

SQL Stream Builder

Date published: 2019-12-17

Date modified: 2023-02-23



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

Running a simple SQL job.....	5
Project structure and development.....	6
Creating a project.....	7
Navigating in a project.....	9
Managing member of a project.....	9
Source control of a project.....	10
Masking information before using source control.....	11
Setting the environment for a project.....	12
Importing a project.....	13
Registering Data Sources in SSB.....	14
Adding Kafka Data Source.....	14
Adding Catalogs.....	17
Connectors.....	24
Using connectors with templates.....	24
Adding new connectors.....	25
Kafka connectors.....	26
CDC connectors.....	28
JDBC connector.....	29
Filesystem connector.....	29
Datagen connector.....	29
Faker connector.....	29
Blackhole connector.....	30
Data formats.....	30
Adding data formats.....	30
Concept of tables in SSB.....	31
Creating Kafka tables using wizard.....	32
Configuring Kafka tables.....	34
Creating Webhook tables.....	44
Creating Flink tables using Templates.....	46
SQL jobs.....	47
Creating and naming SQL jobs.....	47
Running SQL Stream jobs.....	48
Configuring SQL job settings.....	51
Adjusting logging configuration in Advanced Settings.....	53
Configuring YARN queue for SQL jobs.....	56
Managing session for SQL jobs.....	56
Executing SQL jobs in production mode.....	58

Functions.....	58
Creating Javascript User-defined Functions.....	59
Developing JavaScript functions.....	60
Creating Java User-defined functions.....	61
Using System Functions.....	62
 Introduction to Materialized Views.....	 62
Creating Materialized Views.....	63
Configuring Retention Time for Materialized Views.....	69
Materialized View Pagination.....	71
Using Dynamic Materialized View Endpoints.....	71
Using SQL Stream Builder with Cloudera Data Visualization.....	73
 Using SQL job notification.....	 74
 Using SQL Stream Builder REST API.....	 79
 Monitoring SQL Stream jobs.....	 81
Collecting diagnostic data.....	83
 SSB metadata collection using Atlas.....	 84
 Flink SQL.....	 84
Flink DDL.....	84
Managing time in SSB.....	84
Flink DML.....	85
Flink Queries.....	86
Other supported statements.....	86
Data Types.....	87
Dynamic SQL Hints.....	87
SQL Examples.....	88
 Enriching streaming data with join.....	 90
Joining streaming and bounded tables.....	91
Example: joining Kafka and Kudu tables.....	92

Running a simple SQL job

You can use this Getting Started use case to get familiar with the most simple form of running a SQL Stream job.

About this task

The Getting Started contains the basic steps of running a SQL Stream job. When executing the job, you do not need to select a sink as the results are displayed in the browser. SQL Stream Builder provisions a job on your cluster to run the SQL queries. You can select the **Logs** tab to review the status of the SQL job. As data is returned, it shows up in the **Results** tab.

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.
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 **Jobs** from the **Explorer**.

4. Select New Job > Create .

The SQL Editor for the created job opens in a tab.

5. Select *datagen* from the Templates.

The CREATE TABLE statement is imported to the SQL window.

6. Add a SELECT query to the SQL window after the datagen template.

```
SELECT * FROM <table_name>
```

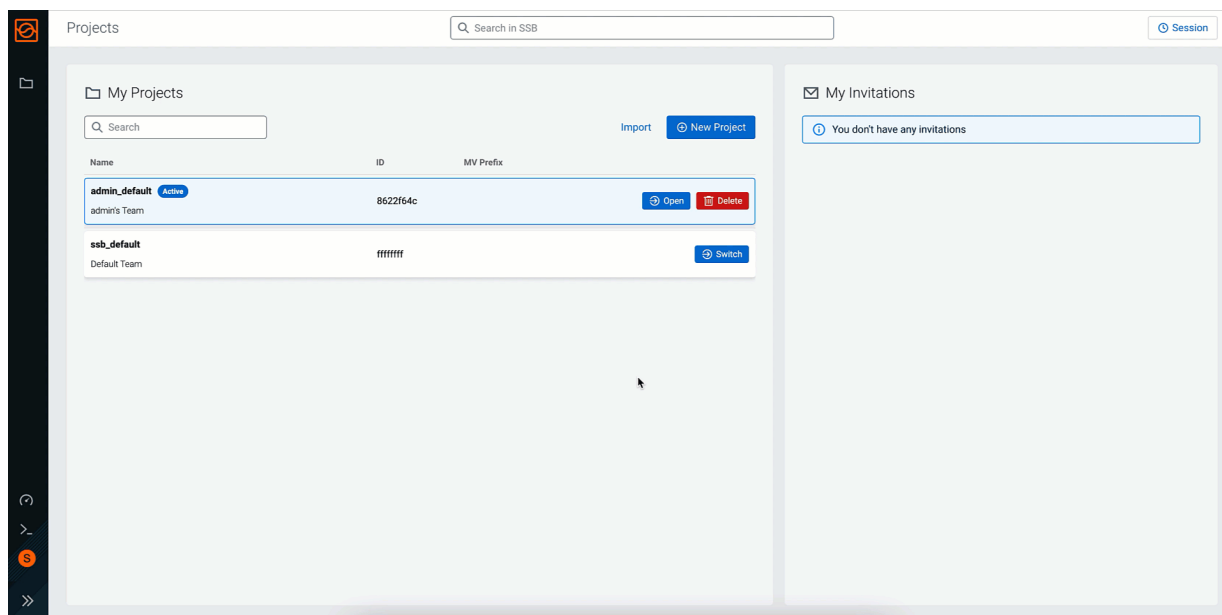


Note: When adding SQL statements to the SQL window, you do not need to add the semicolon (;) at the end of the statement as SSB can run the command without the semicolons.

7. Click Execute.

You can see the generated output on the Results tab.

8. Click on the Stop button to stop the job.



Project structure and development

Projects aim to provide a Software Development Lifecycle (SDLC) for streaming applications in SQL Stream Builder (SSB): they allow developers to think about a task they want to solve using SSB, and collect all related resources, such as job and table definitions or data sources in a central place.

Projects aim to facilitate collaboration between developers by sharing common resources among its members. Projects can be synchronized with a Git repository, allowing easy migration between different clusters. The environment concept in a project allows the templating of cluster-specific or sensitive properties.

A project is a collection of resources, static definitions of data sources, jobs with materialized views, virtual tables, user-defined functions (UDF), and materialized view API keys. These resources are called internal to a project and can be safely used by any job within the project.

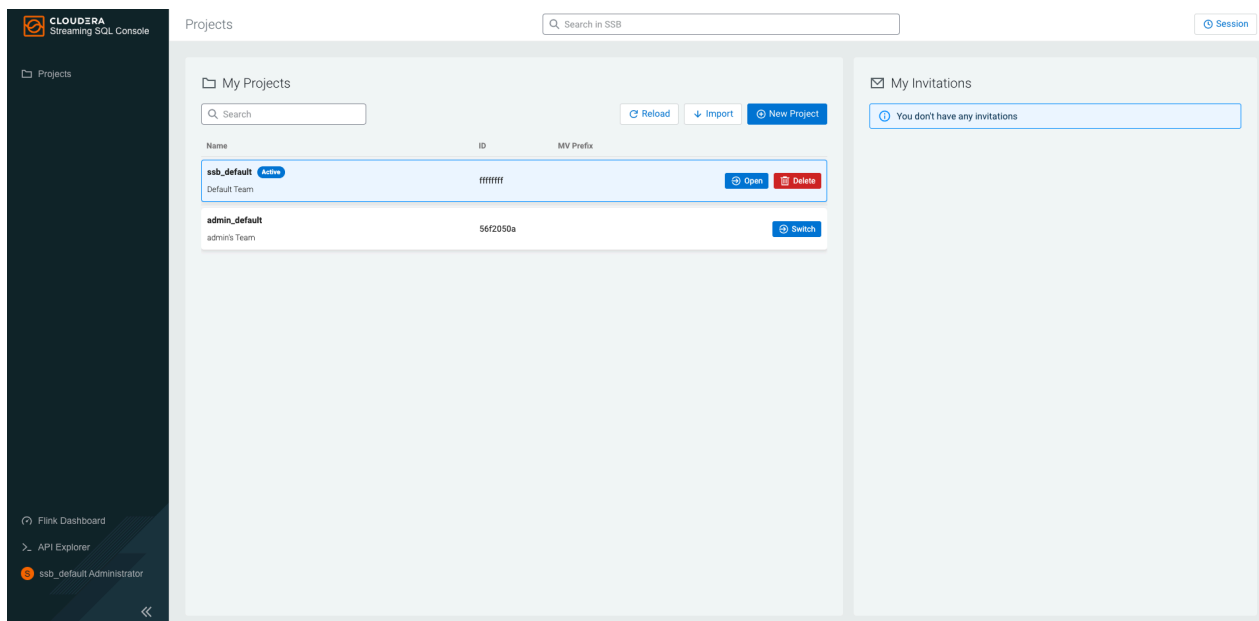
Jobs can also use external resources that the logged in user has access to. These resources are defined in other projects (for example, a UDF or a virtual table in another project), or can come from a Data Source (for example, a Kudu table from the Kudu catalog). External resources are outside the scope of the project, thus they can be changed or altered by external factors. It is recommended to use only internal resources for projects that are intended to be exported.

The following table summarizes the concepts and structural elements of projects in SSB:

Project specific resources	
Jobs	SSB job definitions including SQL, Materialized View configuration and endpoints, Job settings such as checkpointing and parallelism. The status of the job and its associated Flink job (if any) are not part of the definition.
Functions	User-defined JavaScript functions.
Virtual Tables	Virtual tables of the project. These are stored in the <code><project_name></code> database of the ssb catalog. When exporting a project, only these tables (<code>ssb.<project_name>.*</code>) are exported.
Data Sources	Data Sources include Kafka providers and Catalogs in the project. While the definition of the Data Source is considered internal to the project, the tables or topics provided by them are external, as they are managed by an outside system.

Materialized Views	Materialized Views that have been defined for Jobs in the project.
API Keys	API Keys for accessing Materialized View endpoints.
Job Notifications	Job Notification actions (webhook and email notifications) as well as notification groups can be defined in a project. They may be assigned to Jobs to be triggered by failure. Job notifications are not synchronized when using the Git import/export feature.
External resources	
Virtual Tables	Virtual tables from other projects and catalogs that the logged in user has access to. Other members of the active project might not have access to these.
Connectors	Connectors available in SSB for connecting to external systems. These are shared among the whole SSB instance, and are accessible to all users in all projects.
Data Formats	Data Formats available in SSB that can be used by connectors when connecting to external systems. These are shared among the whole SSB instance, and are accessible to all users in all projects.

When you access the Streaming SQL Console, there are already existing projects on the **My Projects** page. The `ssb_default` and `[***USERNAME***]_default` projects are automatically generated projects. The `ssb_default` project and its resources are visible to every user, while the `[***USERNAME***]_default` project and its resources are only visible to that user. Every user who created an account in SSB has its own project generated by default where members can be invited.



Creating a project

Before you can use Streaming SQL Console, you need to create a project where you can submit your SQL jobs and manage your project resources.

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.

2. Click New Project on the **Projects** page of Streaming SQL Console.

The **Create Project** window appears.

3. Provide a Name to the project.
4. Optional: Provide a Description to the project.
5. Optional: Provide a prefix that overrides the Materialized View table names.

Providing a prefix to the Materialized View tables names that overrides the default one is an advanced option.

The tables are prefix by default with the ID of the projects to avoid collision between projects. This configuration allows setting a custom prefix to the Materialized View tables created in the project.

6. Optional: Provide the **Source Settings** of the project.

- a) Add a remote Github repository URL to the **Clone URL** field.

You can use the following example URLs with or without authentication:

- To clone without authentication: `https://github.com/cloudera/ssb-examples.git`
- To clone and push with HTTP basic authentication (username/password): `https://github.com/cloudera/ssb-examples.git`

In case of using basic authentication, the personal access token needs to be provided as password. For more information about creating personal access tokens, see the [Github documentation](#).

- To clone and push with SSH authentication (SSH private key): `git@github.com:cloudera/ssb-examples.git`

In case of using SSH authentication, you need to generate the private key using the following command:

```
ssh-keygen -t ecdsa -b 256 -m pem -C "name@example.com"
```

As not every SSH private key is supported, you can use the following command to convert your key to the acceptable format:

```
ssh-keygen -m pem -p -f [***PATH_TO_PRIVATE_KEY***]
```

- To clone and push a local git repository on the filesystem of SSB host: `file:///usr/src/ssb-examples/`
- b) Provide the branch name or tag of the Github repository to the **Branch** field. This branch is checked out when pulling or pushing to the Github repository.



Warning:

When exporting a project, any sensitive information in CREATE TABLE statements, data sources, or SQL jobs will not be redacted. Ensure that usernames and passwords are hidden with the environment variables where required. For more information, see the [Source control of a project](#) documentation.

7. Enable or disable Allow deletions on import.

Enabling the allow deletion upon import allows you to hard-reset your project to its versioned state when there are untracked resources in it. For example, the job is deleted from the project when importing the project from Github if the job does not exist in the repository. In other words, the upstream version overrides the version you have in Streaming SQL Console. If disabled, the project keeps the resources that were not pushed to the Github repository.

The deletion affects the jobs, data sources, functions, Materialized Views and their API keys.

8. Enable or disable Authentication for the project.

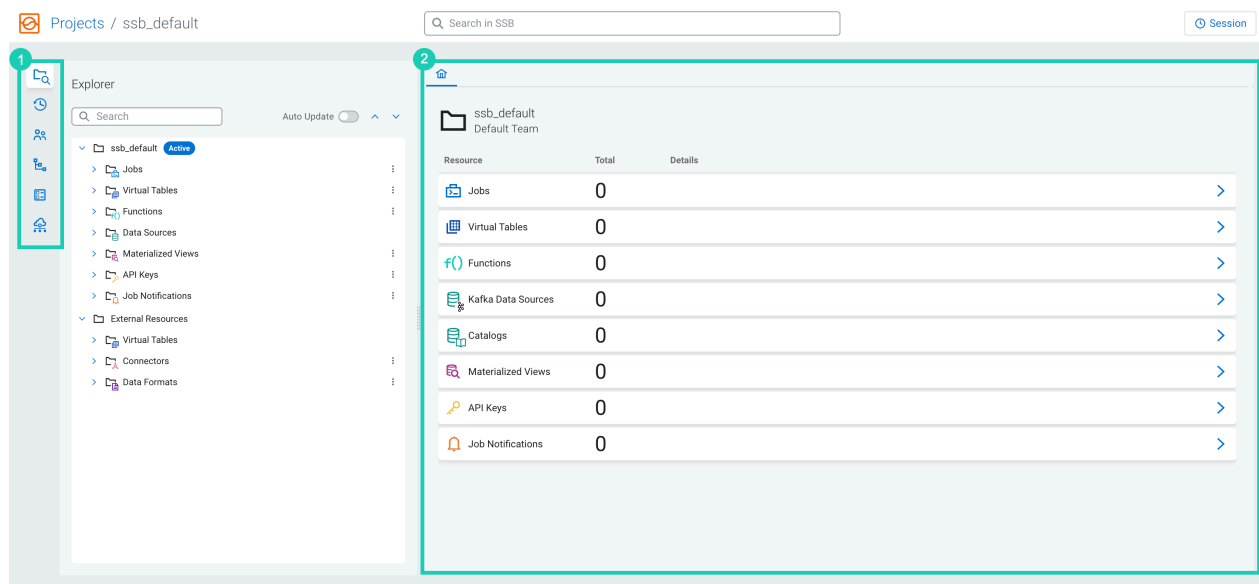
What to do next

After you have created the project, you have the following options to manage the project on the **Projects** page:

1. Click Open to use the created project and start creating SQL jobs.
2. Click Switch to open another project than the active one.
3. Click Delete to remove the project from SSB.

