manipulation Language

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

INSERT

Adds 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);
```

NOTE:

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)

COPY FROM (bulk import)

Quickly inserts 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.

NOTE:

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.

TIP:

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.

NOTE:

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.

NOTE:

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

- **Parsers:**

Parsers 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	YYYY-MM-DD	2017-12-31	

Format Name	Format Pattern	Example	Note
or DEFAULT	[hh:mm:ss [.SSS]]	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	

NOTE:

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)

COPY TO (bulk export)

Saves 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
```

WARNING:

- 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;
```

TRUNCATE TABLE

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;
```

DELETE

Deletes 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).

NOTE:

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.

NOTE:

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 ) ;
```

TIP:

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.

NOTE:

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');
```

Operational Commands

Saved queries

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

You must first save the query using parameters in the query string. You can then execute the saved query specifying the parameter values.

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

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 ( ) ;
```

Examples:

```
-- Save a query using parameters in the query string
SELECT save_query('q1', $$select * from t where xint > ? AND
xdatetime < ? AND xvarchar6 <> 'something'$$)
```

```
-- Execute the saved query with specific values
SELECT execute_saved_query('q1', 1, '2013-12-02 12:01:22')

-- The result: SQream DB will execute the query using the specified
values:
SELECT * from t where xint > 1 AND xdatetime < '2013-12-
02 12:01:22' AND xvarchar6 <> 'something';
```

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 ]

```

NOTE:

See also **WHERE with HIGH_SELECTIVITY**

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;

```

The following table describes the Query options.

See below for more information regarding:

- Supported Value expressions
- Using the JOIN option
- WITH subqueries

Table xx. QUERY options

Name	Description	Example
TOP	Retrieves only the first rows from a query. TOP will be the last operation	

Name	Description	Example
	of the query execution. This means that SQream will limit the results to the end-user after executing the entire statement.	
DISTINCT	Removes duplicate rows.	
FROM	Specifies which tables to read in a query. FROM can either contain table/view names or subqueries.	
WHERE	Filters out rows. See also WHERE with HIGH_SELECTIVITY	SELECT Column1 FROM table1 WHERE column2 <= 1;
GROUP BY	Partitions a table so that aggregates can be applied separately to each partition.	select a,sum(b) from t group by a;
HAVING	Filters out rows after GROUP BY processing.	select a,sum(b) from t group by a having sum(b) > 5;
ORDER BY	Orders the results.	select * from t order by a asc, b desc;
VALUES	Creates a 'literal table value'.	values (1,'a'), (2,'b');
UNION	Concatenates two queries together. SQream currently supports UNION ALL , which doesn't remove duplicate rows.	select * from t union all select * from u ;

NOTE:

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)

Value expressions

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;
```

JOIN

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;
```

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;
```

NOTE:

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

Manual Query Tuning

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.

TIP:

- 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');
```

NOTE:

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

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
```

Boolean

Table 23. Boolean data type

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

NOTE:

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

Numeric types

Table 25. Numeric data types

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

WARNING:

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;`

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'

NOTE:

Time zones are not supported.

NOTE:

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

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).

NOTE:

- 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).

Value expressions

Value expressions are used in the following:

- Select lists
- **ON** conditions
- **WHERE** conditions
- **GROUP BY** expressions
- **HAVING** conditions
- **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
```

The following table describes the value expressions.

See below for more information regarding:

- Special operator
- EXTRACT operator
- CASE expression
- CASE expression
- Identifier rules
- Operator precedences

Table xx. Value expressions

Name	Example	Comment
String literal	SELECT 'string literal'; SELECT 'string literal ' 'with something' ' quoted';	Delimited by single quotes ('), and can contain any printable character other

Name	Example	Comment
	<pre>-- this produces the string "string literal 'with something' quoted" SELECT \$\$string literal 'with something' quoted\$\$; -- Same as above</pre>	<p>than single quote</p> <p>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 (")</p> <p>Similarly, to avoid the single quote issue, use the <i>dollar quoting</i> notation (see examples)</p>
Number literal	<pre>digits digits . [digits] [e [+ -] digits] [digits] . digits [e [+ -] digits digits e [+ -] digits</pre>	<pre>1234 1234.56 12. .34 123.56e-45</pre>
Typed literal	type_name string_literal	type_name is defined above in the type name definition section.
Binary operator	<pre>. + ^ * / % + - >= <= != <> LIKE NOT LIKE RLIKE NOT RLIKE < > = OR AND</pre>	
Unary operator	+ - NOT	
Postfix unary operator	IS NULL IS NOT NULL	
Special operator	See Special operator	
EXTRACT operator	See EXTRACT operator	
CASE expression	See CASE expression	
Identifier Rules	See Identifier rules	
Aggregate function app	<pre>agg_name ([value_expr [, ...]]) agg_name ([DISTINCT] [value_expr [, ...]]) agg_name ::= identifier</pre>	

Name	Example	Comment
Window function app	<pre> window_fn_name ([value_expr [, ...]]) OVER ([value_expr [, ...]] [PARTITION BY value_expr [, ...]] [ORDER BY value_expr [ASC DESC] [, ...]]) window_fn_name ::= identifier </pre>	See the Window functions segment for examples.

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

