

## SSB Application Management

Date published: 2024-06-15

Date modified: 2024-02-28



# Legal Notice

© Cloudera Inc. 2025. 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>Functions.....</b>	<b>4</b>
Creating Python User-defined Functions.....	4
Using System Functions.....	7
<b>Job settings.....</b>	<b>7</b>
<b>Sampling.....</b>	<b>8</b>
Sample results.....	9
<b>Connectors.....</b>	<b>9</b>
Using connectors with templates.....	10
Adding new connectors.....	11
Kafka connectors.....	12
CDC connectors.....	14
JDBC connector.....	15
Filesystem connector.....	15
Webhook connector.....	15
Creating Webhook tables.....	16
Datagen connector.....	18
Faker connector.....	18
Blackhole connector.....	19

# Functions

## Creating Python User-defined Functions

With Cloudera SQL Stream Builder, you can create powerful custom functions in Python to enhance the functionality of SQL.

### About this task

User functions can be simple translation functions like Celsius to Fahrenheit, more complex business logic, or even looking up data from external sources. User functions are written in Python. When you write them, you can create a library of useful functions.



**Important:** Support for JavaScript UDFs was deprecated in the 1.13.0 version of Cloudera Streaming Analytics, and will be removed in a future release. Python UDFs will be used as a replacement, and users should begin migrating their existing UDFs to Python.

### Before you begin

Python UDFs are disabled by default. To enable them:

1. Python UDF execution requires a supported Python version installed **on all nodes**.

- a. Access your Cloudera SQL Stream Builder node's command line. (Execute the following commands as root.)
- b. Install the Python Apache Flink and dependent packages:

```
/usr/local/bin/python[***VERSION***] -m pip install apache-flink==1.19.1
```

2. Go to your cluster in Cloudera Manager.
3. Select Cloudera SQL Stream Builder from the list of services.
4. Go to the Configuration tab and set the following configuration properties:



**Tip:** You can type **python** in the filter field to narrow down the configuration properties.

- a. Python Client Executable (ssb.python.client.executable): the path of the Python interpreter used to launch the Python process when submitting the Python jobs via flink run or compiling the jobs containing Python UDFs. For example /usr/bin/python3
  - b. Python Executable (ssb.python.executable): the path of the Python interpreter used to execute the python UDF worker. For example /usr/bin/python3
  - c. Python UDF Reaper Period (ssb.python.udf.reaper.period.seconds): the interval (in seconds) between two Python UDF Reaper runs, which deletes the Python files of the terminated jobs from the artifact storage.
  - d. Python UDF Time To Live (ssb.python.udf.ttl): The minimum lifespan (in milliseconds, seconds, minutes, or hours) of a Python UDF in the artifact storage. After this the Python UDF Reaper can delete the Python files from the artifact storage.
  - e. Select Enable Python UDFs in SSB and click on the checkbox.
5. Enter a reason for change and click Save Changes.
  6. Navigate to Instances.
  7. Select the Streaming SQL Engine from the list. (Click the checkbox on the left.)
  8. Click Restart from the Actions for selected dropdown menu.



**Important:** The following Python versions are supported in Cloudera SQL Stream Builder:

- 3.8
- 3.9
- 3.10

### To use Python UDFs:

#### Procedure

1. Navigate to the Streaming SQL Console.
  - a) Go to your cluster in Cloudera Manager.
  - b) Select SQL Stream Builder from the list of services.
  - c) Click SQLStreamBuilder Console .  
The **Streaming SQL Console** opens in a new window.
- a) Navigate to Management Console Environments , and select the environment where you have created your cluster.
- b) Select the Streaming Analytics cluster from the list of Data Hub clusters.
- c) Select Streaming SQL Console from the list of services.

The **Streaming SQL Console** opens in a new window.

2. Open a project from the **Projects** page of Streaming SQL Console.
  - a) Select an already existing project from the list by clicking the Open button or Switch button.
  - b) Create a new project by clicking the New Project button.
  - c) Import a project by clicking the Import button.

You are redirected to the **Explorer** view of the project.

3.



Click next to **Functions**.