Navigating in a project

When you open a project in Streaming SQL Console, you can navigate between the Project Manager and the Tab view of the Project Resources.



1. The **Project Manager** consists of the following views:
 - **Explorer** - tree view of the user defined and external resources of the project
 - **SQL History** - history of the queries that have been executed by the logged in user
 - **Members** - list of the users that are invited in the project
 - **Source Control** - configure the synchronization source (Git repository) for the project, import or export changes
 - **Environments** - manage environment configurations, define variables for templating
 - **Data Hub Service Discovery** - automatic discovery of Kafka, Kudu and Schema Registry services in your CDP Public Cloud environment
2. The **Home** tab of a project allows you to quickly access the user defined resources of a project. It also gives you a quick view about the total number of resources in each category with resource specific details. You can open the different resource tabs easily from the **Home** tab to create and manage items in a given resource category.

Managing member of a project

Created or imported projects can be shared with other users in Streaming SQL Console. You can invite members using their Streaming SQL Console username and set the access level to member or administrator.

About this task

When you create a project, you automatically become the owner of that project. You can invite SSB users to your project and give them access to the jobs and resources. You can also set another user to be the admin of your project.

Table 1: Access levels in a project

Member	Creating and managing jobs and resources in a project, synchronizing the project, inviting other Members. Running jobs, stopping their own jobs.
Admin	In addition to Member privileges, Admins can stop jobs of other users, manage users of the project, invite other Admins or Members.
Owner	The privileges of owners are identical to the Admins, but it cannot be revoked and Owners cannot be removed from the project.

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 .
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.

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

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

3. Click Members from the Project Manager.
4. Click Invite User.
5. Provide the SSB Username of the member you want to invite.
6. Set the Access Level for the user.
7. Click Send Invitation.

The **Invite Member** window appears.

The sent invitation is visible on the **Projects** page of Streaming SQL Console under **My Invitations** or at the **Invitations** tab on the **Members** page.



You can delete any members of a project that you are an owner or an admin of using

button next to their username.

Source control of a project

When you provide a Git repository and branch for a project, you can push and pull your project to Git. This enables you to track your versions of your project, create a backup of your project in Git and manage the development lifecycle of your project.

You can provide the Git repository URL and branch for a project when you create one or when you import a project on the **Projects** page of Streaming SQL Console.



Note: The Source Control feature is in Technical Preview and not ready for production deployment. Cloudera encourages you to explore these features in non-production environments and provide feedback on your experiences through the *Cloudera Community Forums*.

After setting up your remote repository, you can navigate to Source Control in your project, and use the Import and Push buttons to synchronize your local state with the state in the repository.

Importing a version of project

When you import a project from Git to SSB in Source Control, the configured Git repository is cloned to a temporary directory by SSB, the specified branch is checked out, and the resource files found in the directory with the project's name are used to import resources to the SSB project.

Pushing a version of project

When you push a project from to Git in Source Control, the configured repository is cloned to a temporary directory by SSB, and the project resources are written to files in a directory with the project's name into the temporary directory.

Changes are added and committed with the specified commit message, and pushed to the configured remote.

Changing the project settings

You can manage the source information of a project under the **Source Settings** tab. You can change the **Clone URL** and **Branch** information of the Git repository. You can enable **Allow deletions** for importing a project, and configure the **Authentication** method that used when interacting with the repository.

Masking information before using source control

When exporting projects from SQL Stream Builder, possible sensitive parameters in CREATE TABLE statements, data sources, or the SQL job will not be redacted. To conceal sensitive information, environment variables can be used.

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.
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 Environments from the Project Manager.
4. Click New Environment.
5. Provide a Name for the environment.

For example, you can name it Dev.
6. Add keys and values as needed.

For example, you can add `psql_password` as key, and provide the password as value.
7. Click Activate to apply the specified information for the project.
8. When creating the table, reference the environment variable as shown in the following example:

```
CREATE TABLE `postgres_cdc_table_1682585103` (
  `order_id` INT,
  `city` VARCHAR(2147483647),
  `street_address` VARCHAR(2147483647),
  `amount` INT,
  `order_time` TIMESTAMP(3),
  `order_status` VARCHAR(2147483647)
) WITH (
  'connector' = 'postgres-cdc',
  'password' = '${ssb.env.psql_password}',
  'username' = '...
```

```
) ;
```

Setting the environment for a project

Environments are sets of user-defined variables that can be substituted into templates in your SSB resources. Environments are like configuration files for your projects that adapt it to be used in a specific environment. They enable you to keep the common resources and logic of a project in a central repository and reuse it across different environments, different clusters.

About this task

Creating an environment file for a project means that users can create a template with variables that could be used to store environment-specific configuration.

For example, you might have a development, staging and production environment, each containing different clusters, databases, service URLs and authentication methods. Projects and environments allow you to write the logic and create the resources once, and use template placeholders for values that need to be replaced with the environment specific parameters.

To each project, you can create multiple environments, but only one can be active at a time for a project. Environments can be exported to files, and can be imported again to be used for another project, or on another cluster.


While environments are applied to a given project, they are not part of the project. They are not synchronized to Git when exporting or importing the project. This separation is what allows the storing of environment-specific values, or configurations that you do not want to expose to the Git repository.

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.
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 Environments from the Project Manager.
4. Click New Environment.
5. Provide a Name for the environment.
Add keys and values as needed.
6. After creating an environment file, you have the following options to manage it:
 - a. Click Activate to apply the specified information for the project. To deactivate an active environment file, you need to click the Deactivate button.
 - b.



Click `Export` to export the environment file.
 - c. Click Import and search for an exported environment file on your computer to import an environment file.
 - d.



Click `Delete` to remove the environment file from the project.

What to do next

You can use a variable defined in your active environment in the following ways:

- As values in the properties section of a CREATE TABLE statement as shown in the following example:

```
CREATE TABLE (...) WITH ('key1'='${ssb.env.yourVariable}')
```

- In the properties of a Kafka data source, for example bootstrap servers, truststore location
- In the values of Catalog properties, for example Schema Registry URL, Kudu masters, Custom Catalog

To refer to an environment variable called *yourVariable*, you can use the following syntax: `${ssb.env.yourVariable}`. You can use multiple substitutions or combine literals and variable substitutions in the same value string as shown in the following example:

```
'some-connector-url' = 'https://${ssb.env.connector-host}:${ssb.env.connector-port}'
```

Besides environments, it is also possible to define additional variables using the `SET_VAR 'x' = 'y';` syntax in your queries. These do not become part of the environment and are not persisted, but live only in the current SQL session. They can be used for substitutions like environment variables, but without the `ssb.env` prefix as shown in the following example:

```
SET_VAR 'yourVariable' = 'someValue';
CREATE TABLE (...) WITH ('key1'='${yourVariable}')
```

You can reset a variable with the `RESET_VAR 'x';` command.

Importing a project

You can use a Git repository to import a SQL Stream Builder project that was previously created and exported using source control.

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.
2. Click Import Project on the **Projects** page of Streaming SQL Console.
The **Import Project** window appears.
3. Optional: Provide a prefix that overrides the Materialized View table names.
Providing a prefix to the Materialized View tables names that overrides the default one is an advanced option. The tables are prefix by default with the ID of the projects to avoid collision between projects. This configuration allows setting a custom prefix to the Materialized View tables created in the project.

4. Provide the Source Settings of the project.**a) Add a remote Github repository URL to the Clone URL field.**

You can use the following example URLs with or without authentication:

- To clone without authentication: `https://github.com/cloudera/ssb-examples.git`
- To clone and push with HTTP basic authentication (username/password): `https://github.com/cloudera/ssb-examples.git`
- To clone and push with SSH authentication (SSH private key): `git@github.com:cloudera/ssb-examples.git`
- To clone and push a local git repository on the filesystem of SSB host: `file:///usr/src/ssb-examples/`

b) Provide the branch name or tag of the Github repository to the Branch field. This branch is checked out when pulling or pushing to the Github repository.**5. Provide the name of the project to the Project field.****6. Enable or disable Allow deletions on import.**

Enabling the allow deletion upon import allows you to hard-reset your project to its versioned state when there are untracked resources in it. For example, the job is deleted from the project when importing the project from Github if the job does not exist in the repository. In other words, the upstream version overrides the version you have in Streaming SQL Console. If disabled, the project keeps the resources that were not pushed to the Github repository.

The deletion affects the jobs, data sources, functions, Materialized Views and their API keys.

7. Enable or disable Authentication for the project.**What to do next**

After you have created the project, you have the following options to manage the project on the **Projects** page:

1. Click Open to use the created project and start creating SQL jobs.
2. Click Switch to open another project than the active one.
3. Click Delete to remove the project from SSB.

Registering Data Sources in SSB

Data Sources are a set of data endpoints to be used as sources, sinks and catalogs. Data Sources allow you to connect to an already installed component on your cluster, then use that provider for adding tables in SQL Stream Builder (SSB).

You can register Kafka as a data source, or Kudu, Hive, Cloudera Schema Registry and Confluent Schema Registry as a catalog and you can also add custom catalogs. When registering the components, SSB can access the already existing topics from Kafka, tables from Kudu and Hive, and the schema in Cloudera Schema Registry and Confluent Schema Registry. This also means that when you update a data source, for example add new topics, tables and schemas, SSB automatically detects the changes.

Adding Kafka Data Source

You need to register Kafka as a Data Source using the Streaming SQL Console to create Kafka tables in SQL Stream Builder (SSB).

Before you begin

- Make sure that you have Kafka service on your cluster.
- Make sure that you have the right permissions set in Ranger.

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.

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 Data Sources from the **Explorer** view.


4.



Click next to **Kafka**.

5. Select New Kafka Source.

The Kafka Source window appears.

 Kafka Data Source ✕

Name *

Brokers (Comma-separated List) *

Protocol *

Cancel

Create

6. Add a Name to your Kafka provider.

7. Add the broker host name(s) to Brokers.

You need to copy the Kafka broker name(s) from Cloudera Manager.

- a) Go to your cluster in Cloudera Manager.
- b) Click Kafka from the list of services.
- c) Click Instances.
- d) Copy the hostname of the Kafka broker(s) you want to use.
- e) Go back to the Add Kafka Source page.
- f) Paste the broker hostname to the Brokers field.



Note: You can add more than one broker hostname by separating them by commas.

- g) Add the default Kafka port after the hostname(s).

Example:

For TLS disabled

```
docs-test-1.vpc.cloudera.com:9092,  
docs-test-2.vpc.cloudera.com:9092
```

For TLS enabled

```
docs-test-1.vpc.cloudera.com:9093,  
docs-test-2.vpc.cloudera.com:9093
```

8. Select the security Protocol.

The connection protocol must be the same as it is configured for the Kafka cluster in Cloudera Manager.

You can choose from the following protocols:

For PLAINTEXT

- a. Click Create.

For SSL

- a. Provide the path to the Kafka TrustStore and Kafka KeyStore with their dedicated passwords.
- b. Click Create.

For SASL SSL

- a. Provide the path to the Kafka TrustStore and Kafka KeyStore with their dedicated passwords.
- b. Choose an SASL Mechanism.
- c. Provide the Username for SASL.
- d. Provide the Password for SASL.
- e. Click Create.

For SASL PLAINTEXT

- a. Choose an SASL Mechanism.
- b. Provide the Username for SASL.
- c. Provide the Password for SASL.
- d. Click Create.

Results

You have registered Kafka as a data source to be able to add Kafka as a table in your SQL query. The already existing Kafka topics can be selected when adding Kafka as a table.

What to do next

After registering a Kafka data source, you can edit, duplicate and delete it from Streaming SQL Console:

1. Open Data Sources from the **Explorer** view.
- 2.



Click next to **Kafka**.

3. Select Manage.

The **Kafka Sources** tab opens where the registered Kafka providers are listed. You have the following options to manage the Kafka sources:

- Click on one of the existing Kafka providers to edit its configurations.

•



Click to remove the Kafka provider.

•



Click to duplicate the Kafka provider with its configurations.

Adding Catalogs

You need to add Cloudera Schema Registry, Kudu, Hive, Confluent Schema Registry or other services as a Catalog using the Streaming SQL Console in SQL Stream Builder (SSB) to use them with Flink DDL.

Before you begin

- Make sure that you have the required service on your cluster.
- Make sure that you have the right permissions set in Ranger for SSB and the services.

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.
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 Data Sources from the **Explorer** view.

- 4.



Click next to **Catalogs**.

- 5. Select New Catalog.
The Add Catalog window appears.

 Catalog ✕


Catalog Type *

Name *


Filters

Database Filter

Table Filter



Validate

 Unvalidated catalog. Please validate before submit.


Cancel

Create

6. Select the Catalog Type from the following options:

For Schema Registry

- a. Add a Name to your catalog.
- b. Select Schema Registry from the **Catalog Type** drop-down.

 Catalog
×

Catalog Type *

Schema Registry ▼

Name *


Kafka Cluster

Enable TLS ☐

Schema Registry URL

http://<hostname>:7788/api/v1

Validate

 Unvalidated catalog. Please validate before submit.

Cancel

Create


- c. Select the Kafka cluster you registered as Data Source.
- d. Enable TLS, if needed for the communication.
 1. If you enabled TLS, provide the Schema Registry Truststore location and password to the SR TrustStore and SR TrustStore Password field.
- e. Add the Schema Registry URL.
 1. Go to your cluster in Cloudera Manager.
 2. Select Schema Registry from the list of services.
 3. Click on Instances.
 4. Copy the Hostname of Schema Registry.
 5. Add the default port of Schema Registry after the hostname.