```

NOTE:

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;

```

NOTE:

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

```

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

NOTE:

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
```

Examples:

```
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

extract_field	Syntax	Result
MILLISECONDS	EXTRACT (MILLISECONDS FROM '1986-06-02 15:31:22.124')	22124.00

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
```

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;
```

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.

Functions and Operators

Operators

Logical

Table 30. Logical operators

Operator	Description	Example
and	logical and (bool, bool) returns bool	TRUE AND FALSE
or	logical or (bool, bool) returns bool	a OR b
not	logical not (bool) returns bool	NOT TRUE

Comparison

Table 31. Comparison operators

Operator	Description	Example	Comments
< > <= >= == !=	Regular binary comparison operations (any, any) returns bool 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).	binary_ comparison_ operator (any, any) returns bool	
between	Exp is between min and max inclusive exp between min and max is shorthand for exp >= min and exp <= max. (exp any, min any, max any) returns bool	a between b and c	
not between	Inverse of between exp not between min and max is shorthand for not (exp between min and max) (exp any, min any, max any) returns bool		
is null	Checks if the argument is null. (any) returns bool	(1+null) is null	

Operator	Description	Example	Comments
<code>is not null</code>	Checks if the argument isn't null. (<i>any</i>) returns bool	<code>(a*b) is not null</code>	You cannot use <code>exp = NULL</code> to test if a value is null, or <code>exp <> NULL</code> to test if a value is not null. You must use IS NULL and IS NOT NULL
<code>in</code>	Tests for membership in a list. (<i>any</i> [, ...]) returns bool	<code>a in (1 , 3 , 5 , 7 , 11)</code>	IN subqueries are not supported.

NOTE:

any is any type..

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
<code>&</code>	bitwise AND	Result is true only if both operands are true.	<code>101 & 110</code>	<code>100</code>
			<code>5 & 6</code>	<code>4</code>
<code> </code>	bitwise inclusive OR	Result is true if any of the	<code>101 110</code>	<code>111</code>

Symbol	Operator	Short Description	Example	Returns
		operands is true.		
			5 6	7
<code>xor</code>	bitwise XOR (eXclusive OR)	Result is true only if one of its operands is true.	<code>xor(101,011)</code>	110
			<code>xor(5,3)</code>	6
<code>~</code>	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.	<code>~0100</code>	-101
			<code>~4</code>	-5
<code>>></code>	Shift right	Moves the bits the number of positions specified by the second operand in the right direction.	<code>8 >>2</code>	2
<code><<</code>	Shift left	Moves the bits the number of positions specified by the second operand in the left direction.	<code>3 <<2</code>	12

NOTE:

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

Mathematical functions and operators

NOTE:

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.

Table 33. Mathematical functions and operators

Name	Description	Example
SQRT	Square root of the argument to the function Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT SQRT (cfloat) FROM table;</code>
ABS	$ x $ - Absolute (positive) value of the argument Parameter:s Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT ABS (cfloat) FROM table;</code>
ROUND	Rounds the number to the nearest precision Parameters: Float/Double (All other numbers available via implicit casting). Int32T Precision (number of places after the decimal point) Returns: Float/Double.	<code>SELECT ROUND (cfloat,2) FROM table;</code>
ASIN	$\sin^{-1}(x)$ – Arcsine (angle in radians whose sine is the argument of the function) Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT ASIN (cfloat) FROM table;</code>
ATAN	$\tan^{-1}(x)$ - Arctangent (angle in radians whose tangent is the argument of the function) Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT ATAN (cfloat) FROM table;</code>
ATN2	Angle in radians between positive X-	<code>SELECT ATN2 (cfloat,cfloat2) FROM table;</code>