## 4. Click New Function.

5. From the dropdown menu select PYTHON.
6. Add a Name to the UDF.  
For example, name the UDF to ADD\_FUNCTION.
7. Add a Description to the UDF. (Optional.)
8. Paste the Python code to the editor.  
For example, a simple addition:

```
from pyflink.table.udf import udf
from pyflink.table import DataTypes

@udf(result_type=DataTypes.BIGINT())
def udf_function(i, j):
    return i + j
```



**Important:** The name of the function always has to be udf\_function.

9. Click Create.
10. Once created, you can use the new User Defined Function in your SQL statement or as a computed column when creating a table.  
For example:

```
-- as a SQL statement
SELECT ADD_FUNCTION(myTable.n1, myTable.n2)

-- as a computed column
CREATE TABLE myTable (
```

```
`number_a` BIGINT,  
`number_b` BIGINT  
`sum` AS ADD_FUNCTION(number_a, number_b) -- evaluate expression and s  
apply the result to queries  
) WITH (  
  'connector' = 'kafka'  
  ...  
) ;
```

## Results

When a Flink job starts, Cloudera SQL Stream Builder will upload the Python file to the Artifact Storage, accessible for Flink to execute the UDF when called. The `ssb.python.udf.reaper.period.seconds` and `ssb.python.udf.ttl` configuration properties (set in Cloudera Manager) control the behavior of SSB to remove Python files associated with terminated jobs.

For more information on using Python UDFs, refer to the [Apache Flink documentation](#).

## Using System Functions

The same set of system functions can be used for Cloudera SQL Stream Builder as for Apache Flink.

As SSB runs on Flink, the following built-in system functions can also be used for data transformations in your SQL Jobs.

- [Comparison Functions](#)
- [Logical Functions](#)
- [Arithmetic Functions](#)
- [String Functions](#)
- [Temporal Functions](#)
- [Conditional Functions](#)
- [Type Conversion Functions](#)
- [Collection Functions](#)
- [JSON Functions](#)
- [Value Construction Functions](#)
- [Value Access Functions](#)
- [Grouping Functions](#)
- [Hash Functions](#)
- [Aggregate Functions](#)
- [Column Functions](#)

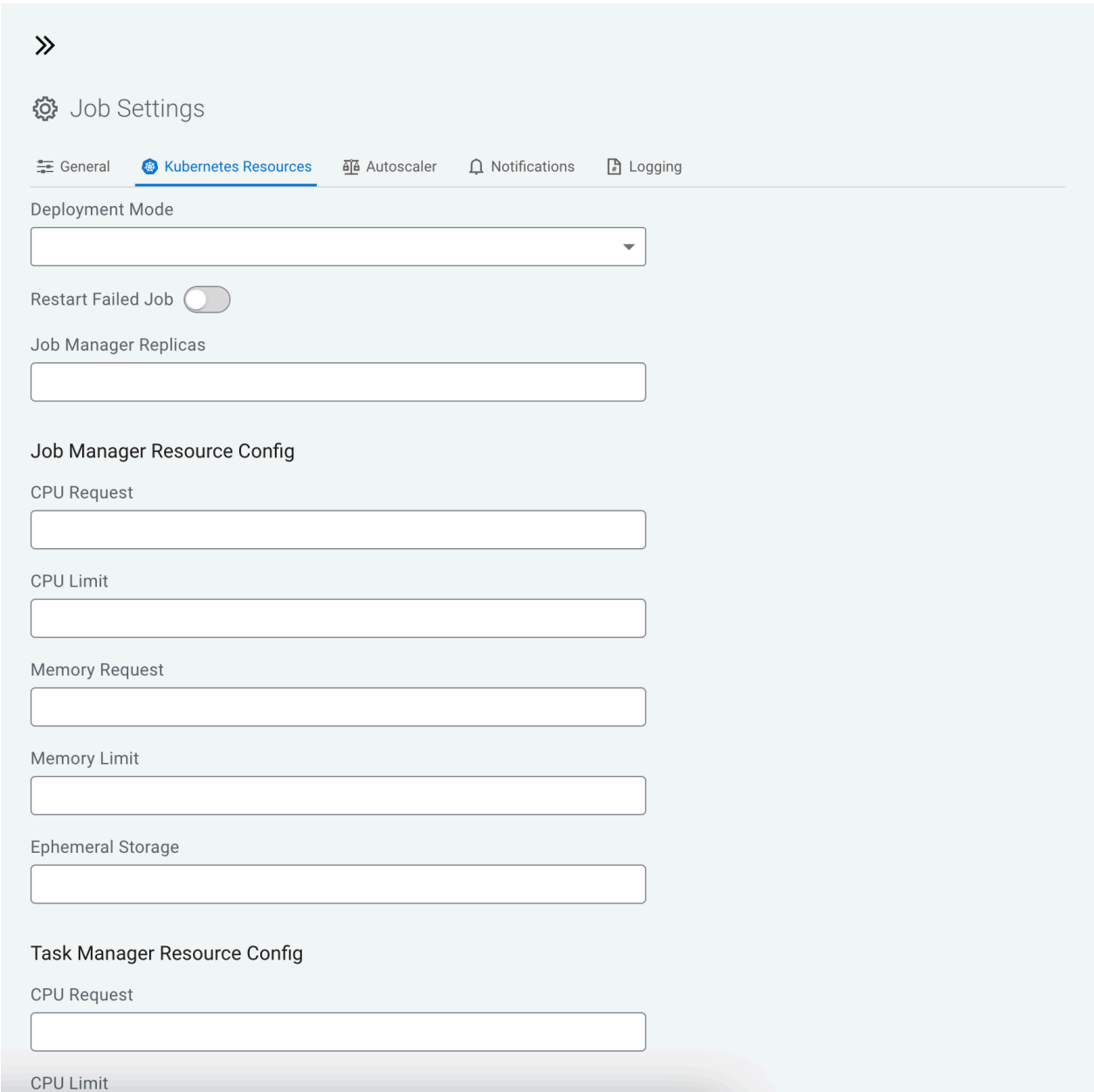
For more information about the list of supported Functions, see the [Apache Flink documentation](#).

## Job settings

Configure jobs before launching on the Cloudera SQL Stream Builder.

Before launching a new SQL job, you can set the following configurations for your job on Streaming SQL Console on the Kubernetes Resources tab:

- JobManager CPU, memory and ephemeral storage
- JobManager replicas (for high availability)
- TaskManager CPU, memory and ephemeral storage
- Flink deployment mode (*native* or *standalone*)



The screenshot shows the 'Job Settings' configuration page for the Cloudera Streaming Analytics Kubernetes Operator. The 'Kubernetes Resources' tab is selected, showing various resource configuration fields. The 'Deployment Mode' is a dropdown menu. The 'Restart Failed Job' option is a toggle switch. The 'Job Manager Replicas' is a text input field. The 'Job Manager Resource Config' section includes fields for 'CPU Request', 'CPU Limit', 'Memory Request', 'Memory Limit', and 'Ephemeral Storage'. The 'Task Manager Resource Config' section includes fields for 'CPU Request' and 'CPU Limit'.

>>

Job Settings

General **Kubernetes Resources** Autoscaler Notifications Logging

Deployment Mode

Restart Failed Job ☐

Job Manager Replicas

Job Manager Resource Config

CPU Request

CPU Limit

Memory Request

Memory Limit

Ephemeral Storage

Task Manager Resource Config

CPU Request

CPU Limit

You also have the option to configure the job related settings on the General tab, and adjust the autoscaler configurations on the Autoscaler tab.



**Important:** For session-mode jobs, these configurations must be set before starting the session cluster. If you modify the default configurations, you need to shut down the existing session cluster and start the job for the changes to take effect.

## Sampling

How to enable sampling for your SQL queries.

To enable sampling, you create a secret with the parameters of the Kafka instance used for sampling before starting helm install.



Example for non-secure setup:

```
kubectl create secret generic ssb-sampling-kafka -n flink \
  --from-literal=SSB_SAMPLING_BOOTSTRAP_SERVERS=kafka.example.com:9092 \
  --from-literal=SSB_SAMPLING_SECURITY_PROTOCOL=PLAINTEXT
```

Example for secure setup:

```
kubectl create secret generic ssb-sampling-kafka -n flink \
  --from-literal=SSB_SAMPLING_BOOTSTRAP_SERVERS=kafka-ssl.example.com:9092 \
  --from-literal=SSB_SAMPLING_SECURITY_PROTOCOL=SSL \
  --from-file=sampling_kafka_truststore.jks=[*** YOUR PATH ***]/truststore.jks \
  --from-literal=SSB_SAMPLING_TRUSTSTORE_PASSWORD=[*** PASSWORD ***]
```

In the values.yaml file, set sampling enabled to true and set your secret.

```
ssb:
  sampling:
    enabled: true
    secure: true
    secretRef: ssb-sampling-kafka
```

## Sample results

How to view sample results from your SQL queries.

Because Cloudera Streaming Analytics - Kubernetes Operator does not install Kafka, in the Cloudera SQL Stream Builder UI you are not able to see any rows from the Flink jobs. To see sampled results from your SQL queries, you need to have a Kafka cluster installed and accessible by both Cloudera SQL Stream Builder and Flink pods, and change ssbConfiguration to configure Cloudera SQL Stream Builder to use Kafka for data sampling:

```
ssbConfiguration:
  application.properties: |+
  kafka.enabled=true
  spring.kafka.bootstrap-servers=example-kafka:9092
  spring.kafka.jaas.enabled=false
  spring.kafka.properties.security.protocol=PLAINTEXT
```

## Connectors

Cloudera SQL Stream Builder supports different connector types and data formats for Flink SQL tables to ease development and access to all kinds of data sources.

The following table summarizes the supported connectors and how they can be used in Cloudera SQL Stream Builder:

Connector	Type	Description
Kafka	source/sink	Supported as exactly-once-sink
Hive	source/sink	Can be used as catalog
Kudu	source/sink	Can be used as catalog
Schema Registry	source/sink	Can be used as catalog


Connector	Type	Description
JDBC	source/sink	Can be used with Flink SQL. PostgreSQL, MySQL and Hive are supported.
Filesystems	source/sink	Filesystems such as HDFS, S3 and so on. Can be used with Flink SQL
Debezium CDC	source	Can be used with Flink SQL. PostgreSQL, MySQL, Oracle DB, Db2 and SQL Server are supported.
Webhook	sink	Can be used as HTTP POST/PUT with templates and headers
PostgreSQL	sink	Materialized View connection for reading views. Can be used with anything that reads PostgreSQL wire protocol
REST	sink	Materialized View connection for reading views. Can be used with anything that reads REST (such as notebooks, applications, and so on)
BlackHole	sink	Can be used with Flink SQL.


## Using connectors with templates

Some of the connectors have default templates in Streaming SQL Console that allows you to create tables with them easily. These predefined templates not only contain the CREATE TABLE statement, but every mandatory and optional argument that is needed for the table.

### Procedure

- Navigate to the Streaming SQL Console.
  - Go to your cluster in Cloudera Manager.
  - Select SQL Stream Builder from the list of services.
  - Click SQLStreamBuilder Console .  
The **Streaming SQL Console** opens in a new window.
  - Navigate to Management Console Environments , and select the environment where you have created your cluster.
  - Select the Streaming Analytics cluster from the list of Data Hub clusters.
  - Select Streaming SQL Console from the list of services.  
The **Streaming SQL Console** opens in a new window.
- Open a project from the **Projects** page of Streaming SQL Console.
  - Select an already existing project from the list by clicking the Open button or Switch button.
  - Create a new project by clicking the New Project button.
  - Import a project by clicking the Import button.

You are redirected to the **Explorer** view of the project.
- 

Click  next to **Jobs** from the **Explorer**.
- Select New Job > Create .  
The SQL Editor for the created job opens in a tab.
- Click Templates.
- Select one of the connector templates.  
The corresponding CREATE TABLE statement is imported to the **SQL Editor**.



### What to do next

After importing the template to the SQL Editor, you need to provide the mandatory properties and you can also fill out the optional arguments if needed. When you click Execute, the table of the chosen connector is created and listed under **Virtual Tables**.

## Adding new connectors

When adding new connectors to Cloudera SQL Stream Builder, you need to specify the property list, data types and must upload the connector JAR file to the Console.

### Procedure

1. Navigate to the Streaming SQL Console.
  - a) Go to your cluster in Cloudera Manager.
  - b) Select SQL Stream Builder from the list of services.
  - c) Click SQLStreamBuilder Console .  
The **Streaming SQL Console** opens in a new window.
  - a) Navigate to Management Console Environments , and select the environment where you have created your cluster.
  - b) Select the Streaming Analytics cluster from the list of Data Hub clusters.
  - c) Select Streaming SQL Console from the list of services.  
The **Streaming SQL Console** opens in a new window.
2. Open a project from the **Projects** page of Streaming SQL Console.
  - a) Select an already existing project from the list by clicking the Open button or Switch button.
  - b) Create a new project by clicking the New Project button.
  - c) Import a project by clicking the Import button.  
You are redirected to the **Explorer** view of the project.
3. Open **External Resources** from the **Explorer** view.
4.  
  
Click  next to **Connectors**.

## 5. Click New Connector.

## 6. Provide a name for the connector as Type.

## 7. Select a data format to be supported for the connector from the Supported Formats.

## 8. Upload the connector JAR file.

## 9. Click Properties to add properties to the connector.

- a) Add a name to the property.
- b) Add a default value to the property.
- c) Add a description to the property.
- d) Click Required to make a property mandatory.
- e) Click Add to specify more properties.

You can specify as many properties as needed for the connector.

## 10. Click Create.

The newly added connector is listed under the **Connectors**.

## Kafka connectors

When using the Kafka connector, you can choose between using an internal or external Kafka service. Based on the connector type you choose, there are mandatory fields where you must provide the correct information.

You can choose from the following Kafka connectors when creating a table in Streaming SQL Console:

**Template: local-kafka**

Automatically using the Kafka service that is registered in the Data Sources, and runs on the same cluster as the Cloudera SQL Stream Builder service. You can choose between JSON, Avro and CSV data types.

Type: source/sink

The following fields are mandatory to use the connector:

- scan.startup.mode: Startup mode for the Kafka consumer. group-offsets is the default value. You can choose from earliest-offset, latest-offset, timestamp and specific-offsets as startup mode.

- **topic:** The topic from which data is read as a source, or the topic to which data is written to. No default value is specified. You can also add a topic list in case of sources. In this case, you need to separate the topics by semicolon. You can only specify the topic-pattern or topic for the sources.
- **format:** The format used to deserialize and serialize the value part of Kafka messages. No default value is specified. You can use either the format or the value.format option.

**Template: kafka**

Using an external Kafka service as a connector. To connect to the external Kafka service, you need to specify the Kafka brokers that are used in your deployment.

Type: source/sink

The following fields are mandatory to use the connector:

- **properties.bootstrap.servers:** Specifying a list of Kafka brokers that are separated by comma. No default value is specified.
- **topic:** The topic from which data is read as a source, or the topic to which data is written to. No default value is specified. You can also add a topic list in case of sources. In this case, you need to separate the topics by semicolon. You can only specify the topic-pattern or topic for the sources.
- **format:** The format used to deserialize and serialize the value part of Kafka messages. No default value is specified. You can use either the format or the value.format option.

**Template: upsert-kafka**

Connecting to a Kafka service in the upsert mode. This means that when using it as a source, the connector produces a changelog stream, where each data record represents an update or delete event. The value in the data records is interpreted as an update of the last value for the same key. When using the table as a sink, the connector can consume a changelog stream, and write insert/update\_after data as normal Kafka message value. Null values are represented as delete. For more information about the upsert Kafka connector, see the [Apache Flink documentation](#).

Type: source/sink

- **properties.bootstrap.servers:** Specifying a list of Kafka brokers that are separated by comma. No default value is specified.
- **topic:** The topic from which data is read as a source, or the topic to which data is written to. No default value is specified. You can also add a topic list in case of sources. In this case, you need to separate the topics by semicolon. You can only specify the topic-pattern or topic for the sources.
- **key.format:** The format used to deserialize and serialize the key part of Kafka messages. No default value is specified. Compared to the regular Kafka connector, the key fields are specified by the PRIMARY KEY syntax.
- **value.format:** The format used to deserialize and serialize the value part of Kafka messages. No default value is specified. You can use either the format or the value.format option.

**Configuring deserialization policy in DDL**

You can configure every supported type of Kafka connectors (local-kafka, kafka or upsert) how to handle if a message fails to deserialize which can result in job submission error. You can choose from the following configurations:

**Fail**

In this case an exception is thrown, and the job submission fails

**Ignore**

In this case the error message is ignored without any log, and the job submission is successful

**Ignore and Log**

In this case the error message is ignored, and the job submission is successful

**Save to DLQ**

In this case the error message is ignored, but you can store it in a dead-letter queue (DLQ) Kafka topic

1. Choose one of the Kafka template types from Templates.
2. Select any type of data format.

The predefined CREATE TABLE statement is imported to the SQL Editor.

3. Fill out the Kafka template based on your requirements.
4. Search for the `deserialization.failure.policy`.
5. Provide the value for the error handling from the following options:
  - a. 'error'
  - b. 'ignore'
  - c. 'ignore\_and\_log'
  - d. 'dlq'

If you choose the `dlq` option, you need to create a dedicated Kafka topic where you store the error message. In this case, you must provide the name of the created DLQ topic for the `deserialization.failure.dlq.topic` property.

6. Click Execute.

For more information about how to configure the error handling using the Kafka wizard, see the [Deserialization tab](#) [Deserialization tab](#) section.

## CDC connectors

You can use the Debezium Change Data Capture (CDC) connector to stream changes in real-time from MySQL, PostgreSQL, Oracle, Db2, SQL Server and feed data to Kafka, JDBC, the Webhook sink or Materialized Views using Cloudera SQL Stream Builder.

### Concept of Change Data Capture

Change Data Capture (CDC) is a process to capture changes in a source system, and update the data within a downstream system or application with the changes.

The Debezium implementation offers CDC with database connectors from which real-time events are updated using Kafka and Kafka Connect. Debezium captures every row-level change in each database table of an event stream. Applications read these streams to see the change events in the same order as they occurred. The change events are routed to a Kafka topic from which Kafka Connect feeds the records to other systems and databases.

For more information about Debezium, see the [official Debezium documentation](#).

CDC in Cloudera Streaming Analytics does not require Kafka or Kafka Connect as Debezium is implemented as a library within the Flink runtime. This means that the captured changes are propagated downstream to any connector that Flink supports. Cloudera Streaming Analytics allows queries to be issued at change data capture time, which means filtering, grouping, joining, and so on, can be performed on the change stream as it comes from the source database.

For more information about the Flink implementation of Debezium, see the [official Apache Flink documentation](#).

From the supported set of Debezium connectors, MySQL, PostgreSQL, Oracle, Db2, and SQL Server are supported in Cloudera Streaming Analytics.



**Note:** You need to configure the databases, users and permissions for the supported connectors before you are able to use them in Cloudera SQL Stream Builder. For more information about setting up the databases, see the [official Debezium documentation](#).

## JDBC connector

When using the JDBC connector, you can choose between using a PostgreSQL, MySQL, Oracle, Hive, Db2 or SQL Server databases. Based on the connector type you choose, there are mandatory fields where you must provide the correct information.

**Template: jdbc**

Using either PostgreSQL, MySQL, Oracle, Hive, Db2 or SQL Server as databases. When you use the JDBC connector, you must specify the JDBC database which are going to be used for the connection.

Type: source/sink

The following fields are mandatory to use the connector:

- url: The URL of the JDBC database. No default value is specified.
- table-name: The name of the JDBC table in the database that you need to connect to. No default value is specified.

## Filesystem connector

When using the Filesystem connector, you can choose between HDFS, S3 and so on storage systems. Based on the connector type you choose, there are mandatory fields where you must provide the correct information.

**Template: filesystem**

Using either HDFS, S3 or any type of storage system. When you use the Filesystem connector, you must specify the path to the file system which are going to be used for the connection.

Type: source/sink

The following fields are mandatory to use the connector:

- path: Path to the root directory of the table data. No default value is specified.
- format: The format used for the file system. No default value is specified.

## Webhook connector

The Webhook connector can be used as HTTP POST/PUT with templates and headers.

To use the webhook connector with [Python UDFs](#):

1. Create your UDF using the SSB UI. Example:

```
from pyflink.table.udf import udf
from pyflink.table import DataTypes
from pyflink.table import Row
@udf(result_type=DataTypes.ROW([
    DataTypes.FIELD("transactionId", DataTypes.STRING()),
    DataTypes.FIELD("ts", DataTypes.STRING()),
    DataTypes.FIELD("accountId", DataTypes.STRING()),
    DataTypes.FIELD("amount", DataTypes.STRING())
]))
def udf_function(transactionId, ts, accountId, amount):
    if amount % 2 != 0:
        processed_amount = amount * 3
        return Row(str(transactionId), str(ts), str(accountId), str(processed_amount))
```

2. Create a webhook table. Refer to *Creating Webhook tables* for more information.

3. Write into the webhook table using the UDF in the query's WHERE statement. Example:

```
INSERT INTO %s
SELECT transactionId, ts, accountId, amount
FROM (
    SELECT transaction(transactionId, ts, accountId, amount)
    FROM %s"
) AS udf_results
WHERE transactionId IS NOT NULL
    AND ts IS NOT NULL
    " AND accountId IS NOT NULL "
    " AND amount IS NOT NULL",
destinationName, sourceName);
```

## Creating Webhook tables


You can configure the webhook table to perform an HTTP action per message (default) or to create code that controls the frequency (for instance, every N messages). When developing webhook sinks, it is recommended to check your webhook before pointing at your true destination.

### Procedure

1. Navigate to the Streaming SQL Console.
2. Open a project from the **Projects** page of Streaming SQL Console.
  - a) Select an already existing project from the list by clicking the Open button or Switch button.
  - b) Create a new project by clicking the New Project button.
  - c) Import a project by clicking the Import button.

You are redirected to the **Explorer** view of the project.

- 3.

Click  next to **Virtual Tables**.



#### 4. Select New Webhook Table .

The **Webhook Table** window appears.

**Webhook Table**

Table Name \*

Connector: **webhook2**

HTTP Endpoint \*

HTTP Method \*: **PUT**

Max Retries: **4**

Initial Expiry: **1s**

Back-off Rate: **3**

☒ SSL Validation

☐ Ignore Failed Records

**Request Template** | Http Headers

```

1 {
2   "incident":{
3     "type":"incident",
4     "title":"${icao} is too high!",
5     "body":{
6       "type":"incident_body",
7       "details":"Airplane with id ${icao} has reached an altitude of ${altitude} meters."
8     }
9   }
10 }

```

Create Cancel



**Important:** In the Connector menu, select **webhook2**. The v1 connector is deprecated and will be removed in an upcoming release of Cloudera Streaming Analytics - Kubernetes Operator.

#### 5. Provide a name to the Table.

#### 6. Enter an HTTP endpoint. The endpoint must start with `http://` or `https://`.



**Note:** You can use **hookbin** for testing of the webhook sink. Paste the **hookbin** endpoint into the text field, and inspect the output on the **hookbin** site. Once you have the right output result, then point it at your final endpoint.

#### 7. Add a Description about the webhook sink.

#### 8. Select POST or PUT in the HTTP Method select box.

#### 9. Choose to Disable SSL Validation, if needed.

#### 10. Enable Request Template, if needed.

a) If you selected Yes, then the template defined in the Request Template tab is used for output.

This is useful if the service you are posting requires a particular data output format. The data format must be a valid JSON format, and use `"${columnname}"` to represent fields. For example, a template for use with Pagerduty looks like this:

```

{
  "incident":{
    "type":"incident",
    "title":"${icao} is too high!",

```

```

      "body": {
        "type": "incident_body",
        "details": "Airplane with id ${icao} has reached an altitude of
${altitude} meters."
      }
    }
  }
}

```

11. In the Code editor, you can specify a code block that controls how the webhook displays the data.

For a webhook that is called for each message the following code is used:

```

// Boolean function that takes entire row from query as Json Object
function onCondition(rowAsJson)
{return true; // return false here for no-op, or plug in custom
  logic}
onCondition($p0)

```



**Note:** The rowAsJson is the result of the SQL Stream query being run in the {"name":"value"} format.

12. Add HTTP headers using the HTTP Headers tab, if needed.

Headers are name:value header elements. For instance, Content-Type:application/json, etc.

13. Click Create.

## Results

The Webhook table is ready to be used after the INSERT INTO statement in your SQL query.

## Datagen connector

The Datagen connector can be used as a source to generate random data. The Datagen connector works out of the box, no mandatory field is required to use the connector. The Datagen connector is useful when you want to experiment and try out Cloudera SQL Stream Builder or to test your SQL queries using random data.

### Template: datagen

You do not need to provide any type of information when using the Datagen connector and template.

Type: source

## Faker connector

The Faker connector can be used as a source to generate random data. To use the Faker connector, you need to specify the usage The Datagen connector is useful when you want to experiment and try out Cloudera SQL Stream Builder or to test your SQL queries using random data.

### Template: faker

You do not need to provide any type of information when using the Datagen connector and template.

Type: source

The following fields are mandatory to use the connector:

- `fields.#.expression`: The Java Faker expression to generate the values for a specific field. For more information about the list and use of the Faker expressions, see the [flink-faker documentation](#).

## Blackhole connector

The Blackhole connector can be used as a sink where you can write any type of data into. The Blackhole connector works out of the box, no mandatory field is required to use the connector. The Blackhole connector is useful when you want to experiment and try out Cloudera SQL Stream Builder or to test your SQL queries using random data.

**Template: blackhole**

You do not need to provide any type of information when using the Blackhole connector and template.

Type: sink