Example:

```
http://docs-test-1.vpc.cloudera.com:7788/api/v1
```

For Kudu

- a. Select Kudu from the Catalog Type drop-down.

 Catalog ×

Catalog Type *

Kudu

Name *

Kudu Masters

<hostname>:<port>


Filters

Database Filter

Table Filter

+

Validate

 Unvalidated catalog. Please validate before submit.

Cancel

Create

b. Add the host URL of Kudu Masters.


1. Go to your cluster in Cloudera Manager.
2. Select Kudu from the list of services.
3. Click on Instances.
4. Copy the Hostname of the Master Default Group.
5. Add the default port of Kudu after the hostname.

Example:

```
docs-test-1.vpc.cloudera.com:7051
```

For Hive

- a. Select Hive from the Catalog Type drop-down.

 Catalog ×

Catalog Type *

Hive

Name *

Default Database

default


Filters

Database Filter

Table Filter

+

Validate

 Unvalidated catalog. Please validate before submit.

Cancel


Create

b. Provide a Name to the Hive catalog created in SSB.

c. Provide the name of the Default Database in Hive.

For Confluent Schema Registry

a. Select Confluent Schema Registry from the **Catalog Type** drop-down.

 Catalog
 ✕

Catalog Type *

Name *

Kafka Cluster


Schema Registry URL

Authentication method

Username

Password

Validate


 Unvalidated catalog. Please validate before submit.

- b. Add a Name to your catalog.
- c. Select the Kafka cluster you registered as Data Source.
- d. Add the Confluent Cloud Schema Registry URL provided by your Confluent connection configurations.
- e. Choose an Authentication method from the following:

BASIC

If the basic authentication method is selected, you must provide the username and password.

CUSTOM

Custom authentication methods include bearer authentication, SSL authentication and so on. If the custom authentication method is selected, you must provide the authentication property keys to the **Custom Properties** field as they are set for the Confluent Cloud Schema Registry. For example, if the Confluent Cloud Schema Registry is configured with bearer authentication, you need to provide the `bearer.auth.credentials.source` and `bearer.auth.token`. If SSL authentication is configured, the

schema.registry.ssl.keystore.location, schema.registry.ssl.keystore.password and so on must be provided.

For Custom

- a. Select Custom from the Catalog Type drop-down.

Catalog

Catalog Type *

Custom

Name *

Custom Properties

Property Key Property Value

Filters

Database Filter Table Filter

Validate Unvalidated catalog. Please validate before submit. Cancel Create

- b. Provide a Property Key.
- c. Provide a Property Value.

If needed, you can specify more custom properties by using the plus icon.

7. Click on Add Filter.
 - a) Provide a Database and Table filter if you want to select specific tables to use from the catalog.
8. Click on Validate.
9. If the validation is successful, click Create.

Results

You are ready to use the added catalog in SSB with Flink DDL. The already existing schemas in Schema Registry and Confluent Schema Registry, tables in Kudu and Hive are automatically imported to SSB.


What to do next

After registering a catalog, you can edit, duplicate and delete it from Streaming SQL Console:

1. Open Data Sources from the **Explorer** view.

2.



Click  next to **Catalogs**.


3. Select Manage.

The **Catalogs** tab opens where the registered catalogs are listed. You have the following options to manage the catalog sources:

- Click on one of the existing catalogs to edit its configurations.


•



Click  to remove the catalog.

•



Click  to duplicate the catalog with its configurations.

Connectors

SQL Stream Builder (SSB) 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 SSB:

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
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

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.
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 **Jobs** from the **Explorer**.

4. Select New Job > Create .

The SQL Editor for the created job opens in a tab.

5. Click Templates.

6. 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 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.
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.

- Add a name to the property.
- Add a default value to the property.
- Add a description to the property.
- Click Required to make a property mandatory.
- 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 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 values. 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. After selecting this option, you need to further provide the name of the created DLQ topic.

6. Click Execute.

Related Information

[Adding a Kafka sink with Schema Registry catalog](#)

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 SQL Stream Builder (SSB).

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 (CSA) 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. CSA 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 SSB. 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 or Hive 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 or Hive 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.

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 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 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 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

Data formats

There is a set of table formats that can be used with the supported connectors of SQL Stream Builder (SSB). The data format is a storage format that defines how to map binary data to the columns of a table. You can use the data formats based on which connector they are supported for.

Adding data formats


When adding new data formats to SQL Stream Builder, you need to specify the property list and must upload the data format 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.
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 **Data Formats**.

5. Click New Connector.

6. Provide a name for the data format as Type.
7. Upload the data format JAR file.
8. Click Properties to add properties to the data format.
 - 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 data format.

9. Click Create.

The newly added data format is listed under the **Data Formats**.

Concept of tables in SSB

The core abstraction for Streaming SQL is a Table which represents both inputs and outputs of the queries. SQL Stream Builder (SSB) tables are an extension of the tables used in Flink SQL to allow a bit more flexibility to the users. When creating tables in SSB, you have the option to either add them manually, import them automatically or create them using Flink SQL depending on the connector you want to use.

A Table is a logical definition of the data source that includes the location and connection parameters, a schema, and any required, context specific configuration parameters. Tables can be used for both reading and writing data in most cases. You can create and manage tables either manually or they can be automatically loaded from one of the catalogs as specified using the Data Sources section.

In SELECT queries the FROM clause defines the table sources which can be multiple tables at the same time in case of JOIN or more complex queries.

When you execute a query, the results go to the table you specify after the INSERT INTO statement in the SQL window. This allows you to create aggregations, filters, joins, and so on, and then route the results to another table. The schema for the results is the schema that you have created when you ran the query.

For example:

```
INSERT INTO air_traffic -- the name of the table sink
SELECT
lat,lon
FROM
airplanes -- the name of the table source
WHERE
icao <> 0;
```

Table types in SSB

Kafka Tables

Apache Kafka Tables represent data contained in a single Kafka topic in JSON, AVRO or CSV format. It can be defined using the Streaming SQL Console wizard or you can create Kafka tables from the pre-defined templates.

Tables from Catalogs

SSB supports Kudu, Hive and Schema Registry as catalog providers. After registering them using the Streaming SQL Console, the tables are automatically imported to SSB, and can be used in the SQL window for computations.



Note: You cannot edit the properties of the already existing tables that are automatically imported from the catalogs. To distinguish between editable and non-editable tables, in other words, user defined and catalog tables, the Edit and Delete table options are not available on the Tables page.

Flink Tables

Flink SQL tables represent tables created by the standard CREATE TABLE syntax. This supports full flexibility in defining new or derived tables and views. You can either provide the syntax by directly adding it to the SQL window or use one of the predefined DDL templates.

Webhook Tables

Webhooks can only be used as tables to write results to. When you use the Webhook Tables the result of your SQL query is sent to a specified webhook.

Creating Kafka tables using wizard

After registering a Kafka data source, you can use the Kafka table wizard in Streaming SQL Console to create a Kafka table.

About this task

You can query your streaming data using Kafka tables in SQL Stream Builder (SSB). You have the option to use the Kafka service in your environment, or connect to an external Kafka service. When creating Kafka tables you can use the Add Kafka wizard, the predefined templates or you can directly add a custom CREATE TABLE statement with the required properties in the SQL window.

You can also create Kafka tables using one of the Kafka templates. For more information, see the [Kafka connectors](#) and [Using connectors with templates](#) sections.

Before you begin

- Make sure that you have registered Kafka as a Data Source.

- Make sure that you have created topics in Kafka.



Important: When creating the topic for the Kafka sink, make sure to not use log compaction as it can cause the SQL job to fail.

- Make sure there is generated data in the Kafka topic.
- Make sure that you have the right permissions set in Ranger.

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.

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. Click New Kafka Table .

The **Kafka Table** window appears.


 Kafka Table
 ✕

Table Name *

Kafka Cluster *

Data Format *

Topic Name *

Schema Definition

Event Time

Properties

Deserialization

```

1  {
2    "doc": "Default schema - modify as necessary",
3    "namespace": "com.eventador.exampleschema",
4    "type": "record",
5    "name": "exampleSchema",
6    "fields": [
7      {
8        "type": "string",
9        "name": "name"
10     },
11     {
12       "type": "int",
13       "name": "temp"
14     }
15   ]
16 }

```

✓ Schema is valid

Detect Schema

Cancel

Create and Review

5. Provide a Table Name.



Note: You will use this name in the FROM clause when running the SQL statement.

6. Select a registered Kafka provider as Kafka cluster.

7. Select the Data format.

- You can select JSON as data format.
- You can select AVRO as data format.

8. Select a Kafka topic from the list.



Note: The automatically created topics for the websocket output is also listed here. Select the topic you want to use for the SQL job.

9. Add a customized schema to Schema Definition or click Detect Schema to read a sample of the JSON messages, and automatically infer the schema.



Note: The Detect Schema functionality is only available for JSON data. If there are no messages in the topic, then no schema will be inferred. If your schema contains a field named timestamp, this causes a schema validation error as timestamp is a reserved word used for Kafka internal timestamps.

10. Customize your Kafka Table with the following options:

- a) Configure the Event Time if you do not want to use the default Kafka Timestamps.
- b) Configure an Input Transform on the Data Transformations tab.



Note: The Input Transformation is only supported for JSON data formats.

- c) Configure any Kafka properties required on the Properties tab.
- d) Select a policy for deserialization errors on the Deserialization tab.

For more information about how to configure the Kafka table, see the *Configuring Kafka tables* section.

11. Click Create and Review.

Results

The Kafka Table is ready to be used for the SQL job either at the FROM or at the INSERT INTO statements.

Related Information

[Adding a Kafka sink with Schema Registry catalog](#)

Configuring Kafka tables

The user-defined Kafka table can be configured based on the schema, event time, input transformations and other Kafka specific properties using either the Kafka wizard or DDL.

Schema Definition tab

When using the Add Kafka table wizard on the Streaming SQL Console, you can configure the schema under the Schema tab.

Schema is defined for a given Kafka source when the source is created. The data contained in the Kafka topic can either be in JSON or AVRO format.



Note: When using Schema Registry with Kafka, JSON data format is supported when using CSA with CDP Private Cloud Base 7.1.8.

Kafka Table
 ✕

Table Name *

Kafka Cluster *

Local Kafka

Data Format *

JSON

Topic Name *

connect-status

Schema Definition

Event Time

Data Transformation

Properties

Deserialization

```

1  {
2    "doc": "Default schema - modify as necessary",
3    "namespace": "com.eventador.exampleschema",
4    "type": "record",
5    "name": "exampleSchema",
6    "fields": [
7      {
8        "type": "string",
9        "name": "name"
10     },
11     {
12       "type": "int",
13       "name": "temp"
14     }
15   ]
16 }

```

Schema is valid
 [Detect Schema](#)

[Cancel](#)
[Create and Review](#)

When specifying a schema you can either paste it to the Schema Definition field or click the Detect schema button to identify the schema used on the generated data. The Detect Schema functionality is only available for JSON data.



Note: If your schema contains a field named timestamp, this causes a schema validation error as timestamp is a reserved word used for Kafka internal timestamps.

Related Information

[Adding a Kafka sink with Schema Registry catalog](#)

Event Time tab

When using the Add Kafka table wizard on the Streaming SQL Console, you can configure the event time under the Event Time tab.

You can specify Watermark Definitions when adding a Kafka table. Watermarks use an event time attribute and have a watermark strategy, and can be used for various time-based operations. The Event Time tab provides the following properties to configure the event time field and watermark for the Kafka stream:

- Input Timestamp Column: name of the timestamp column in the Kafka topic from where the watermarks are mapped to the Event Time Column of the Kafka table
- Event Time Column: default or custom name of the resulting timestamp column where the watermarks are going to be mapped in the created Kafka table
- Watermark seconds: number of seconds used in the watermark strategy. The watermark is defined by the current event timestamp minus this value.

You have the following options to configure the watermark strategy for the Kafka tables:

- Using the default Kafka Timestamps setting
- Using the default Kafka Timestamps setting, but providing custom name for the Event Time Column
- Not using the default Kafka Timestamps setting, and providing all of the Kafka timestamp information manually
- Not using watermark strategy for the Kafka table

Using the default Kafka Timestamp setting

By default, the Use Kafka Timestamps feature is enabled when you create the Kafka table. In the Event Time Column, the new event time field is extracted from the Kafka message header with the *'eventTimestamp'* predefined column name.

<
Schema Definition
Event Time
Data Transformation
Properties
Des
>

Event Time Column

Watermark Seconds

Use Kafka Timestamps ☒

After saving your changes, you can view the created DDL syntax for the table next to the SQL Editor on the **Console** page.

The following DDL example shows the default setting of the Event Time Column and Watermark Seconds where the corresponding fields were not modified.

```
'eventTimestamp' TIMESTAMPS(3) METADATA FROM 'timestamp',
WATERMARK FOR 'eventTimestamp' AS 'eventTimestamp' - INTERVAL '3' SECOND
```

Using the default Kafka Timestamp setting with custom Event Time Column name

When you want to modify the timestamp field of the DDL from the stream itself, you must provide a custom name of the Event Time Column. You can also add a custom value to the Watermark Seconds. The following example shows that *'ets'* is the custom name for the Event Time Column, and *'4'* is the custom value for the Watermark Seconds.

<
Schema Definition
Event Time
Data Transformation
Properties
Des
>

Event Time Column

ets

Watermark Seconds

4

Use Kafka Timestamps ☒

The Event Time Column can only be modified if the following requirements are met for the timestamp field of the Input Timestamp Column:

- The column type must be "long".
- The format must be in epoch (in milliseconds since January 1, 1970).

The DDL syntax should reflect the changes made for the watermark strategy as shown in the following example:

```
'ets' TIMESTAMP(3) METADATA FROM 'timestamp',
WATERMARK FOR 'ets' - INTERVAL '4' SECOND
```

Manually providing the Kafka timestamp information

When you want to manually configure the watermark strategy of the Kafka table, you can provide the timestamp column name from the Kafka source, and add a custom column name for the resulting Kafka table. Make sure that you provide the correct column name for the Input Timestamp Column that exactly matches the column name in the Kafka source data.

To manually provide information for the watermark strategy, disable the Use Kafka Timestamps feature using the toggle, and provide the following information to the column name fields:

- Input Timestamp Column: name of the timestamp field in the Kafka source
- Event Time Column: predefined *'eventTimestamp'* name or custom column name of the timestamp field in the created Kafka table

As an example, you have a timestamp column in the source Kafka topic named *'ts'*, and want to add a new timestamp column in your Kafka table as *'event_time'*. You provide the original timestamp column name in the Input Timestamp Column as *'ts'*, and add the custom *'event_time'* name to the Event Time Column.

[Schema Definition](#) [Event Time](#) [Data Transformation](#) [Properties](#) [Description](#)

Input Timestamp Column

ts

Event Time Column

event_time

Watermark Seconds

3

Use Kafka Timestamps ☐

This results in that the watermarks from the *'ts'* column is going to be mapped to the *'event_time'* column of the created Kafka table. As *'event_time'* will become the timestamp column name in the Kafka table, you must use the custom name (in this example the *'event_time'*) when querying the Kafka stream. This configuration of the timestamp columns is optional.

The Event Time Column can only be modified if the following requirements are met for the timestamp field of the Input Timestamp Column:

- The column type must be "long".
- The format must be in epoch (in milliseconds since January 1, 1970).

Not using watermark strategy for Kafka table

In case you do not need any watermark strategies, disable the Use Kafka Timestamps feature using the toggle, and leave the column and seconds field empty.

<
Schema Definition
Event Time
Data Transformation
Properties
Des
>

Input Timestamp Column

Event Time Column

Watermark Seconds

Use Kafka Timestamps ☐



Note: Flink validates the input and output schemas of the data. You can only insert data into a Kafka topic, if the input and output data matches based on the schema defined for the topic. When customizing the timestamp column, make sure that the output data has the same schema.



Note: When configuring the timestamp for the Kafka tables, you must consider the timezone setting of your environment as it can affect the results of your query. For more information, see the [Known Issues](#) in the Release Notes.

Data Transformations tab

When using the Add Kafka table wizard on the Streaming SQL Console, you can apply input transformation under the Transformations tab. Input transformations can be used to clean or arrange the incoming data from the source using javascript functions.

Input Transforms are a powerful way to clean, modify, and arrange data that is poorly organized, has changing format, and has data that is not needed or otherwise hard to use. With the Input Transform feature of SQL Stream Builder, you can create a javascript function to transform the data after it has been consumed from a Kafka topic, and before you run SQL queries on the data.

You can use Input Transforms in the following situations:

- The source is not in your control, for example, data feed from a third-party provider
- The format is hard to change, for example, a legacy feed, other teams of feeds within your organization
- The messages are inconsistent
- The data from the sources do not have uniform keys, or without keys (like nested arrays), but are still in a valid JSON format
- The schema you want does not match the incoming topic



Note: When using Input Transforms the schema you define for the Kafka table is applied on the output of the transformed data.

•



You can use the Input Transforms on Kafka tables that have the following characteristics:

- Allows one transformation per source.
- Takes record as a JSON-formatted string input variable. The input is always named `record`.
- Emits the output of the last line to the calling JVM. It could be any variable name. In the following example, `out` and `emit` is used as a JSON-formatted string.

A basic input transformation looks like this:

```

var out = JSON.parse(record.value);    // record is input, parse JSON f
ormatted string to object

// add more transformatio
ns if needed
JSON.stringify(out);                  // emit JSON formatted
  string of object
  
```

Adding transformation to Kafka tables

When adding a Kafka table using the wizard, you can specify the input transformation on the Data Transformation tab. You have one of the following steps to apply an input transformation:

1. Add your javascript transformation code to the **Data Transformation** window.

Make sure the output of your transform matches the Schema definition detected or defined for the Kafka table.

2. Click Install default template and schema.

The Install Default template and schema option fills out the Data Transformation box with a template that you can use to create the Input Transform, and matches the schema with the format.

Kafka record metadata access

There are cases when it is required to access additional metadata from the Kafka record to implement the correct processing logic. SQL Stream Builder has access to this information using the Input Transforms functionality.

The following attributes are supported in the headers:

```

record.topic
record.key
record.value
record.headers
record.offset
record.partition
  
```

For example, an input transformation can be expressed as the following:

```

var out = JSON.parse(record);
out['topic'] = message.topic;
out['partition'] = message.partition;
JSON.stringify(out);
  
```

For which you define a schema manually, or use the Detect Schema feature:

```
{
  "name": "myschema",
  "type": "record",
  "namespace": "com.cloudera.test",
  "fields": [
    {
      "name": "id",
      "type": "int"
    },
    {
      "name": "topic",
      "type": "string"
    },
    {
      "name": "partition",
      "type": "string"
    }
  ]
}
```

The attribute `record.headers` is an array that can be iterated over:

```
var out = JSON.parse(record);
var header = JSON.parse(record.headers);
var interested_keys = ['DC']; // should match schema definition

out['topic'] = record.topic;
out['partition'] = record.partition;
Object.keys(header).forEach(function(key) {
  if (interested_keys.indexOf(key) > -1) { // if match found for schema,
    set value
    out[key] = header[key];
  }
});
JSON.stringify(out);
```

For which you define a schema as follows:

```
{
  "name": "myschema",
  "type": "record",
  "namespace": "com.cloudera.test",
  "fields": [
    {
      "name": "id",
      "type": "int"
    },
    {
      "name": "topic",
      "type": "string"
    },
    {
      "name": "partition",
      "type": "string"
    },
    {
      "name": "DC",
      "type": "string"
    }
  ]
}
```



```
]
}
```

Properties tab

When using the Add Kafka table wizard on the Streaming SQL Console, you can configure the properties under the Properties tab.

You can specify certain properties to define your Kafka source in detail. You can also add customized properties additionally to the default ones. To create properties, you need to give a name to the property and provide a value for it, then click Actions.

Kafka Table ✕

Table Name *

Kafka Cluster *

Data Format *

Topic Name *

Schema Definition

Event Time

Properties

Deserialization

Default Read Position

Beginning of topic

Consumer Group

Use random consumer group

Custom Properties

Property Key

Property Value

+

✓ Schema is valid

Cancel

Create and Review

Deserialization tab

When creating a Kafka table, you can configure how to handle errors due to schema mismatch using DDL or the Kafka wizard.

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

Using the Kafka wizard

When you create the Kafka table using the wizard on the Streaming SQL Console, you can configure the error handling with the following steps:

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 .
2. Click Create Job or select a previous job on the **Getting Started** page.

You are redirected to the **Console** page.

3. Select Add tables > Apache Kafka .

The **Add Kafka table** window appears

4. Select Deserialization tab.

Kafka Table ✕

Table Name *

Kafka Cluster *

Data Format *

Topic Name *

Schema Definition

Event Time

Properties

Deserialization

Deserialization Failure Handler Policy

✔ Schema is valid

Cancel

Create and Review

5. Choose from the following policy options under **Deserialization Policy:**

- Fail
- Ignore
- Ignore and Log
- Save to DLQ

If you choose the Save to DLQ option, you need to create a dedicated Kafka topic where you store the error message. After selecting this option, you need to further select the created DLQ topic.

- Click Create and Review.

Assigning Kafka keys in streaming queries

Based on the Sticky Partitioning strategy of Kafka, when null keyed events are sent to a topic, they are randomly distributed in smaller batches within the partitions.

As the results of the SQL Stream queries by default do not include a key, when written to a Kafka table, the Sticky Partitioning strategy is used. In many cases, it is useful to have more fine-grained control over how events are distributed within the partitions. You can achieve this in SSB by configuring a custom key based on your specific workload.

For example:

```
SELECT sensor_name AS _eventKey --sensor_name becomes the key in the output
kafka topic
FROM sensors
WHERE eventTimestamp > current_timestamp;
```

To configure keys in DDL-defined tables (those that are configured using the Templates), refer to the official [Flink Kafka SQL Connector](#) documentation for more information (specifically the `key.format` and `key.fields` options).

Performance & Scalability

The Kafka and SQL Stream Builder integration enables you to use the Kafka-specific syntax to customize your SQL queries based on your deployment and use case.

You can achieve high performance and scalability with SQL Stream Builder, but the proper configuration and design of the source Kafka topic is critical. SQL Stream Builder can read a maximum of one thread per Kafka partition. You can achieve the highest performance configuration when setting the SQL Stream Builder threads equal to or higher than the number of Kafka partitions.

If the number of partitions is less than the number of SQL Stream Builder threads, then SQL Stream Builder has idle threads and messages show up in the logs indicating as such. For more information about Kafka partitioning, see the [Kafka partitions](#) documentation.

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.
 - 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.
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.

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.

Creating Flink tables using Templates

You can use the predefined templates to create tables by choosing one of the connector templates on the Console page of Streaming SQL Console. The Flink SQL templates are predefined examples of CREATE TABLE statements which you can fill out with your job specific values.

You can create tables using the predefined templates in SQL Stream Builder. The predefined templates consist of the CREATE TABLE statement, and every connection property that is needed for the given connector. You only need to fill out the templates and execute them in the Streaming SQL Console to create the table. When filling out the templates, you can add configuration based on what is specified in the given connector template. You can also customize the table name, column names and timestamp information for a template.

You can choose from the following templates based on the connector type:

Blackhole

The BlackHole connector can be used to write all input records into. It is designed for high performance testing and UDF to output, not a substantive sink.

Datagen

The Data generator connector can be used to sample randomly generated data in SSB. You can use this connector to try out and test SQL queries as records are generated until the job is running.

Faker

The Faker connector can be used to generate fake data based on the Java faker expression. You can use this connector to try out and test SQL queries as records are generated until the job is running.

Filesystem

The filesystem connector can be used to access partitioned files in file systems, and enables reading and writing from a local or distributed file system such as local, HDFS, S3 and so on. You only need to specify the data format that is used in the file system. You can choose from the following formats:

- Avro
- JSON
- CSV
- ORC
- Parquet

JDBC

The JDBC connector enables reading data from and writing data into any relational databases with a JDBC driver. You can use PostgreSQL, MySQL and Hive as databases for the connector.

Debezium CDC

You can use the Debezium CDC connector to stream changes in real-time from MySQL, PostgreSQL, Oracle, Db2 and SQL Server into Kafka. Debezium provides a unified format schema for changelog and supports serializing messages using JSON and Avro.

You can access and import the Templates from Streaming SQL Console:

1. Navigate to the **Streaming SQL Console**.
 - a. Go to your cluster in Cloudera Manager.
 - b. Click on SQL Stream Builder from the list of Services.
 - c. Click on the SQLStreamBuilder Console.

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

2. Click Create Job or select a previous job on the **Getting Started** page.

You are redirected to the **Console** page.

3. Click Templates at the SQL Editor.
4. Select the template you want to use.

The template is imported to the SQL window.

5. Customize the fields of the template.
6. Click Execute.

The table is created based on the selected template, and appears next to the SQL Editor.

For full reference on the Flink SQL DDL functionality, see the official [Apache Flink](#) documentation.

SQL jobs

Creating and naming SQL jobs

You need to create or select an already existing SQL job on the Streaming SQL Console to be able to submit SQL queries.

About this task

You can manage your SQL jobs and resources using the **Explorer** view or the **Home** tab of your project. Every job and resource related action is the same for both views of Streaming SQL 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.
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  **New Job** on the **Explorer** view.

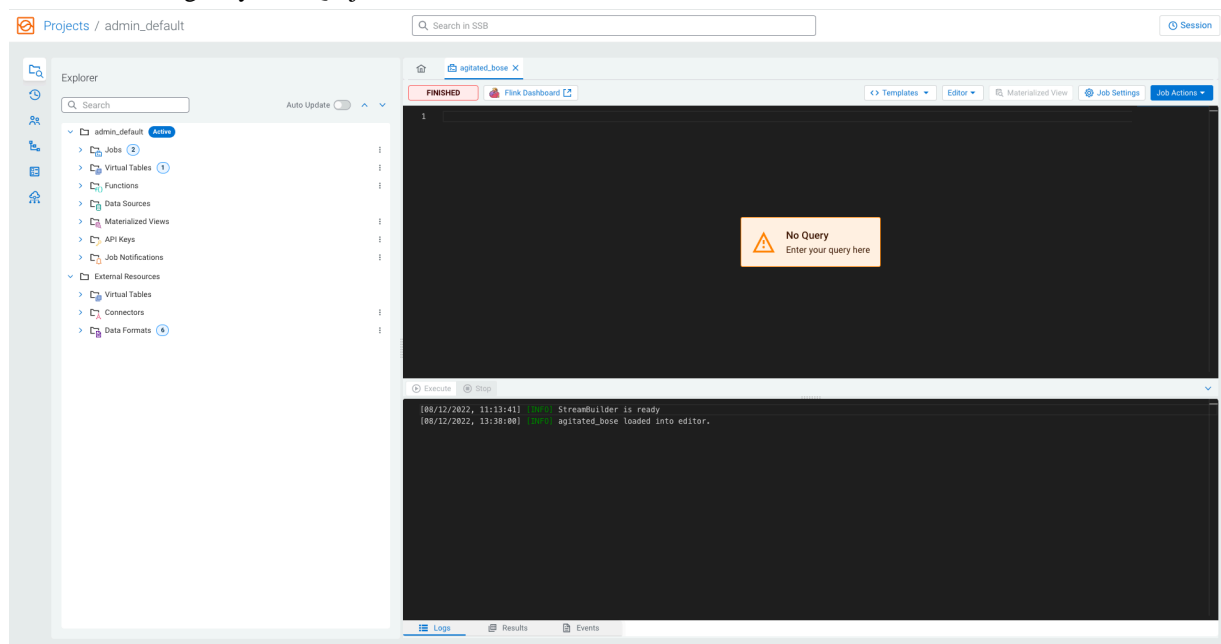
4. Provide a Name to your SQL job.

You have the following options to name your SQL job:

- Use the predefined name in the Job Name field when opening the Streaming SQL Console
- Use the Generate Job Name button to generate a new random name for your job
- Manually provide the name of the job to the Job Name field

5. Click Create.

You are redirected to the SQL Editor where you can submit your SQL queries, create and manage Materialized Views and configure your SQL jobs.



What to do next

After a new job is created, you can save the job and its settings using the **Job Actions Save** button. When you make a change to the SQL query, job settings or create a Materialized View, you have the option to save the changes with the job. You can also revert your changes to the latest saved version of the job using the **Revert Changes** button. You can also delete the job from the SQL Editor using the **Delete** button.

Running SQL Stream jobs

Every time you run an SQL statement in the SQL Stream console, it becomes a job and runs on the deployment as a Flink job. You can manage the running jobs using the **Jobs** tab on the UI.

About this task



There are two logical phases to run a job:

1. **Parse:** The SQL is parsed and checked for validity and then compared against the virtual table schema(s) for correct typing and key/columns.
2. **Execution:** If the parse phase is successful, a job is dynamically created, and runs on an open slot on your cluster. The job is a valid Flink job.

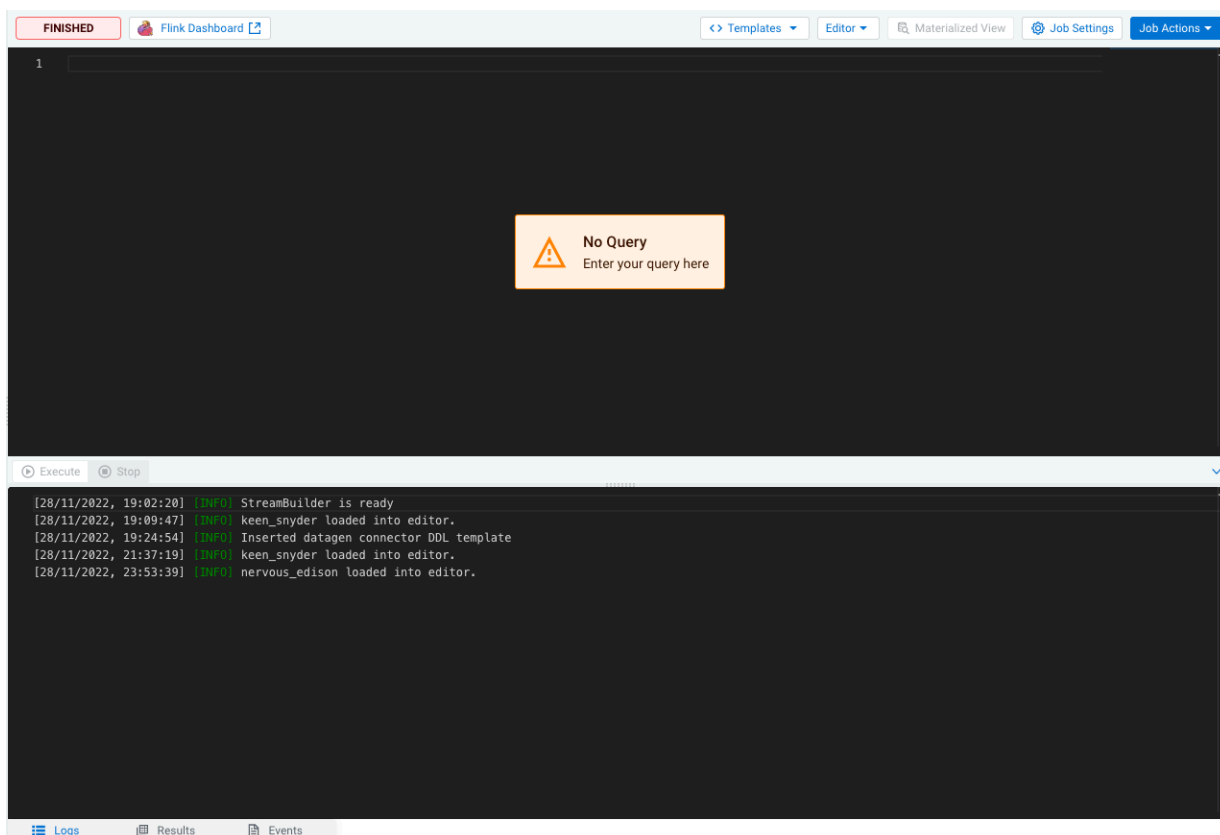
Before you begin

- Make sure that you have registered a Data Source if you use the Kafka service on your cluster.
- Make sure that you have added Kudu, Hive or Schema Registry as a catalog if you use them for your SQL job.

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.
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 **Jobs** from the **Explorer**.
4. Select New Job .
The SQL Editor for the created job opens in a tab.
5. Provide a name for the SQL job.
 - a) Optionally, you can click Generate Random Name to generate a name for the SQL job.
6. Click Create.
7. Create a table using one of the following options:
 - Using the Add Kafka or Webhook table wizard.
 - Using the Templates.
 - Add your custom CREATE TABLE statement to the SQL window.

8. Add a SQL query to the SQL Editor.

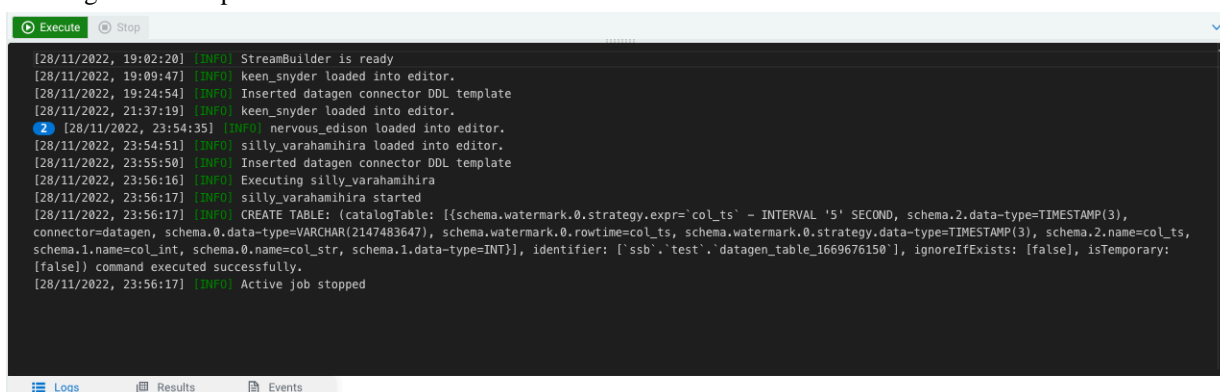


Note: You cannot start a job without adding a SQL statement in the SQL editor window. In case, there are no SQL statements provided and you click Execute, the following error message is displayed: You must provide a SQL query.

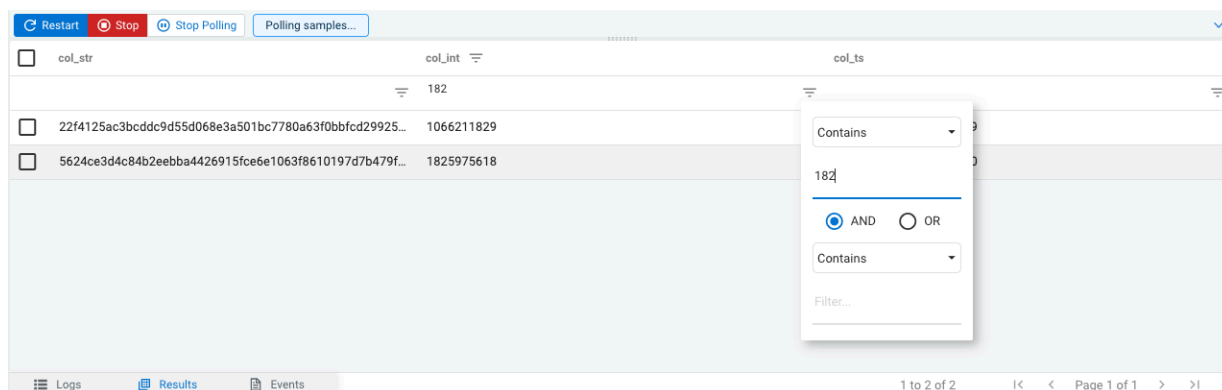
When starting a job, the number of slots consumed on the specified cluster is equal to the parallelism setting. The default is one slot. To change the parallelism setting and more job related configurations, click Job Settings.

9. Click Execute.


The Logs window updates the status of SSB.



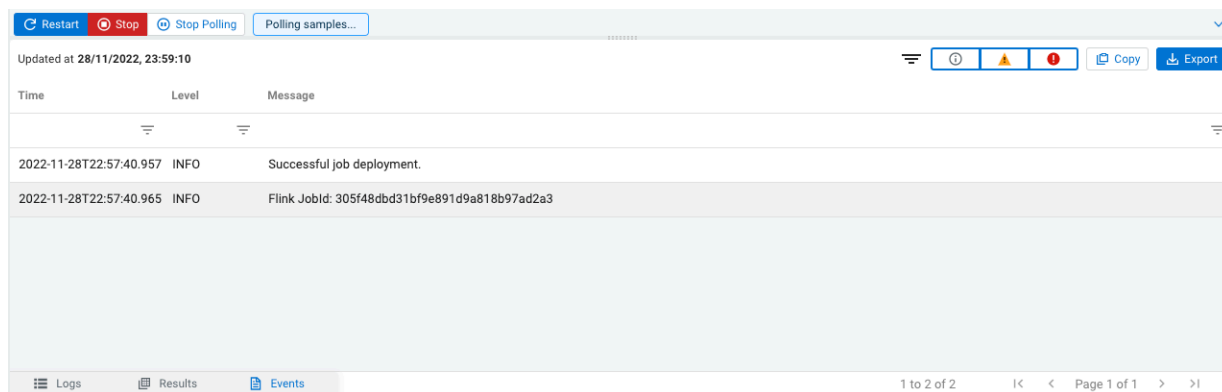
10. Click Results to check the sampled data.



You can pause or restart the sampling of a SQL job with the Pause and Refresh buttons. The status of the polling can also be seen next to the Pause button. The Results tab also allows you to change the order of the samples, and to filter the samples using different types of conditions that you can also combine. For changing the order, click on the name of the column, the direction of the ordering is indicated next to the column name by arrows.

For filtering, you can either click the  button next to the column name or under the row of the column name. The filtering value can be changed by typing the values directly to the filtering row, the conditions need to be changed using the filtering arrow.

11. Click Events to check the job related information.



Results

A job is generated that runs the SQL continuously on the stream of data from a table, and pushes the results to a table, to the Results tab or to a Materialized View.

Configuring SQL job settings

If you need to further customize your SQL Stream job, you can add more advanced features to configure the job restarting method and time, threads for parallelism, sample behavior, exactly once processing and restoring from savepoint.

Before running a SQL query, you can configure advanced features by clicking on the Job Settings button at the SQL Editor.



Job Settings

 Settings

 Job Notifications

General

Execution Mode

SESSION - Job runs with current Session

Job Parallelism (threads)

1

Restore From Savepoint ☐

Sample

Sample Count ⓘ

100

Sample Window Size ⓘ

100

Sample Behavior

Sample all messages (may impact performance)

Checkpoint

Enable Checkpointing ☐

Execution Mode

You can select the execution mode for the SQL job that can be a session or a per-job mode. For more information, see the [Executing SQL jobs in production mode](#) documentation.

Job parallelism (threads)

The number of threads to start to process the job. Each thread consumes a slot on the cluster. When the Job Parallelism is set to 1, the job consumes the least resources. If the data source supports parallel reads, increasing the parallelism can raise the maximum throughput. For example, when using Kafka as a data source, setting the parallelism to the equal number as the partitions of the topic can be a starting point for performance tuning.

Restore From Savepoint

You can enable or disable restoring a SQL job from a Flink savepoint after stopping it. The savepoint is saved under `hdfs:///user/flink/savepoints` by default.

Sample Count

The number of sample entries to poll before sampling stops. To poll an unlimited number of sample entries, add 0 as the *Sample Count* value.

Sample Window Size

The number of sample entries from active sampling displayed in the Results tab. To display an unlimited number of sample entries, add 0 as the *Sample Window Size* value.

Sample Behavior

You have the following options to choose the behavior of the sampled data under the Results tab:

- Sample all messages
- Sample one message every second
- Sample one message every five seconds

Enable Checkpointing

You can enable and disable checkpointing for a SQL job. By default the checkpointing is enabled.

Checkpoint Mode

Switching between checkpointing modes. You can choose between At Least Once or Exactly Once.

Checkpoint Interval

The time in milliseconds between checkpointing attempts.

Checkpoint Timeout

The maximum time in milliseconds until a checkpointing attempt is timed out.

Tolerable Checkpoint Failures

The number of checkpointing attempts until the job is aborted.

Failure Restart Strategy

Switching between restarting strategies when a job fails. You can choose between enabling and disabling auto recovery. By default auto recovery is enabled for checkpointing.

Adjusting logging configuration in Advanced Settings

You can customize the logging configurations for the SQL Stream Builder (SSB) job on the Streaming SQL Console in per-job mode or session mode. Adjusting the log configuration enables you to control the log levels of all the underlying libraries: Flink, Hadoop, Kafka, Zookeeper, other common libraries, and connectors to get more or less information in your job's log.

About this task

The customization of the log configuration works differently based on the job deployment mode:

Session mode

The `execution.target` is set to *yarn-session* mode, this is the default execution mode.

The log configuration is set at the start time if the Flink YARN session is applied to every job execution. For example, the current log configuration is applied if and only if the Flink YARN session is not set on the Session tab of the Compose page.



Note: The Reset Session button only resets the SSB Session, not the underlying Flink YARN session. To do that, you have to kill the YARN application that is indicated under Flink Yarn Session on the Session tab.

Per-job mode

The `execution.target` is set to *yarn-per-job* mode.

When you change the default execution mode to per-job, the currently applied log configuration is going to be used for the job. To configure the execution mode, you need to start the SQL query with the following line:

```
SET 'execution.target'='yarn-per-job' ;
```

Procedure

1. Click New Job or select a previous job on the **Jobs** page.
You are redirected to the **SQL Editor** of the job.
2. Click Job Settings.

3. Click Advanced.

Job Settings

Settings **Advanced** Job Notifications

Custom Log Configuration ⓘ

Log configuration won't be persisted

```

rootLogger.level = INFO
rootLogger.appenderRef.file.ref = MainAppender
#Uncomment this if you want to _only_ change Flink's logging
#logger.flink.name = org.apache.flink
#logger.flink.level = INFO

# The following lines keep the log level of common libraries/connectors on
# log level INFO. The root logger does not override this. You have to manually
# change the log levels here.
logger.akka.name = akka
logger.akka.level = INFO
logger.kafka.name= org.apache.kafka
logger.kafka.level = INFO
logger.hadoop.name = org.apache.hadoop
logger.hadoop.level = INFO
logger.zookeeper.name = org.apache.zookeeper
logger.zookeeper.level = INFO

# Log all infos in the given file
appender.main.name = MainAppender
appender.main.type = File
appender.main.append = false
appender.main.fileName = ${sys:log.file}
appender.main.layout.type = PatternLayout
appender.main.layout.pattern = %d{yyyy-MM-dd HH:mm:ss,SSS} %-5p %-60c %x - %m%n

# Suppress the irrelevant (wrong) warnings from the Netty channel handler
logger.netty.name = org.apache.flink.shaded.akka.org.jboss.netty.channel.DefaultChannelPipeline

```

4. Modify the settings based on your requirements.

5. Close the **Job Settings** window.

6. Click Save to save the job settings.

7. Add and execute a SQL statement.

8. Click **SQL Jobs**.

9. Search for the job you have executed previously.

10. Click **Flink Dashboard**.

The **Flink Dashboard** opens in a new window.

11. Click Task Managers > Logs .

The log information appears in the log window based on your custom configurations.

Configuring YARN queue for SQL jobs

You can configure the YARN application queue with a custom value for a SQL job using the Streaming SQL Console.

With YARN queues, you can deploy applications to a specific subset of nodes with separate or limited resources, according to configuration. This enables you to have a separate execution environment with limited resources to experiment with new applications without impacting any production operation.

The YARN application queue can only be configured in a per-job execution target using the production mode of SQL Stream Builder. This means that the default value of the YARN queue does not change, and you can only customize it to the specific job that is executed in the production mode.

The default value of YARN application queue can be configured in Cloudera Manager. After accessing the SQL Stream Builder service on your cluster, you need to open the configurations, and search for YARN application queue where you can change the default value. This default value is overwritten when using the SET statement in production mode for specifying the YARN queue for a SQL job.

For more information about running jobs in production mode, see the [Executing SQL jobs in production mode](#) section.

You can configure the YARN application queue using the SET statement in Streaming SQL Console as the following code example shows:

```
--PROD
set 'execution.target' = 'yarn-per-job';
set 'yarn.application.queue' = 'my-queue';
select * from datagen_table_1648801198
```

Managing session for SQL jobs

By default, the SQL Stream jobs are running in a session cluster. This means that multiple Flink jobs run in the same YARN session sharing the cluster, allocated resources, the Job Manager and Task Managers. The session starts when you open the Streaming SQL Console. You can reset the session, and set the properties of the session using the Streaming SQL Console.

Resetting a session

When you reset your session, every configuration, temporary table and view, default database and catalog will be lost.

»

🕒 Session

🔍 Search Reset Session

General Properties Environment Variables User Defined Variables

Default Catalog	ssb
Default Database	test
Flink Yarn Session	
cluster_id	application_1669640044713_0005
tracking_url	http://docstest-1.docstest.root.hwx.site:8088/proxy/application_1669640044713_0005/
Session ID	35c74260-c55e-42fb-8a2e-3b614f72b1be
Session Start	2022-11-28 13:46:08 (9 hours ago)
Temporary Tables	
Temporary Views	
Username	admin

Configuring properties for a session

You can configure the session properties using the SET statement in the SQL window.

1. Use the SET statement in the **SQL Editor** to configure a session property.

Example:

```
SET state.backend=rocksdb
```

You can review all of the configurable parameters on the Properties tab of the **Session** window.

»

🕒 Session

🔍 Search Reset Session

General Properties Environment Variables User Defined Variables

atlas.collection.enabled	true
execution.attached	true
execution.buffer-timeout	100
execution.checkpointing.externalized-checkpoint-retention	RETAIN_ON_CANCELLATION
execution.checkpointing.max-concurrent-checkpoints	1
execution.checkpointing.min-pause	0
execution.checkpointing.mode	EXACTLY_ONCE
execution.checkpointing.snapshot-compression	false
execution.checkpointing.timeout	60000
execution.job-listeners	org.apache.atlas.flink.hook.FlinkAtlasHook

2. Click Execute.

Executing SQL jobs in production mode

As by default the SQL jobs are running in a session cluster, there is a risk in case of a cluster failure that every job is affected within that cluster. However, you can set a per-job production mode in SQL Stream Builder to create a dedicated environment for your production jobs.

Production mode means that the deployed SQL job (Flink job) runs in a separate SQL session and on a dedicated YARN cluster configured specifically for that particular job. You can set the execution mode in the **Job Settings** window, although the YARN execution target can be overwritten using a SET statement in the SQL Editor.

Setting production mode in Job Settings

To set the production mode for your SQL jobs in the **Jobs Setting** page, you need to select the execution mode with the following steps:

1. Click Job Settings at the **SQL Editor**.
2. Select PROD - Job runs in isolated Cluster at Execution Mode.
3. Add a SQL statement to the SQL Editor you want to execute.
4. Click Execute.

Setting YARN execution target in SQL Editor

To set the YARN execution mode for your SQL jobs in the **SQL Editor**, you need to use the SET statement, and execute the SQL query after defining the execution target and properties in the same window:

```
set 'execution.target' = 'yarn-per-job';
set 'logging.configuration.file' = '/tmp/log4j.properties';
select * from datagen_table_1631781644;
```

In the above example, the execution target is set to per-job to create a new YARN application for the job. Setting the execution target to per-job allows you to have an individual cluster for the specific job. The additional properties that you configure using the SET statement overwrites the properties that are configured for the running session. However, when you set properties for the production mode, the settings of the session cluster are not affected.



Note: You can override the execution mode set in the **Job Settings** window using the SET statement in the **SQL Editor**.

1. Set the execution mode to per-job.

```
set 'execution.target' = 'yarn-per-job';
```

2. Add additional configuration to the production job.
3. Add a SQL statement you want to execute.

Example:

```
set 'execution.target' = 'yarn-per-job';
set 'state.backend' = 'rocksdb';
select * from faker_table_1631781644;
```

4. Click Execute.

Functions

Creating Javascript User-defined Functions

With SQL Stream Builder, you can create user functions to write powerful functions in JavaScript that you can use 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 JavaScript. When you write them, you create a library of useful functions.

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.
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.

User Defined Function



Properties

Name *

Description

Enter text ...

Output Type *

Input Types

Add Input Type

Function (JavaScript) Usage: 0

```
1 function myFunction(input){
2   | return "Hello World 2";
3   }
4
5 myFunction($p0); // this line must exist
```

Cancel

Create

5. Add a Name to the UDF.

For example, name the UDF to HELLO_WORLD.
6. Add a Description to the UDF.

7. Select the Output and Input data type.

For example, select STRING.

8. Paste the JavaScript code to the editor.

For example:

```
// check to see if the card is VISA
function HELLO_WORLD(card){
  var cardType = "Other";
  if (card.charAt(0) == 4){
    cardType = "Visa";
  }
  return cardType;
}
HELLO_WORLD($p0); // this line must exist
```

9. Click Create.

10. Once created, you can use a User Function in your SQL statement:

For example:

```
-- simple usage
SELECT HELLO_WORLD(card) AS IS_VISA
FROM ev_sample_fraud;

-- in the predicate
SELECT amount, card
FROM ev_sample_fraud
WHERE HELLO_WORLD(card) = "Visa";
```



Note: Valid inputs can be a field in the source virtual table or any other valid input. Functions must be in upper case.



Note: User Functions have access to the Java 8 API, this increases the overall usefulness and power. For example:

```
function GETPLANE(icao) {
  try {
    var c = new java.net.URL('http://yyyyyy.io' + icao).openConnection();
    c.requestMethod='GET';
    var reader = new java.io.BufferedReader(new java.io.InputStreamReader(c.inputStream));
    return reader.readLine();
  } catch(err) {
    return "Unknown: " + err;
  }
}
GETPLANE($p0);
```

Developing JavaScript functions

When developing JavaScript functions that are more complicated than just simple logic, it is recommended to use the `jjs` command-line utility to create and iterate while writing functions.

About this task

After the function performs the required task, migrate it to the console. Additionally these files/functions can be saved in a source code control system like git/Github.

Procedure

1. Create a file for your function.



Note: It is recommended to name the file with the same name as that of the function.

2. Create some sample input when calling the function.
3. Call `jjjs` on the command line to test the function.

```
$>cat TO_EPOCH.js
function TO_EPOCH(strDate) {
  var strFmt = "yyyy-MM-dd HH:ss:mm";
  var c = new java.text.SimpleDateFormat(strFmt).parse(strDate).getTime()
/1000;
  return c.toString();
}

print(TO_EPOCH("2019-02-02 22:23:13"));

then
$>jjs TO_EPOCH.js
1549167203
```

What to do next

After you have successfully developed the JavaScript code, copy and paste only the function to your code window when creating the JavaScript function in SQL Stream Builder.

Creating Java User-defined functions

You can create User Defined Functions (UDF) using Java after manually adding the UDF function JAR file that contains the UDF class to the Flink connectors. After uploading the JAR file on the Streaming SQL Console, you can use the Java UDFs for your SQL jobs.

Procedure

1. Create a Java UDF function JAR file.

You can use the following example as a sample:

```
package udf;
import org.apache.flink.table.functions.ScalarFunction;

public class UdfTest extends ScalarFunction {
  public String eval(String input){
    return "Hello World " + input;
  }
}
```


2. 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.

3. 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.

4.



Click  next to **Functions**.

5. Click Upload JAR.
6. Click Choose File.
7. Search and select the Java UDF function JAR file.
8. Click Create.

Results

The selected JAR file is uploaded, and listed under **Functions** and ready to be used in your SQL queries.

Using System Functions

The same set of system functions can be used for 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.

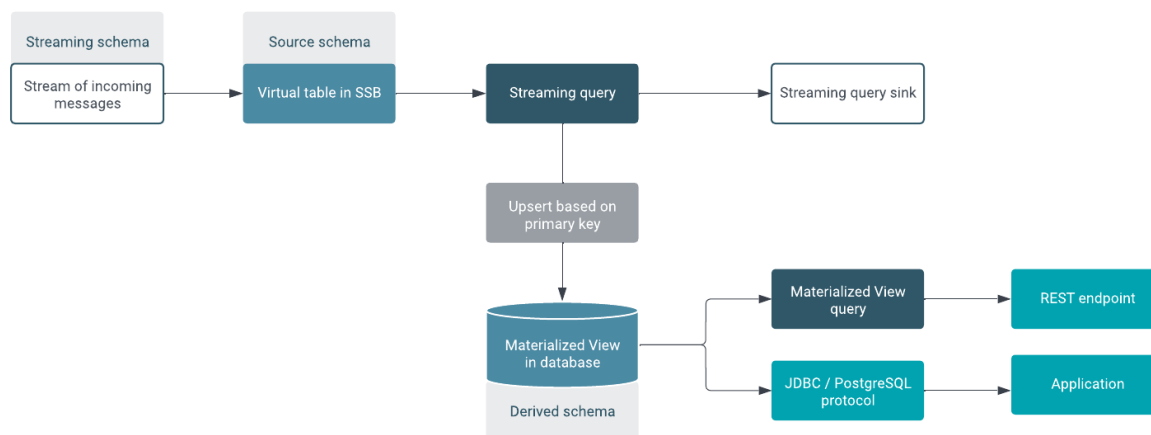
- [Comparison Functions](#)
- [Logical Functions](#)
- [Arithmetic Functions](#)
- [String Functions](#)
- [Temporal Functions](#)
- [Conditional Functions](#)
- [Type Conversion Functions](#)
- [Collection Functions](#)
- [JSON Functions](#)
- [Value Construction Functions](#)
- [Value Access Functions](#)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

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

Introduction to Materialized Views

SQL Stream Builder (SSB) has the capability to materialize results from a streaming SQL query to a persistent view of the data that can be read using REST API/endpoints. Business Intelligence tools and applications can use the Materialized View REST endpoint to query streams of data without deploying database systems. In addition to REST, Materialized Views can also be queried using JDBC/ PostgreSQL wire protocol.

Materialized Views are in synchronization with the mutating stream - they are updated by a primary key as data flows through the system. The data is updated by a given key, and it represents the latest view of the data by key. The following illustration shows the process of a Materialized View query in streaming SQL.



The following schemas are defined in the illustration:

- Streaming schema: refers to what SSB knows from the source provider, for example Kafka
- Source schema: refers to the schema of the table defined by the catalog in SSB
- Derived schema: refers to the schema of the Materialized View that is created in the database as a result of the streaming query against the catalog table

For example: vehicleID Z latest latitude and longitude is X and Y. As the vehicle moves, the latitude and longitude for the vehicleID are updated. The primary key is defined at creation time and is immutable.

Materialized Views can be created as mutating snapshots of the queried data result that is updated by a given key. The data is always the latest representation of itself by key (analogous to a primary key in most RDBMS systems).

You can query the Materialized Views using a GET request over REST, which returns a JSON response as "Content-Type: application/json". The queries are not defined at query time. Rather, they are curated, saved, and granted access through the Cloudera platform. You can configure a REST endpoint to query the Materialized View. Multiple query conditions can be created to allow various ways to query the same data. This is sometimes referred to as a 'pull query'.

Null Keys

In situations where information about a specific item is updated incrementally, for example, data about something is received in multiple messages, there may be keys missing from the incoming messages (for example, the streaming schema). In this case SSB continues to consume these messages, but marks the missing key as NULL at the sink, therefore SSB will also update the Materialized View with an upsert including the NULL values. To change this behavior, the "ignore null" option can be used, which tells SSB to update the relevant record only with the non-null values of the incoming message. Similarly, when a key is removed from the source schema, but not from the streaming schema, SSB ignores the key on the stream of incoming messages.

Creating Materialized Views

After executing a SQL Stream job, you can set up the Materialized Views to have a snapshot of your queried data. You can use the URL Pattern from the Materialized View to visualize the generated data.

Before you begin

- Make sure that PostgreSQL is installed and configured to SQL Stream Builder (SSB) to create Materialized Views.

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.

2. Click Create Job or select a previous job on the **Getting Started** page.

You are redirected to the **Console** page. The Materialized View button will be available when you add a query to the SQL Editor.

3. Click Materialized View.

The screenshot shows the 'Materialized View' configuration page. At the top, there's a title 'Materialized View'. Below it is a 'Configuration' section. A warning message in an orange box says 'Please select or create an API Key'. The configuration includes several fields: 'Primary Key' (a dropdown menu), 'Retention (Seconds)' (a text input with '300'), 'Min Row Retention Count' (a text input), and 'API Key' (a dropdown menu). To the right of these fields are three toggle switches: 'Enable MV', 'Recreate on Job Start', and 'Ignore NULLs'. Below the configuration fields are three icons: a plus sign, a pencil, and a trash can. At the bottom, there's a 'Queries' section with a button '+ Add New Query' and a message box that says 'No Queries'.

4. Switch Enable MV toggle.
5. Select a Primary Key.

If this list is empty, then no SQL is specified in the **SQL Editor** or the SQL query is invalid. Select a key as a primary key for the Materialized View. All data will be updated by this key.
6. Select a Retention Period.

Data not being mutated during this period is removed from the view.
7. Enable or disable Recreate on Job Start.

If enabled, the Materialized View is deleted when a job is started or restarted.
8. Enable or disable Ignore NULLS.

If enabled, NULL values will NOT update values that are non null - they are ignored.

9. Select an API Key.

In case there are no API Keys, click Add API key, or click Materialized Views from the main menu. The add API key window appears. Provide a name for the API key, and click Save Changes.

API Key

×

Key Value

0862eb36-70cf-4ad9-840a-0a3d175cc4ca

Key Name

Cancel Save

To check your created API keys:

a. Click Materialized Views from the main menu.

The Materialized Views page appears.

b. Select API Keys tab.**10. Click Add New Query to create the Materialized View query.**

The Materialized View Query Configuration window appears.

Query Editor

×

URL /api/v1/query/5195/


URL Pattern *

Description (Optional)

Columns Filters

Select Column

Select Column Select All Unselect All

 No Columns Selected

Cancel Apply Apply and Save Job

11. Provide a name in the **URL Pattern** field.
12. Provide a description of the Materialized View Query, if needed.

13. Customize the Materialized View in the Query Editor.

Query Editor

URL

/api/v1/query/5195/test

URL Pattern *

test

Description (Optional)

ColumnsFilters

Select Column

Select ColumnSelect AllUnselect All

Name	Alias	Type	
col_str	col_str	VARCHAR	
col_int	col_int	INTEGER	
col_ts	col_ts	TIMESTAMP_WITHOUT_TIME_ZONE	

Cancel

Apply

Apply and Save Job

a) Select the columns of the SQL job you want to use in the Materialized View Query.
You can select every column in the table by clicking on the Select All button.

b) Click Filters tab to apply computations and further enrichment of your data.

67

Query Editor ✕

URL `/api/v1/query/5195/test`

URL Pattern * Description (Optional)

Columns Filters

AND OR + Rule + Ruleset

Field	Operator	Value
col_str	equal	

Cancel Apply Apply and Save Job

c) Provide details of the filter by selecting the Field, Operator and Value.

You can add more rules or set a ruleset for the query.

14. Click Apply or Apply and Save Job.

The URL for the Materialized View appears under the **Queries** header.

Materialized View

Configuration

Primary Key ☒ Enable MV

Retention (Seconds) ☐ Recreate on Job Start

Min Row Retention Count

API Key ☐ Ignore NULLs

+ ✎ ✖

Queries + Add New Query

`/api/v1/query/5195/test?key=0862eb36-70cf-4ad9-840a-0a3d175cc4ca` 📄 ✎ ✖

Results

You can click the created REST endpoint to review the data, or copy it and visualize the queried data in a Business Intelligence tool, notebook, code and so on. You can also review the list of Materialized View and API keys on the Materialized View page.

Configuring Retention Time for Materialized Views

When creating Materialized Views, you can configure how the system should retain the data rows in the Materialized Views. You can either choose between retaining the data by time or the row count.

The Materialized Views configuration allows you to set one of the following configuration parameters for data retention:

- Retention Time
- Min Row Retention Count

You can specify the Retention Time and Min Row Retention Count when creating a Materialized View for a SQL job on the Compose page of the Streaming SQL Console.

Retention Time

Retention Time is specified in seconds, and it tells the system to retain data rows as old as the specified retention time. The rows that are outside of the retention time are removed from the Materialized View.



Note: You can only add a Retention Time value, if the Min Row Retention Count field is empty or set to 0.

The following example shows how to set a Retention Time of 300 seconds. This means that only those rows are included in the Materialized View that are within the 300 seconds of the job execution. The older rows are removed from the Materialized Views.

The screenshot shows the 'Materialized View' configuration page. At the top left is a search icon and the title 'Materialized View'. At the top right is a 'Revert Changes' button. Below the title is the 'Configuration' section. It contains several fields and controls:

- Primary Key**: A dropdown menu with 'col_str' selected. To its right is an edit icon (pencil).
- Retention (Seconds)**: A text input field containing '300'.
- Min Row Retention Count**: A text input field containing '0'.
- API Key**: A dropdown menu with 'test' selected. To its right is an edit icon (pencil).
- Enable MV**: A toggle switch that is turned on (blue).
- Recreate on Job Start**: A toggle switch that is turned off (grey).
- Ignore NULLs**: A toggle switch that is turned off (grey).

At the bottom left of the configuration area are two buttons: a blue '+' button and a red trash can icon.

Minimum Row Retention Count

The Min Row Retention Count parameter indicated to the system to maintain a specific number of data rows in the Materialized View.



Note: You can only add a Min Row Retention Count value, if the Retention Time field is empty or set to 0.

The following example shows how to configure the system to retain the last 5000 data rows. This means that only the first 5000 data rows are included in the Materialized View, and the data rows from 5001 are removed from the Materialized View.

Materialized View Revert Changes

Configuration

Primary Key ⓘ
col_str ⓘ

Retention (Seconds) ⓘ
[Empty field] ⓘ

Min Row Retention Count ⓘ
5000 ⓘ

API Key ⓘ
test ⓘ

+ 🗑️

Enable MV ⓘ Recreate on Job Start ⓘ
Ignore NULLs ⓘ

Retaining all data without limit

In order to retain all data rows, regardless of time, or number of rows in the Materialized View, both settings can be reset to zero (0). This indicates to the system that all data must be preserved without a time limit.

The following example shows how to configure the Retention parameters to keep all of the data regardless of time:

Materialized View Revert Changes

Configuration

Primary Key ⓘ
col_str ⓘ

Retention (Seconds) ⓘ
0 ⓘ

Min Row Retention Count ⓘ
0 ⓘ

API Key ⓘ
test ⓘ

+ 🗑️

Enable MV ⓘ Recreate on Job Start ⓘ
Ignore NULLs ⓘ

Materialized View Pagination

You can set a limit and order the results of a Materialized View query by adding values to the limit and offset configurations.

After creating a Materialized View, you can order and limit the results after opening the Materialized View. The default limit of the results is 100 entries. You can set the limit and offset using the REST URL. You need to modify the URL address in your browser by setting the value and the offset value as shown in the following example:

```
localhost:18131/api/v1/query/5275/summary_query?key=6ef33a7a-c82a-4a72-946b-5fe0b33e033c&limit=20&offset=3
```

When you edit the Materialized View, the set limit and offset change based on the new filters set for the Materialized View query. If you set an invalid number for the limit and offset, an error message is displayed.

Using Dynamic Materialized View Endpoints

You can use static or dynamic REST endpoints when creating Materialized Views in SQL Stream Builder. After setting filters for the Materialized View query, you can further filter down the results by using variables in the endpoint URL.

Difference between static and dynamic endpoints

When using a static endpoint, the endpoint URL serves as a constant string, the process of filtering is constant and final. The results are displayed based on the configurations and filters of the query. In case of a static endpoint, you provide the information in the URL pattern field only with a reference purpose. For example, in the following static endpoint the Filter is set to *id greater than 0*, and the endpoint URL is *positive/id*:

Query Editor
×

URL /api/v1/query/5197/positive/id

URL Pattern * ⓘ

positive/id

Description (Optional)

Columns Filters

Select Column

Select Column

Select All

Unselect All

Name	Alias	Type	
id	id	INTEGER	
power	power	VARCHAR	
age	age	INTEGER	

Cancel

Apply

Apply and Save Job

In case of a dynamic endpoint, you have the option to set dynamic variables in the endpoint URL. This means that when providing the information in the URL pattern field, you also include a generic variable that later can be specified when using the REST endpoint. With dynamic endpoints, you can further filter the queried results based on the given variable. When creating the dynamic endpoint, you need to make sure that the variable is surrounded by curly braces to signal that it is a variable as *minimumId* in the example shown below. You also need to reference the variable in the same way when setting the Filters for the Materialized View Query. For example, in the following dynamic endpoint the Filter is set to *id greater than {minimumId}*, and the endpoint URL is *id/bigger/than/{minimumId}*:

Query Editor ✕

URL /api/v1/query/5197/id/bigger/than/{minimumId}

URL Pattern * ⓘ Description (Optional)

id/bigger/than/{minimumId}

Columns Filters

AND

OR

+ Rule

+ Ruleset

Field	Operator	Value
<div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">id</div>	<div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">greater</div>	<div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">{minimumId}</div>

Cancel

Apply

Apply and Save Job

Using dynamic endpoints

You can use a static endpoint by clicking on it on the Materialized View page or by copying the URL and pasting it to a browser, application or tool. As the endpoint is static, the URL can be invoked and the aspects of filtering is constant. However, as a dynamic endpoint contains a variable, you need to replace it by an actual value. This value makes the endpoint and the filtering dynamic. In the following example, a static and dynamic endpoint is compared:

URL Pattern	Description
/api/v1/query/5424/id/bigger/than/{minimumId}?key=225342de-8588-4db9-8bfc-e52d9dccf34f	
/api/v1/query/5424/postive/id?key=225342de-8588-4db9-8bfc-e52d9dccf34f	

When you click on a dynamic endpoint, you are prompted to provide a value for the variable as shown in the following illustration:

Dynamic endpoint

Please specify the values of the variables in this dynamic endpoint:

minimumId

Close

Go

After entering a value, the filtered results are shown in a new browser tab:

```

<  →  ↻  Not Secure | mgergely-1.vpc.cloudera.com:18131/api/v1/query/5424/id/bigger/than/0?key=225342de-8588-4db9-8bfc-e52d9dccf34f
[{"id":8,"power":null,"age":561}, {"id":5,"power":"Endurance","age":390}, {"id":3,"power":null,"age":29}, {"id":4,"power":"Size Changing","age":920}, {"id":1,"power":"Flight","age":null}, {"id":2,"power":"Symbiote Costume","age":457}, {"id":6,"power":"Mind Control","age":284}, {"id":7,"power":null,"age":104}, {"id":9,"power":"Durability","age":336}]

```

The advantage of a dynamic endpoint is that by providing a variable in the URL, you are able to narrow the queried results to the value you exactly need from that dataset. In the following example, the *minimumId* variable is set to 5, so the "id" of the shown entries are bigger than 5:

```

<  →  ↻  Not Secure | mgergely-1.vpc.cloudera.com:18131/api/v1/query/5424/id/bigger/than/5?key=225342de-8588-4db9-8bfc-e52d9dccf34f
[{"id":8,"power":"Levitation","age":659}, {"id":9,"power":"Astral Trap","age":204}, {"id":7,"power":"Summoning","age":681}, {"id":6,"power":"Omnitrix","age":758}]

```

Using multiple variables in one endpoint

You can use multiple variables in a dynamic endpoint (*/id/between/{minimumId}/{maximumId}*), and you can also use the variables for different filters within a Materialized View query. In this case, the variables are replaced at every occasion by the same value you provide when using the dynamic endpoint.

Using SQL Stream Builder with Cloudera Data Visualization

You can create Business Intelligence reports from the Materialized Views created in SQL Stream Builder (SSB) using Cloudera Data Visualization. To integrate SSB with Data Visualization, you need to provide the PostgreSQL database information of SSB in Data Visualization.

About this task

After creating Materialized Views of your SQL Stream job, you can create visualized reports from your queried result with Cloudera Data Visualization.

Cloudera Data Visualization enables you to explore data and communicate insights by using visual objects. You can connect to your data in Cloudera Data Platform (CDP), create state-of-the-art visualizations on top of your datasets, build informative dashboards and applications, and publish them anywhere across the data lifecycle.

When connecting SSB with Cloudera Data Visualization, you need to provide the PostgreSQL database information that stores the Materialized View data in the Cloudera Data Visualization web interface. For the connection, you can find the PostgreSQL database information in Cloudera Manager.

Before you begin

You have created a Materialized View query on your running SQL job.

Procedure

1. Go to your cluster in Cloudera Manager.
2. Click SQL Stream Builder from the list of services.
3. Click Configuration.
4. Filter down the configuration parameters to Materialized View Engine.

The list of Materialized View Engine parameters are displayed.

5. Save the ssb.mve.datasource related information.

Regarding the PostgreSQL information you need for the connection, the Database URL (JDBC) configuration parameter contains the PostgreSQL hostname, the PostgreSQL port and the PostgreSQL database name.

The screenshot shows the Cloudera Manager configuration interface. On the left, a 'Filters (1)' sidebar is visible with 'SCOPE' and 'CATEGORY' sections. The 'SCOPE' section lists 'SQL_STREAM_BUILDER-1 (Se...', 'Load Balancer', 'Materialized View Engine', and 'Streaming SQL Engine'. The 'CATEGORY' section lists 'Main', 'Advanced', and 'Database'. The main configuration area shows three parameters: 'Database URL (JDBC)' with value 'jdbc:postgresql://docstest-1.docstest.root.hwx.site:5432/ssb_mve', 'Database User' with value 'ssb.mve.datasource.username', and 'Database Password' with value 'ssb.mve.datasource.password'. Each parameter has a 'Materialized View Engine Default Group' dropdown set to 'Undo'.

When creating a data connection in Cloudera Data Visualization, you need to provide the connection information and credential as shown in the following example:

- Database Name: ssb_mve
- Database Host: docstest-1.docstest.root.hwx.site
- Database Port: 5432
- Database User: ssb_mve
- Database Password: [****POSTGRES SQL DATABASE PASSWORD****]

The Database User and Database Password for the Materialized View Engine is configured based on what username and password was provided when installing the SQL Stream Builder service on your cluster.

What to do next

After collecting the necessary information to the connection, you need to access the Cloudera Data Visualization web interface and create a connection to the PostgreSQL database of SSB using the SQL Stream Builder connector. For more information, see the [Cloudera Data Visualization](#) documentation.



Important: The SQL Stream Builder connector is provided as a technical preview at this time in Cloudera Data Visualization. The tool is still under development and not recommended for a production environment.

Using SQL job notification

You can set up notifications for SQL jobs using Streaming SQL Console to alert individuals and teams about job failures. The notifications include email and webhook alerts, which you can also group together for easier organization.

You can add job notifications to SQL jobs to send alerts and information when the status of the job is changed. The information in the notification can include the job name, job ID, cluster ID, last exception, and Flink job ID.

You can create the following notification actions for SQL jobs:

Email

Sending notification email to the provided email address

Webhook

Sending notification to a webhook address using POST and PUT

You can also collect the Email and Webhook notifications into a **Group** to organize the notifications. These groups do not trigger any action, but serve as an organizational element.

After creating and adding the notification to a SQL job, anytime a change occurs for that job, a notification is sent to the specified address.

Enabling job notifications

Before you are able to use the job notifications, you need to enable it in Cloudera Manager and based on the notification type configure the email and webhook parameters.

1. Go to your cluster in Cloudera Manager.
2. Select SQL Stream Builder from the list of services.
3. Click Configuration.
4. Search for *Enable job notifications functionality*.
5. Enable the job notifications by checking the checkbox.

The following parameters also need to be configured for the SQL Stream Builder (SSB) service in Cloudera Manager based on the type of the notification:

Configuration	Description
Job notifications monitoring interval	Sets the interval of the job monitoring in seconds.
Mail server host for job notifications	The host of the SMTP server for job failure notifications.
Mail server username for job notifications	The username to access the SMTP server for job failure notifications.
Mail server password for job notifications	The password to access the SMTP server for job failure notifications.
SMTP authentication for job notifications	Enable SMTP authentication for job notifications.
StartTLS for job notifications	Use the StartTLS command to establish a secure connection to the SMTP server for job notifications.
Job notifications sender mail address	Sender mail address for job notifications.
Mail server port for job notifications	The port of the SMTP server for job failure notifications.
Job notifications webhook sender parallelism	Number of threads used by the job notification task to call user-specified webhooks when notifying about a failed or missing job.



Note: In case the YARN cluster of a failed job no longer exists, SSB attempts to fetch the information about the failed job from the Flink History Server. If information cannot be fetched by the first attempt, SSB attempts to fetch the information again based on the interval configured for the History server fetch retry time parameter.

Creating job notifications

After enabling the job notification function, you can create email, webhook and group notifications with Streaming SQL Console.

1. Navigate to Streaming SQL Console.
 - a. Go to your cluster in Cloudera Manager.
 - b. Select SQL Stream Builder from the list of services.
 - c. Click Streaming SQL Console.

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

2. Open a project from the **Projects** page.
 - 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 **Job Notifications**.

4. Select from the following options based on which notification type you want to use:

For Email

- a. Click New Email Notification .

The **Create Email Notification** window appears.

Create Email Notification

Name *

Type

EMAIL

Email address *

Subject ⓘ

Message template ⓘ

Save

- b. Provide a Name for the notification.
- c. Add the Email address you want to send the notification.
- d. Provide a Subject.
- e. Provide a Message.

The message of the Webhook notification must be a valid JSON.

You can add placeholder items to the payload that will be converted into the specific job information. You can add the placeholders in \${placeholder} format, and the following placeholders are available:

- jobName
- jobStatus
- jobStatusDescription
- ssbJobId
- flinkJobId
- clusterId
- lastException

- f. Click Save.

For Webhook

- a. Click New Webhook Notification .

The **Create Webhook Notification** window appears.

Create Webhook Notification

Name *

Type

WEBHOOK

Method * POST

Webhook address *

Payload template ⓘ

HTTP Headers

Header Name	Value

Save

- b. Provide a Name for the notification.
- c. Select between POST or PUT method.
- d. Provide a Webhook address.
- e. Provide a Payload to the webhook.



Warning: The payload must be a valid JSON object.

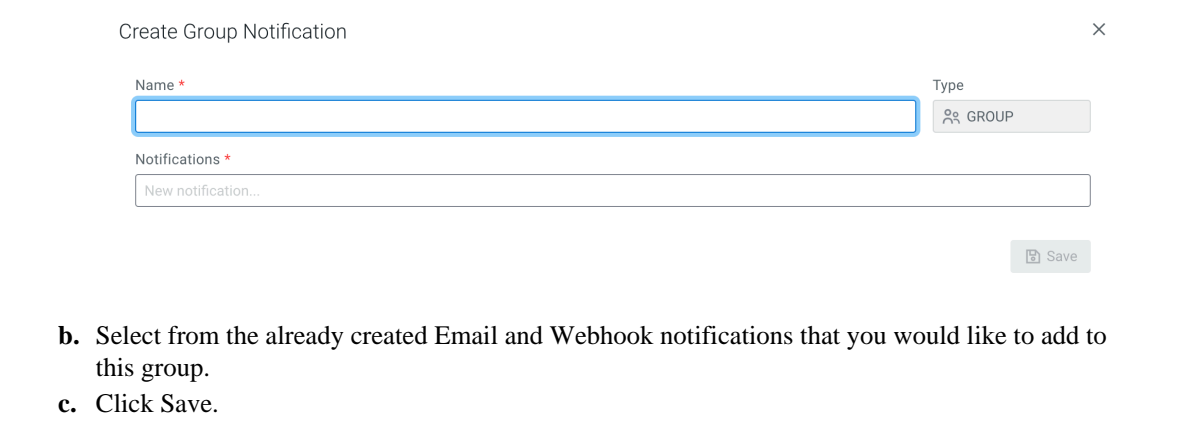
You can add placeholder items to the payload that will be converted into the specific job information. You can add the placeholders in `${placeholder}` format, and the following placeholders are available:

- `jobName`
 - `jobStatus`
 - `jobStatusDescription`
 - `ssbJobId`
 - `flinkJobId`
 - `clusterId`
 - `lastException`
- f. Add **HTTP Headers** if needed by specifying the Header Name and Value.
 - g. Click Save.

For Groups

- a. Click New Group Notification .

The **Create Group Notification** window appears.



Create Group Notification [X]

Name *

Type

Notifications *

b. Select from the already created Email and Webhook notifications that you would like to add to this group.

c. Click Save.

Enabling notifications for a SQL job

After creating a job notification, you can enable it for a selected job from the **Job Settings** window.

1. Select Job Settings from the **SQL Editor**.
2. Click Job Notifications.
3. Search for the name that you have provided when creating the notification.
4. Click on the notification.

The selected notification is added to the job and the notification is sent with the information provided in the message template.

Using SQL Stream Builder REST API

You can use the REST API to monitor, manage and configure the SQL Stream jobs with GET, POST and DELETE HTTP methods. You can use the SQL Stream Builder (SSB) REST API in command line, import them to REST API Tools or by accessing the Swagger UI.

The following HTTP methods are available for SSB:

- GET to query information about the specified endpoint
- POST to create resources for the specified endpoint
- PUT to update existing resources for the specified endpoint as a whole
- PATCH to partially update existing resources for the specified endpoint
- DELETE to remove objects from the specified endpoint

The [REST API Reference document](#) contains the available endpoints for SQL Stream Builder.

You can also reach the SSB REST API reference document from on the main menu of Streaming SQL Console.

You can use the SSB REST API with Command Line Interface (CLI), you can also import the REST API swagger.json file to a REST API Tool, for example Postman, and you also have the option to use the REST API with the Swagger User Interface (UI).

The Streaming SQL Engine API details the following operations for SQL Stream Builder and Flink:

- Heartbeat
- Admin Operations
- User Operations
- User Keytab Operations
- Data Source Operations
- Table Operations

- Project Operations
- Project Environment Operations
- Project Sync Operations
- Project Invitation Operations
- Project Permission Operations
- SQL Operations
- SSB Session Operations
- Job Operations
- Sampling Operations
- Flink Job Operations
- Flink Session Cluster Operations
- Artifact Operations
- UDF Artifact Operations
- UDF Operations
- API Key Operations
- Connector Operations
- Data Format Operations
- Diagnostic Operations

Using REST API with CLI

When using the SSB REST API with CLI Tool, you need to create the POST or GET commands with curl, and also include the Streaming SQL Engine hostname, port and the required operation for the endpoint. The submitted commands return the information, or complete the process you have requested and display the status in the CLI.

The following examples show a secured and unsecured GET method to list the SQL jobs:

For Unsecured

```
$ curl -H 'Username: <your_username>' \
      'https://<cluster_domain>:<streaming_sql_engine_server_port>/api/
v1/ssb/jobs'
```

For Secured with SPNEGO

```
$ kinit <your_user>
$ curl -ik --negotiate -u: \
      'https://<cluster_domain>:<streaming_sql_engine_server_port>/api/
v1/ssb/jobs'
```

For Secured with Knox

```
$ curl -ik --negotiate -u <knox_username>:<knox_password> \
      'https://<cluster_domain>:<streaming_sql_engine-server_port>/gat
eway/cdp-proxy-api/ssb-sse-api/api/v1/ssb/jobs'
```



Note: When using the unsecured endpoints, you must provide the Username HTTP header for the command as an authentication method to indicate which user sends the request.

Using REST API Tool

When using REST API Tools, you need to connect to the cluster that hosts the SQL Stream Builder REST API. You can also use the following URL in a browser to download the swagger.json file when you need to manually upload the REST API endpoints for a Tool.

```
http(s)://<cluster-domain>:<streaming-sql-engine-server-port>/swagger/api-docs/public
```

Using Swagger UI

You can access the Swagger UI to use the SSB REST API by adding your hostname and the Streaming SQL Engine port to the Swagger UI URL.

```
http(s)://<cluster-domain>:<streaming-sql-engine-server-port>/swagger-ui/index.html?configUrl=/swagger/api-docs/swagger-config
```

You can also download the swagger.json for the SSB REST API using the Swagger UI.

Monitoring SQL Stream jobs

You can use the Streaming SQL Console to review the status, properties and log of your SQL Stream jobs executed in SQL Stream Builder. Using the Flink Dashboard, you can also monitor the Flink job that is submitted when you execute a SQL query.

Using the Streaming SQL Console

When using the **Jobs** tab in Streaming SQL Console to monitor your SQL jobs, you can review the ID, the Name, the Start time, the State of the submitted jobs, and the User who submitted the SQL job.

1.



Click next to **Jobs**.

2. Select Manage.

By default, all of the jobs are displayed on the **Jobs** tab.

Name	Created	Username	ID	State
keen_snyder	2022-11-28 18:09:47 (6 hours ago)		5195	
nervous_edison	2022-11-28 22:53:39 (1 hour ago)		5196	
silly_varahamihira	2022-11-28 22:54:51 (1 hour ago)	admin	5197	
zen_mclintock	2022-11-28 23:44:30 (18 minutes ago)		5198	

3.



Click .

4. Select Events.

The **Events** window appears detailing the Flink job information that can be further filtered based on the log level settings.

Events of nifty_brown

Updated at 28/09/2022, 15:30:36

Copy

Export

Time	Level	Message
2022-09-27T17:01:57.345	INFO	Successful job deployment.
2022-09-27T17:01:57.349	INFO	Flink JobId: a3d84394c6522b205795672b3a72ee0f
2022-09-28T10:02:26.803	INFO	Successful job deployment.
2022-09-28T10:02:26.805	INFO	Flink JobId: 8eb4242f16ac93292fe5a12d1e7104e4
2022-09-28T10:03:23.94	INFO	Successful job deployment.
2022-09-28T10:03:23.942	INFO	Flink JobId: 7ad21a7460ea7e481c58c4e1674aa702
2022-09-28T10:05:58.054	INFO	Successful job deployment.
2022-09-28T10:05:58.056	INFO	Flink JobId: 08dd2c036048ae6d63d862340e1aa721
2022-09-28T10:17:08.83	INFO	Successful job deployment.
2022-09-28T10:17:08.831	INFO	Flink JobId: 6b64265f91c6d9974f09ed97b9fabc2b

1 to 10 of 10

<

>

Page 1 of 1

Close

History of SQL queries

You can review and reuse the SQL queries that were previously executed on the **SQL History** view of the Project Manager. When you click on one of the SQL queries, it is automatically imported to the SQL window for execution. You can filter the SQL queries by the time they were last run or by the user who run them.

SQL History **2**

Search in SQL history Reload

SQL	Details
<code>SELECT * FROM datagen_table_166...</code>	admin
2022-11-28 22:57:40 (1 hour ago)	
<code>CREATE TABLE `datagen_table_16...` `col_str` STRING, `col_int` INT, `col_ts` TIMESTAMP(3),</code>	admin
2022-11-28 22:56:17 (1 hour ago)	

Using the Flink Dashboard

You can also monitor your running SQL jobs using the Flink Dashboard. You can easily reach the Flink Dashboard from the main menu of Streaming SQL Console or from the SQL Editor.

Collecting diagnostic data

You can collect diagnostic data from SQL Stream Builder (SSB) and Flink using Cloudera Manager, and send it to Cloudera Support for easier troubleshooting.

When using SSB and Flink, Cloudera Manager automatically collects the usage and diagnostic data from the services. The collected data is analyzed to further improve the software, and to help you in support cases when you encounter an error. You can configure the frequency of data collection, and you can also manually trigger sending the diagnostic data to Cloudera.

For more information about the diagnostic data collection and configuration, see the [Cloudera Manager documentation](#).

The following telemetry data is collected when using SSB and Flink:

- Number of users
- Number of jobs
- Number of registered data sources and connectors
- Number of Materialized Views
- Number of API keys
- Number of tables
- Number of catalogs
- Total tasks slots in use
- Log files from Streaming Engine and Materialized View Engine
- Log files from YARN to collect Flink Job and Task Manager processes

SSB metadata collection using Atlas

As SQL Stream Builder (SSB) is built on Flink, you can use the Apache Atlas and Flink integration, and collect the metadata of your Flink jobs submitted from SSB. Using Atlas, you can find, organize and manage different assets of data about your Flink applications and how they relate to each other.

To use Atlas with SSB, you need to create entity type definitions for Flink in Atlas, and enable the metadata collection in Cloudera Manager for Flink.

For more information, see the [Flink metadata collection using Atlas](#).

Flink SQL

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

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

Managing time in SSB

Time attributes define how streams behave for time based operations such as window aggregations or joins. For Kafka tables you can use the Event Time tab to create source provided or user provided timestamp and watermarks. For other tables you can define time related attributes in the Flink DDL or directly in the SQL query. You can use timestamps that are already provided in the source or you can use custom timestamps.

Source-provided timestamps

Source-provided timestamps are inserted directly into the data stream by the source connector. This query uses the source-provided `order_time` field to perform a temporal join on multiple Kafka topics:

```
-- Table of orders
CREATE TABLE orders (
  order_id      STRING,
  price         DECIMAL(32,2),
  currency      STRING,
  order_time    TIMESTAMP(3),
  WATERMARK FOR order_time AS order_time
) WITH (/* ... */);

-- Table of currency rates
CREATE TABLE currency_rates (
  currency      STRING,
  conversion_rate DECIMAL(32, 2),
  update_time   TIMESTAMP(3),
  WATERMARK FOR update_time AS update_time
) WITH (/* ... */);

-- Event time temporal join to enrich orders with currencies
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
```

User-provided timestamps

You can also specify timestamps contained in the data stream itself. For example, if your schema includes a field called `"order_time"`, it is possible to construct a query such as:

```
-- Table of orders
-- Converts order_time_string field to timestamp
CREATE TABLE orders (
  order_id      STRING,
  price         DECIMAL(32,2),
  currency      STRING,
  order_time_string STRING,
  order_time as to_timestamp(order_time_string),
  WATERMARK FOR order_time AS order_time
) WITH (/* ... */);
```

When an invalid timestamp is found in the stream (for example, NaN), the timestamp of the message is going to be 0. This way the message is excluded from the current window.

When your data does not include a timestamp in a suitable format, it is possible to compute a new timestamp column from another existing column using the Input Transform feature of SSB.

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"> Inserting columns or values into a table Adding OVERWRITE to overwrite the existing data in the table

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

Flink Queries

Flink SQL supports querying data with SELECT statements 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"> Selecting all data in a table Selecting data using different types of operations

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

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

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

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

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

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 serve 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 at an exact point of time: use the currency rates from 12:00.

For more information about 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. By 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 Source.
- 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"> order_number price order_time item_id
Kudu - ItemMeta	<ul style="list-style-type: none"> id info

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:

