

## Flink SQL

Date published: 2019-12-17

Date modified: 2022-02-28



# Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

# Contents

<b>Flink SQL Overview.....</b>	<b>4</b>
Flink DDL.....	4
Flink DML.....	4
Flink Queries.....	4
Other supported statements.....	5
Data Types.....	5
Dynamic SQL Hints.....	6
SQL Examples.....	6
 <b>Enriching streaming data with join.....</b>	 <b>9</b>
Joining streaming and bounded tables.....	10
Example: joining Kafka and Kudu tables.....	11

## Flink SQL Overview

As Flink SQL is used in SQL Stream Builder, you can execute the supported Flink DDL, DML and query statements directly from the SQL window in Streaming SQL Console.

### Flink DDL

Flink SQL supports Data Definition Language (DDL) statements to create, modify and remove objects within a data structure.

The following table summarizes the supported DDL statements in SQL Stream Builder:

DDL	Description	Option
CREATE TABLE	Creating table for the SQL query. You can create tables based on the supported connectors.	<ul style="list-style-type: none"> <li>Adding WATERMARK and PRIMARY KEY information</li> <li>Creating table with LIKE clause</li> </ul>
CREATE VIEW	Creating custom views using columns from tables. There is no physical data behind a view.	<ul style="list-style-type: none"> <li>Adding queries, expressions and joins</li> </ul>
CREATE FUNCTION	Creating User Defined Function (UDF) for a query.	<ul style="list-style-type: none"> <li>Using Javascript or Java language</li> <li>Using Functions tab on Streaming SQL Console</li> </ul>
DROP TABLE	Deleting a table, view or function.	<ul style="list-style-type: none"> <li>Using Streaming SQL Console functionality to delete table, view or function</li> </ul>
DROP VIEW		
DROP FUNCTIONS		
ALTER TABLE	Modifying table, view or function properties.	<ul style="list-style-type: none"> <li>Using RENAME TO or SET for table</li> <li>Using RENAME TO or AS for view and function</li> </ul>
ALTER VIEW		
ALTER FUNCTION		

### Flink DML

Flink SQL supports Data Manipulation Language (DML) statements to manipulate the data itself with adding, deleting or modifying.

The following table summarizes the supported DML statements in SQL Stream Builder:

DML	Description	Option
INSERT INTO	Inserting query results into a specified table.	<ul style="list-style-type: none"> <li>Inserting columns or values into a table</li> <li>Adding OVERWRITE to overwrite the existing data in the table</li> </ul>

### Flink Queries

Flink SQL supports querying data with SELECT statement and using different types of operations.

You can query data from a table using the SELECT statement.

Query	Description	Option
SELECT	Querying data from a table using different operations.	<ul style="list-style-type: none"> <li>Selecting all data in a table</li> <li>Selecting data using different types of operations</li> </ul>

The following table summarizes the supported operations for SELECT statement in SQL Stream Builder:

Operation	Description	Option
WHERE	Querying data based adding filters.	<ul style="list-style-type: none"> <li>Using boolien expression</li> </ul>
JOIN	Joining data from tables based on a equivalent column.	<ul style="list-style-type: none"> <li>Using temporal joins based on event time or processing time</li> <li>Using lookup join</li> </ul>
GROUP BY	Grouping results using built-in or user defined functions.	<ul style="list-style-type: none"> <li>Using with streaming table procudes updated results</li> </ul>
ORDER BY	Ordering results to be sort based on a specified expression.	<ul style="list-style-type: none"> <li>Using with streaming table the primary sorting key needs to be time</li> </ul>

## Other supported statements

Beside the supported DDL, DML and SELECT, the supported statements also include DESCRIBE, SHOW and SET that you can use in SQL Stream Builder.

The following table summarizes the supported statements for in SQL Stream Builder:

Operation	Description
DESCRIBE TABLES	Describing the schema of a table Showing a list of existing tables, views, functions, databases or catalogs
SHOW TABLES	
SHOW VIEWS	
SHOW FUNCTIONS	
SHOW DATABASES	
SHOW CATALOGS	
SET	Setting properties of session

## Data Types

The logical type of a value to declare input and output types of operations in a table ecosystem is described by data types. Flink support a set of pre-defined data types that can be also used in SQL Stream Builder (SSB).

The following list summarizes the pre-defined data types in Flink and SQL Stream Builder:

- CHAR
- VARCHAR
- STRING
- BOOLEAN
- BINARY
- VARBINARY
- BYTES
- DECIMAL - Supports fixed precision and scale.
- TINYINT
- SMALLINT

- INTEGER
- BIGINT
- FLOAT
- DOUBLE
- DATE
- TIME - Only supports a precision of 0.
- TIMESTAMP
- TIMPESTAMP\_LTZ
- INTERVAL - Only supports interval of MONTH and SECOND(3).
- ARRAY
- MULTISSET
- MAP
- ROW
- RAW
- Structured types - Only exposed in user-defined functions.

For more information about Data Types in Flink SQL, see the [Apache Flink documentation](#).

## Dynamic SQL Hints

SQL hints are supported for SQL Stream Builder (SSB) that allows you to use the dynamic table options. With the dynamic table options, you can alter any option of a table on a query level.

The dynamic table options of Flink SQL allows you to specify and override options for a table in a SQL query. The hint is formatted as a SQL comment containing an `OPTIONS()` clause, for example:

```
/*+ OPTIONS('key1'='value1', 'key2'='value2') */
```

You need to place the clause directly afte the table name where you need to change the options. You can use separate clauses for separate tables within a SQL query.

For example, you can use the following SQL hint to dynamically configure a Kafka consumer group at run time:

```
SELECT t1.column_a, t2.column_b
FROM tableName_A /*+ OPTIONS('properties.group.id'='my_consumer_group') */ AS t1
JOIN tableName_B /*+ OPTIONS('properties.group.id'='my_other_consumer_group') */ AS t2
ON t1.column_a = t2.column_b
```

Any option of a table that is accessible in the DDL of a connector can be overridden using the SQL hints. The options of a connector can be viewed on the **Connector** page of Streaming SQL Console.

For more information about the SQL Hints, see the official [Apache Flink documentation](#).

## SQL Examples

You can use the SQL examples for frequently used functions, syntax and techniques in SQL Stream Builder (SSB). SSB uses Calcite Compatible SQL, but to include the functionality of Flink you need to customize certain SQL commands.

### Metadata commands

```
-- show all tables
SHOW tables;
-- describe or show schema for table
```

```
DESCRIBE payments;
DESC payments;
```

### Timestamps, intervals and time

```
-- eventTimestamp is the Kafka timestamp
-- as unix timestamp. Magically added to every schema.
SELECT max(eventTimestamp) FROM solar_inputs;

-- make it human readable
SELECT CAST(max(eventTimestamp) AS varchar) as TS FROM solar_inputs;

-- date math with interval
SELECT * FROM payments
WHERE eventTimestamp > CURRENT_TIMESTAMP-interval '10' second;
```

### Aggregation

```
-- hourly payment volume

SELECT SUM(CAST(amount AS numeric)) AS payment_volume,
CAST(TUMBLE_END(eventTimestamp, interval '1' hour) AS varchar) AS ts
FROM payments
GROUP BY TUMBLE(eventTimestamp, interval '1' hour);

-- detect multiple auths in a short window and
-- send to lock account topic/microservice

SELECT card,
MAX(amount) as theamount,
TUMBLE_END(eventTimestamp, interval '5' minute) as ts
FROM payments
WHERE lat IS NOT NULL
AND lon IS NOT NULL
GROUP BY card, TUMBLE(eventTimestamp, interval '5' minute)
HAVING COUNT(*) > 4 -- >4==fraud
```

### Working with arrays

```
-- unnest each array element as separate row
SELECT b.*, u.*
FROM bgp_avro b,
UNNEST(b.path) AS u(pathitem)
```



**Note:** Arrays start at 1 not 0.

### Union ALL

```
-- union two different tables
SELECT * FROM clickstream
WHERE useragent = 'Chrome/62.0.3202.84 Mobile Safari/537.36'
UNION ALL
SELECT * FROM clickstream
WHERE useragent = 'Version/4.0 Chrome/58.0.3029.83 Mobile Safari/537.36'
```