Name	Description	Example
	<p>axis and the ray from the origin where x and y are the first and second arguments.</p> <p>Parameters: Float/Double (All other numbers available via implicit casting) / Float/Double (All other numbers available via implicit casting). Returns: Float/Double.</p>	
COS	<p>cos x - Trigonometric cosine of the angle in radians</p> <p>Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.</p>	SELECT COS (cfloat) FROM table;
COT	<p>cot x - Trigonometric cotangent of the angle in radians</p> <p>Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.</p>	SELECT COT (cfloat) FROM table;
CEILING or CEIL	<p>Returns the smallest integer value that is greater than or equal to the argument.</p> <p>Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.</p>	SELECT CEILING (cfloat) FROM table; SELECT CEIL (cfloat) FROM table;
LOG10	<p>log10 x- Base 10 logarithm of the argument</p> <p>Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.</p>	SELECT LOG10 (cfloat) FROM table;
LOG	<p>ln x - Natural base logarithm (ln or loge) of the argument</p> <p>Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.</p>	SELECT LOG (cfloat) FROM table;
LOG (base-y)	<p>Base-y Logarithm of the x parameter, where x,y are the arguments</p> <p>Parameters: Float/Double (All other numbers available via implicit casting). Integer Base Returns: Float/Double.</p>	SELECT LOG (cfloat,cint) FROM table; SELECT LOG (cfloat,8) FROM table;

Name	Description	Example
MOD	Returns the remainder from a division of argument#1 by argument #2. Parameters: Integer Returns: Integer	<code>SELECT MOD (cint,cint) FROM table;</code>
FLOOR	Returns the smallest integer to the argument. Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT FLOOR (cfloat) FROM table;</code>
SIN	Trigonometric sine of the angle in radians Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT SIN (cfloat) FROM table;</code>
SQUARE	x^2 - Square of the argument Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT SQUARE (cfloat) FROM table;</code>
TAN	Tangent of the argument Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT TAN (cfloat) FROM table;</code>
PI	Mathematical constant Parameters: None Returns: Double (value of Pi to 10 digits after the decimal point).	<code>SELECT PI () FROM table;</code>
POWER	Performs a power of one value over the other (x raised to the power of y). Parameters: Float/Double (All other numbers available via implicit casting) / Float/Double (All other numbers available via implicit casting). Returns: Float/Double.	<code>SELECT POWER (cfloat1,cfloat2) FROM table;</code>
TO_HEX	Converts an integer to its base-16 string representation. Parameters: Int/Long parameter. Returns: VarChar - Base-16 string representation.	<code>SELECT TO_HEX (x) FROM table;</code>
TRUNC	Returns the argument truncated to a whole number.	<code>SELECT TRUNC (cfloat) FROM table;</code>

Name	Description	Example
	Parameters: Float/Double (All other numbers available via implicit casting). Returns: Float/Double (same data type as parameter).	

Check functions

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 all be the same type.

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

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).

Table 34. 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	

From type	To Type	Context	Comments
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	
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	

CAST

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
```

** 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

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

NOTE:

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

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

A type 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.

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.

Table 35. String functions and operators on VARCHAR

Name	Description	Example	Comment
LOWER	Converts a string to lowercase .	SELECT LOWER (varchar_column) FROM table;	
UPPER	Converts a string to uppercase.	SELECT UPPER (varchar_column) FROM table;	
LEN	Returns the length of a varchar.	SELECT LEN (varchar_column) FROM table;	Trailing whitespace on the right is ignored: LEN on 'abc' and 'abc ' will both return 3
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	Checks if a string matches a regex pattern.	SELECT * FROM table WHERE varchar_ column RLIKE '[0-9]+\$';	Does not work on NVARCHAR columns
SUBSTRING	Returns a specific substring of a string.	SELECT SUBSTRING (varchar_ column ,start,length) FROM table;	<ul style="list-style-type: none"> ◦ Parameters: See String Parameters ◦ Returns: String of the resulting substring operation <p>If start ≠ 1, then the substring begins from</p>

Name	Description	Example	Comment
			<p>the first character but the length is reduced.</p> <pre>substring ('abc',1,2) == 'ab' substring ('abc',0,2) == substring ('abc',1,1) == 'a'</pre>
REGEXP_COUNT	<p>Counts regex matches in string.</p> <p>For example, the pattern '[1-9]' appears once in '01' and twice in '12'.</p>	<pre>SELECT REGEXP_COUNT (col, '[0-9]', 2) FROM table;</pre>	<ul style="list-style-type: none"> Parameters: See String Parameters Returns: Number of matches of the regex pattern in the string (int)
REGEXP_INSTR	<p>Matches regex and returns the position in a string of the n-th occurrence.</p>	<pre>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;</pre>	<ul style="list-style-type: none"> Parameters: See String Parameters Returns: Position of the first occurrence of the regex pattern (int)
REGEXP_SUBSTR	<p>Matches regex and returns it.</p>	<pre>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;</pre>	<ul style="list-style-type: none"> Parameters: See String Parameters Returns: String of the matched column
ISPREFIXOF	<p>Checks if one string is a prefix of the other.</p>	<pre>SELECT ISPREFIXOF (x,y) FROM table</pre>	<p>Internal function.</p> <p>"isprefix(x,y)" is equivalent to "y LIKE x + '%'", but more efficient.</p>

Name	Description	Example	Comment
 (Concatenation)	String concatenation - concatenates two string values.	<pre>SELECT fname ' ' lname FROM customers;</pre>	
CHARINDEX	Returns the position of a subexpression in an expression.	<pre>SELECT CHARINDEX (y,x,1) FROM table;</pre>	<ul style="list-style-type: none"> 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	Returns the position of a pattern in an expression.	<pre>SELECT PATINDEX ('%[0-9]%',x) FROM table;</pre>	<ul style="list-style-type: none"> 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	Trims leading whitespace from the left side of the string.	<pre>SELECT LTRIM (x) FROM table;</pre>	<ul style="list-style-type: none"> Parameters: VarChar Returns: VarChar
REVERSE	Reverses a string.	<pre>SELECT REVERSE (x) FROM table;</pre>	<ul style="list-style-type: none"> Parameters: VarChar Returns: VarChar
RTRIM	Trims trailing whitespace from the right side of the string.	<pre>SELECT RTRIM (x) FROM table;</pre>	<ul style="list-style-type: none"> Parameters: VarChar Returns: VarChar
TRIM	Removes any	<pre>SELECT TRIM (x)</pre>	<ul style="list-style-type: none"> Parameters:

Name	Description	Example	Comment
	whitespace from the beginning and end of a string (leading and trailing).	FROM table;	VarChar ◦ Returns: VarChar

String 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. Occurrence number - (int) - which occurrence of the pattern - optional. When unset, default is 1 - first occurrence
5. Match start/end position - (int) - 0 for match's start position, 1 for its end - optional. When unset, default is 0

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.

Table 36 String functions and operators on NVARCHAR

Name	Description	Example	Comment
LOWER	Converts ASCII string to lowercase	SELECT LOWER (nvarchar_column) FROM table;	For non-ASCII characters the function will return the original column data.
UPPER	Converts ASCII string to uppercase	SELECT UPPER (nvarchar_column) FROM table;	For non-ASCII characters the function will return the original column data.
LEN	Returns the length of a nvarchar while trimming whitespace.	SELECT LEN (nvarchar_column) FROM table;	
CHAR_LENGTH	Returns the length of a nvarchar without trimming whitespaces.	SELECT CHAR_LENGTH (nvarchar_column) FROM table;	

Name	Description	Example	Comment
LIKE	<p>Checks if a string matches a LIKE pattern.</p> <p>For NVARCHAR columns the following options for like function exists:</p> <ul style="list-style-type: none"> • '%string' • 'string%' • '%string%' <p>Also used as NOT LIKE.</p>	<pre>SELECT * FROM table WHERE nvarchar_column LIKE '%string%'; SELECT * FROM table WHERE nvarchar_column NOT LIKE '%string%';</pre>	Only literal patterns are supported. Column references are not supported as a pattern.
SUBSTRING	Returns a specific substring of a string	<pre>SELECT SUBSTRING (nvarchar_ column ,start,length) FROM table;</pre>	<p>Parameters:</p> <ul style="list-style-type: none"> ◦ start - the starting point of the substring. A value 1 represents the first character. ◦ Returns: String of the resulting substring operation <p>If start ≠ 1, then the substring begins from the first character but the length is reduced.</p> <pre>substring ('abc',1,2) == 'ab' substring ('abc',0,2) == substring ('abc',1,1) == 'a'</pre>
 (Concatenation)	String concatenation - concatenates two string values	<pre>SELECT fname lname FROM customers;</pre>	
CHARINDEX	Searches an expression in a string nvarchar column and returns	<pre>SELECT CHARINDEX (y,x,1) FROM table;</pre>	<ul style="list-style-type: none"> ◦ Parameters: NVarChar - the subexpression. as a scalar.

Name	Description	Example	Comment
	its starting position if found.		<p>NVarChar - the column name. Int (optional) - starts the search from this index</p> <ul style="list-style-type: none"> Returns: Int - the position of the subexpression in the expression or 0 if it wasn't found
LEFT	Returns the left part of a character string with the specified number of characters.	<pre>SELECT LTRIM (x) FROM table;</pre>	<ul style="list-style-type: none"> Parameters: VarChar Returns: VarChar
REPLACE	Replaces a sub-string with another sub-string of the same size.	<pre>SELECT REPLACE (x, 'a', 'b') FROM table; SELECT REPLACE (x, '1', '*') FROM table; SELECT REPLACE (x, '123', '321') FROM table;</pre>	<ul style="list-style-type: none"> Parameters: NVarChar Returns: NVarChar <p>LIMITATIONS</p> <ul style="list-style-type: none"> A sub-string can only be replaced 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.
REVERSE	Reverses a string;	<pre>SELECT REVERSE (x) FROM table;</pre>	<ul style="list-style-type: none"> Parameters: NVarChar Returns: NVarChar
RIGHT	Returns the right part of a character string with the specified number of characters.	<pre>SELECT RIGHT (x, 3) FROM table;</pre>	<ul style="list-style-type: none"> Parameters: NVarChar Returns: NVarChar
OCTET_LENGTH	Returns the length in bytes (octets) of	<pre>SELECT OCTET_ LENGTH (x) FROM</pre>	

Name	Description	Example	Comment
	the nvarchar column value (being the number of bytes in binary string).	table;	

NOTE:

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.

Pattern matching syntax

Table 35. 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

Regular Expression Pattern Matching Syntax

Table 36. 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

Syntax	Description
[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.

Date and Datetime

Table 37. Date and Datetime functions

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

Name	Syntax	Description	Return data type
		examples below.	
datepart	<code>datepart (interval, date_column)</code> <code>datepart (interval, datetime_column)</code>	Returns the number of a the specified datepart of a date or datetime.	Integer
datediff	<code>datediff (interval, *startdate*, *enddate*)</code>	Returns the difference between two dates based on the specified interval. Same as function EXTRACT.	Integer
dateadd	<code>dateadd (interval, number, date_column)</code> <code>dateadd (interval, number, datetime_column)</code>	Adds a time/date interval to a date.	Datetime
to_unixts	<code>to_unixts (datetime_column)</code>	Converts to unix timestamp, seconds since epoch.	Bigint
to_unixtsms	<code>to_unixtsms (datetime_column)</code>	Converts to unix timestamp, milliseconds since epoch.	Bigint
from_unixts	<code>from_unixts (bigint_column)</code>	Converts unix timestamp, seconds since epoch.	Datetime
from_unixtsms	<code>from_unixtsms (bigint_column)</code>	Converts unix timestamp, milliseconds since epoch.	Datetime
eomonth	<code>eomonth (datetime_column)</code> <code>eomonth (date_column)</code>	Returns the last day of the month (end of month).	Datetime / Date

See also EXTRACT

Table 38. Interval Options

Interval	Shorthand aliases
year	YYYY, YY
quarter	qq, q
month	mm, m
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

```

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:

Example:

```
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.

Example:

```
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 39. Geospatial functions

Name	Type	Description	LIMITATIONS
<code>*point_in_polygon*</code>	<code>(_point_long_ *float*, _point_lat_ *float*, _poly_ long1_ *float*, _ poly_lat1_ *float*, + + ... + + _poly_ long5_ *float*, _ poly_lat5_ *float*) *returns* *bool*</code>	point inside polygon	The point arguments cannot be literals. The polygon arguments can either be all columns or all literals.
<code>*line_crosses_polygon*</code>	<code>(_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*</code>	line crosses polygon	The polyline arguments cannot be literals. The polygon arguments can either be all columns or all literals.

Aggregate functions

Table 40. Aggregate functions

Name	Syntax	Return type	Description
avg	<code>avg(<i>anynumber</i>)</code>	float	average
count	<code>count(<i>any</i>)</code>	int	count
max	<code>max(<i>any</i>)</code>	<i>any</i>	maximum
min	<code>min(<i>any</i>)</code>	<i>any</i>	minimum
sum	<code>sum(<i>anynumber</i>)</code>	<i>anynumber</i>	sum
stddev	<code>stddev(<i>anynumber</i>)</code>	float	standard deviation

any can be any type as defined in Data types.

anynumber is any numeric type, as defined in Numeric Types

WARNING:

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;`

Window functions

Table 41. Window functions

Name
<code>rank()</code>
<code>row_number()</code>
<code>min()</code>
<code>max()</code>
<code>sum()</code>

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;
```

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.)

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;
```

NOTE:

- 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 $$
import sys
sys.path.append("/home/sqream/pythonpath")
import writefile as f
f.main()
return 1
$$ LANGUAGE PYTHON;

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

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();
```

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');
```

Examples:

```
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;
```