## Math

```
-- simple math
SELECT 42+1 FROM mylogs;
-- inline
SELECT (amount+10)*upcharge AS total_amount
FROM payments
WHERE account_type = 'merchant'
```

```
-- convert C to F
SELECT (temp-32)/1.8 AS temp_fahrenheit
FROM reactor_core_sensors;
```

```
-- daily miles accumulator, 100:1
-- send to persistent storage microservice
-- for upsert of miles tally
SELECT card,
SUM(amount)/100 AS miles,
TUMBLE_END(eventTimestamp, interval '1' day)
FROM payments
GROUP BY card, TUMBLE(eventTimestamp, interval '1' day);
```

## Joins

```
-- join multiple streams
SELECT o.name,
      sum(d.clicks),
      hop_end(r.eventTimestamp, interval '20' second, interval '40' second)
FROM click_stream o join orgs r on o.org_id = r.org_id
      join models d on d.org_id = r.org_id
GROUP BY o.name,
      hop(r.eventTimestamp, interval '20' second, interval '40' second)
```

```
-- join with temporal table where LatestRates is a temporal table
SELECT
  o.amount, o.currency, r.rate, o.amount * r.rate
FROM
  Orders AS o
  JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r
  ON r.currency = o.currency
```

## Hyperjoins

Joins are considered "hyperjoins" because SQL Stream Builder has the ability to join multiple tables in a single query, and because the Kafka table is created from a data provider, these joins can span multiple clusters/connect strings, but also multiple types of sources (join Kafka and a database for instance).

```
SELECT us_west.user_score+ap_south.user_score
FROM kafka_in_zone_us_west us_west
FULL OUTER JOIN kafka_in_zone_ap_south ap_south
ON us_west.user_id = ap_south.user_id;
```

## Misc SQL tricks

```
-- concatenation
```



```
SELECT 'testme_' || name FROM logs;
```

```
-- select the datatype of the field
SELECT eventTimestamp, TYPEOF(eventTimestamp) as mytype FROM airplanes;
```

### Escaping and quoting

Typical escaping and quoting is supported.

- Nested columns

```
SELECT foo.`bar` FROM table; -- must quote nested column
```

- Literals

```
SELECT "some string literal" FROM mytable; -- a literal
```

### Built In Functions

```
-- convert EPOCH time to timestamp
select EPOCH_TO_TIMESTAMP(1593718981) from ev_sample_fraud;

-- convert EPOCH milliseconds to timestamp
select EPOCHMILLIS_TO_TIMESTAMP(1593718838150) from ev_sample_fraud;
```

## Enriching streaming data with join

In SQL Stream builder, you can enrich your streaming data with values from a slowly changing dataset using join statements.

### Regular join

Join statements in SQL serves to combine columns and rows from two or more tables based on a shared column. When you join tables from a slowly changing source such as HDFS, Kudu, Hive and so on, you can simply use the regular JOIN syntax of SQL. The following example shows a regular INNER JOIN where the *Orders* table is joined with *Product* table based on the *productId*:

```
SELECT * FROM Orders
INNER JOIN Product
ON Orders.productId = Product.id
```

A regular join can only be used with bounded tables. In a streaming context data is produced continuously, and with a regular join both sides of the join would need to be buffered indefinitely to store all of the events that would match with the result of the SQL query. To get results from a given amount of time and to join streaming tables, a time boundary needs to be specified. This means the tables not only need to be joined by a key or column, but also on a time attribute.

### Interval join

When joining streaming tables, the time attribute can be defined in the SQL syntax using BETWEEN and an interval value:

```
SELECT *
FROM Orders o, Shipments s
WHERE o.id = s.order_id
```

```
AND o.order_time BETWEEN s.ship_time - INTERVAL '4' HOUR AND s.ship_time
```

In this case, *Orders* table is joined with the *Shipments* table and the results are going to be generated based on the *id* column as long as the order time and shipment time is within four hours of each other. The condition of this scenario is that the events in the streams happen almost at the same time with minimal delay, so the time boundary can be defined between an approximate interval.

When you want to join a streaming table with a slowly changing table, time attributes can differ as one of the tables stores data over a long period of time, while the streaming table receives new data continuously. To join these types of tables, a time needs to be defined that can serve as a reference point for both types of tables.

## Joining streaming and bounded tables

Beside regular join and interval join, in Flink SQL you are able to join a streaming table and a slowly changing dimension table for enrichment. In this case, you need to use a temporal join where the streaming table is joined with a versioned table based on a key, and the processing or event time.

A versioned table is a table that contains a time attribute, and reflects the records from a table at a specific point of time. When you use append-only or regularly updated sources, the values related to a key are updated over a long period of time. For example, a table can contain the currency rates since last month. At every change of the currency rate, a new value is added to the stream, therefore to the table. With creating a versioned table of the currency rate, you can specify which rate you need in an exact point of time: use the currency rates from 12:00.

For more information of Version Tables, see the official [Apache Flink documentation](#).

### Temporal join

After determining the version of the bounded table, you also need to define a time for the streaming table. In Flink, event time and processing time can be specified. When using event time, you need to create a temporal join.

Event time is the time that each individual event occurred on its producing device. To have an event time attribute in your SQL query, you need to define a timestamp column with a watermark definition column when creating the table:

```
CREATE TABLE orders (
  order_id    STRING,
  price       DECIMAL(32,2),
  currency    STRING,
  order_time  TIMESTAMP(3),
  WATERMARK FOR order_time AS order_time
) WITH (
  ...
)
```

When you want to use event time in a JOIN, you need to refer to the event time column as defined for the table in the *FOR SYSTEM\_TIME AS OF* part of the SQL query:

```
SELECT
  order_id,
  price,
  currency,
  conversion_rate,
  order_time,
FROM orders
LEFT JOIN currency_rates FOR SYSTEM_TIME AS OF orders.order_time
ON orders.currency = currency_rates.currency
```

## Lookup join

As a special case of temporal join, you can use the processing time as a time attribute. In Flink, processing time is the system time of the machine, also known as “wall-clock time”. When you use the processing time in a JOIN SQL syntax, Flink translates into a lookup join and uses the latest version of the bounded table. The following example shows the join syntax that needs to be used for enriching streaming data:

```
SELECT o.order_id, o.total, c.country, c.zip
FROM Orders AS o
      JOIN Customers FOR SYSTEM_TIME AS OF PROCTIME()
      ON o.customer_id = c.id
```

In the above example, *Customers* serves as the lookup table. The `FOR SYSTEM_TIME AS OF PROCTIME()` syntax indicates that you always want to look up in the latest version of the table. With including the processing time in the SQL syntax, you can query the latest version of a lookup table, and enrich your streaming data with the corresponding value.



**Note:** When using SQL Stream Builder, you can simply use the `PROCTIME()` function as the version of the lookup table when performing a lookup join. This means that you do not need to create and reference a processing time column in your probe stream anymore.

In SQL Stream Builder, the following connectors are supported as lookup tables:

- Kudu
- Hive
- JDBC

## Example: joining Kafka and Kudu tables

Using lookup join, you can join Kafka and Kudu tables to enrich the streaming data of Kafka with information from the Kudu tables. In this example, Orders of a Kafka streaming table are enriched with metadata information from a Kudu table.

As a prerequisite for the example, the following steps were completed in SQL Stream Builder:

- Registering Kafka as a Data Provider.
- Registering Kudu as a Catalog.
- Creating Orders Kafka table.
- Creating ItemMeta Kudu table.
- Generating data for Kafka and Kudu tables.

The tables in the example contain the following information:

Tables	Columns
Kafka - Orders	<ul style="list-style-type: none"> <li>• order_number</li> <li>• price</li> <li>• order_time</li> <li>• item_id</li> </ul>
Kudu - ItemMeta	<ul style="list-style-type: none"> <li>• id</li> <li>• info</li> </ul>

In the scope of this example, the *Orders* table will be joined with latest version of the *ItemMeta* table based on the item *id*, and the selected information is sampled under the **Results** tab of the Streaming SQL Console:

```
SELECT order_time, item_id, info, price
FROM Orders
      JOIN kudu.default_database.ItemMeta FOR SYSTEM_TIME AS OF PROCTIME()
      ON item_id = id
```

After running the SQL query, the results are continuously sampled to the Streaming SQL Console, the rows of *order\_time*, *item\_id* and *price* are enriched with *info* column from the *ItemMeta* Kudu table:

