02: Getting Started with Flink SQL

CONFLUENT
Global Education

Module Overview

This module contains five lessons:

- Why Flink SQL for Stream Processing?
- Flink SQL features
- First Flink SQL Queries:
  - SQL Statements
  - Operators
  - Data Types
- Streaming vs. batch in Flink SQL
- Flink SQL on Confluent Cloud

02a: Why Flink SQL for Stream Processing?

Description

Learning why Flink SQL is suitable for Stream Processing in comparison to traditional SQL solutions.

Why Flink SQL for Stream Processing?

Flink SQL simplifies complex data processing tasks using familiar SQL syntax for both streaming and batch processing

---

- Implementing Flink stream processing apps requires special skills:
  - Java/Scala experience
  - In-depth knowledge of streaming concepts like time and state
  - Knowledge of distributed data processing
- Everybody knows and uses SQL
- SQL queries are optimized and efficiently executed
- Unified syntax and semantics for batch & streaming data

Traditional SQL vs. Streaming SQL

Basically all tables that are processed with SQL queries change over time

Traditional SQL:

- Batch processing on static datasets
- Higher latency, data processed in batch
- Stateless processing model
- Often used for historical analysis and reporting

Streaming SQL:

- Continuous processing to incoming data
- Low-latency enabling real-time analytics
- Stateful processing model
- Often used for applications requiring timely insights, such as fraud detection or monitoring

The syntax of the SQL query is the same!

Basically all tables that are processed with SQL queries change over time, i.e. transactions from applications or bulk inserts from ETL processes.

The main differences between Traditional SQL and Streaming SQL are:

- Processing Model:
  - Traditional SQL: Designed for batch processing on static datasets. Queries are executed on a snapshot of data, and the results are computed over the entire dataset at once.
  - Streaming SQL: Tailored for real-time data processing. Queries are continuously applied to incoming data streams, providing real-time analytics and insights.
- Data Model:
  - Traditional SQL: Works with static, stored datasets typically found in databases or data warehouses.
  - Streaming SQL: Deals with dynamic, continuously changing data streams. Data is processed as it arrives, allowing for low-latency analysis of live data.
- Time Sensitivity:
  - Traditional SQL: Often used for historical analysis and reporting where the time factor is not critical.
  - Streaming SQL: Focuses on real-time or near-real-time processing, making it suitable

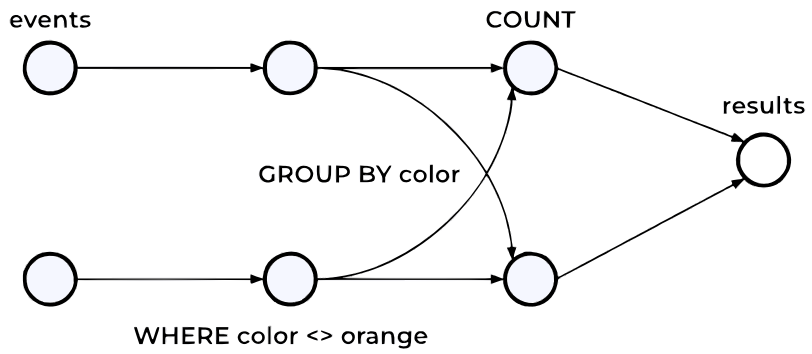
for applications requiring timely insights, such as fraud detection or monitoring.

- Query Semantics:
  - Traditional SQL: Expresses queries in a point-in-time manner, operating on a snapshot of data.
  - Streaming SQL: Supports temporal queries, enabling the analysis of data over time windows, event time, and processing time.
- Windowing and Aggregation:
  - Traditional SQL: Typically aggregates data over the entire dataset or partitions.
  - Streaming SQL: Supports window-based operations to aggregate and analyze data over specific time intervals or based on event counts.
- Infinite Data Streams:
  - Traditional SQL: Not inherently designed to handle infinite data streams and continuous data arrival.
  - Streaming SQL: Built to process unbounded data streams, allowing for continuous analysis without a defined endpoint.
- State Management:
  - Traditional SQL: Assumes a stateless processing model where each query is executed independently.
  - Streaming SQL: Requires stateful processing to maintain context and history across streaming data, often necessitating the use of stateful operators.
- Latency:
  - Traditional SQL: Typically involves higher latency as it processes data in batches.
  - Streaming SQL: Provides low-latency processing, enabling real-time analytics and quicker response times.

Both Traditional SQL and Streaming SQL share similar syntax, their underlying processing models and use cases are fundamentally different, catering to the needs of either batch processing on static data or real-time processing on dynamic data streams.

Streaming SQL Example

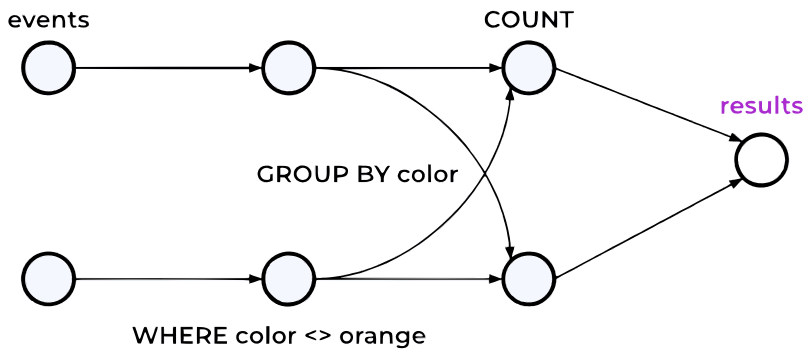
```
INSERT INTO results
SELECT color, COUNT(*)
FROM events
WHERE color <> orange
GROUP BY color;
```



The example we saw before can be implemented using Flink SQL – here's what that would look like.

Streaming SQL Example

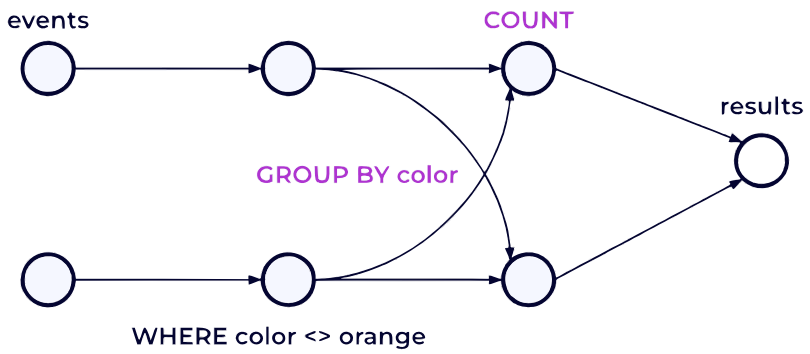
```
INSERT INTO results
SELECT color, COUNT(*)
FROM events
WHERE color <> orange
GROUP BY color;
```



The first line of this SQL statement – INSERT INTO results – sets up the results table so that it is used as the sink for the application that Flink will create to execute this query.

Streaming SQL Example

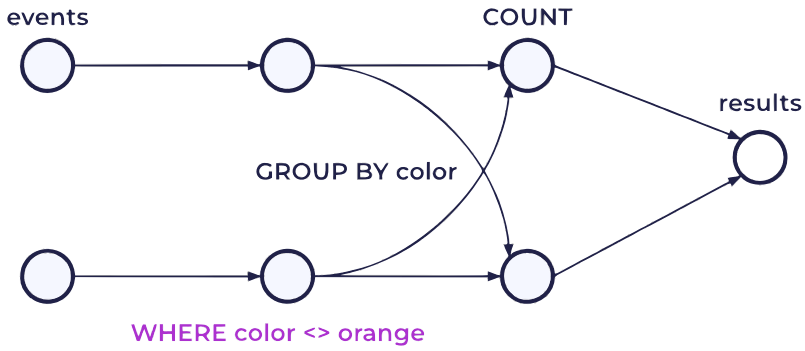
```
INSERT INTO results
SELECT color,
COUNT(*)
FROM events
WHERE color <> orange
GROUP BY color;
```



The SELECT, COUNT, and GROUP BY clauses sets up the part of the job that repartitions the stream by color, and counts the events of each color.

Streaming SQL Example

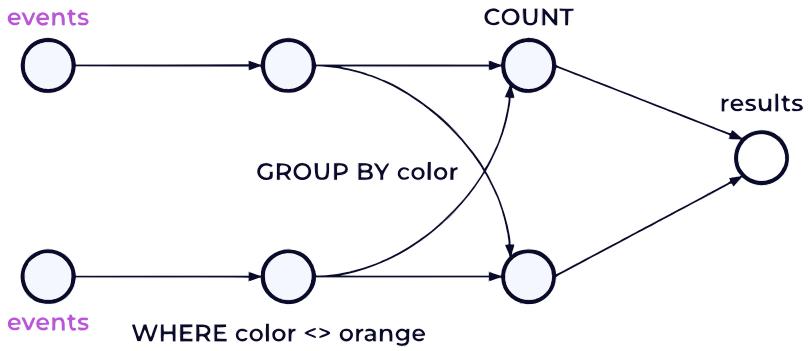
```
INSERT INTO results
SELECT color, COUNT(*)
FROM events
WHERE color <> orange
GROUP BY color;
```



WHERE color IS NOT EQUAL TO orange creates this filtering step in the job graph.

Streaming SQL Example

```
INSERT INTO results
SELECT color, COUNT(*)
FROM events
WHERE color <> orange
GROUP BY color;
```



And finally, the FROM events clause specifies the event source for this streaming application.

Lesson 02b

02b: Flink SQL Syntax

Description

Describing the Flink SQL Syntax. DDLs, DMLs, functions and data types.

Flink SQL - DDL Statements

Command	Description
CREATE TABLE	Define a new table in Flink SQL.
ALTER TABLE	Modify the structure of an existing table.
DESCRIBE	Retrieve information about a table, such as its columns, data types, and other properties.
RESET	Reset configuration options to their default values.
SET	Configure Flink SQL execution parameters and options.
SHOW	Display information about the current Flink SQL environment.
USE CATALOG	Switch the current catalog in the Flink SQL session.
USE DATABASE	Switch the current database within the active catalog.

Flink SQL is a rather complete implementation of the SQL standard. If you know SQL well, you'll find that Flink SQL includes pretty much everything you might hope to find, and the features listed here are all available for both stream and batch processing

Data Definition Language (DDL) statements are imperative verbs that define metadata in Flink SQL by adding, changing, or deleting tables.

Unlike Data Manipulation Language (DML) statements, DDL statements modify only metadata and don't change data.

See the complete syntax reference for more information.

Flink SQL - DML Statements

Command	Description
SELECT	Specify columns you want to retrieve for data manipulation
INSERT	Insert data into a target table
ORDER BY	Sort the result rows according to the specified column(s)
GROUP BY	Clause for grouping together in order to aggregate
OVER	Compute an aggregated value for every row over a range of ordered rows
JOIN	Join rows from two tables (INNER, LEFT, RIGHT, FULL OUTER)
WITH	Define a Common Table Expression (CTE) within a query
WINDOWS	Define time windows based on type (TUMBLE, HOP, CUMULATE, SESSION), size and other properties

Data Manipulation Language (DML) statements are declarative verbs that read and modify data in Apache Flink® tables. Unlike Data Definition Language (DDL) statements, DML statements modify only data and don't change metadata.

There are more statements, like LIMIT, Top-N queries, deduplication queries, pattern recognition queries and more. See the complete syntax reference for more information.

Example using `OVER`:

```
SELECT order_id, order_time, amount,
  SUM(amount) OVER (
    PARTITION BY product
    ORDER BY order_time
    RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW
  ) AS one_hour_prod_amount_sum
FROM orders
```

Example. `WITH` defines a Common Table Expression (CTE) within a query. CTE can be thought as temporary view:

```
WITH orders_with_total AS (
    SELECT order_id, price + tax AS total
    FROM orders
)
SELECT order_id, SUM(total)
FROM orders_with_total
GROUP BY order_id;
```

Example of a Window Aggregation using the command TUMBLE:

```
SELECT window_start, window_end, SUM(points) as `sum`
  FROM TABLE(
    TUMBLE(TABLE gaming_player_activity_source, DESCRIPTOR($rowtime), INTERVAL '10'
MINUTES))
  GROUP BY window_start, window_end;
```


Flink SQL - Operators

There are more than 175 built-in functions:

- Aggregate Functions (AVG, COUNT, MAX, MIN, SUM, etc.)
- Collection Functions (ARRAY, ARRAY_CONTAINS, MAP, etc.)
- Comparison Functions (BETWEEN, EXISTS, IN, IS NULL, LIKE, etc.)
- Conditional Functions (CASE, IF, GREATEST, IS_ALPHA, etc.)
- Datetime Functions (CURRENT_DATE, DATE_FORMAT, INTERVAL, TO_TIMESTAMP, etc.)
- Hash Functions (SHA224, SHA256, SHA512, etc.)
- JSON Functions (IS JSON, JSON_ARRAY, JSON_EXISTS, JSON_QUERY, etc.)
- Numeric Functions (ABS, CEILING, LOG, PI, SIN, COS, RAND, etc.)
- String Functions (CONCAT, CHARACTER_LENGTH, PARSE_URL, TRIM, etc.)

See the complete syntax reference for more information.

Flink SQL - Data Types

Rich set of native data types:

- ARRAY
- BIGINT
- BINARY
- BOOLEAN
- BYTES
- CHAR
- DATE
- DECIMAL
- DOUBLE
- FLOAT
- INT
- MAP
- MULTISET
- NULL
- ROW
- SMALLINT
- TIME
- TIMESTAMP
- TIMESTAMP_LTZ
- TINYINT
- VARCHAR/STRING

A data type describes the logical type of a value in a SQL table. You use data types to declare the input and output types of an operation.

The Flink data types are similar to the SQL standard data type terminology, but for efficient handling of scalar expressions, they also contain information about the nullability of a value.

These are examples of SQL data types:

```
INT
INT NOT NULL
ROW<fieldOne ARRAY<BOOLEAN>, fieldTwo TIMESTAMP(3)>
```

See the complete set of data types for more information.

02c: Stream vs. Batch Processing

Description

Describing the differences between stream and batch processing.

Streaming vs. Batch

STREAMING	BATCH
Bounded or unbounded streams	**Only bounded streams**

A good way to understand what streaming is to contrast it with batch processing.

Flink does, in fact, support both stream and batch processing: You write your code once, and depending on the context in which it is run, it runs as a streaming job, or a batch job.

For Flink, batch is nothing more than a special case in the runtime. The effect it has is to enable a bunch of optimizations.

Those optimizations for batch mode are only possible with bounded streams. Unbounded streams require streaming processing.

Streaming vs. Batch

STREAMING	BATCH
Bounded or unbounded streams	Only bounded streams
Entire topology must always be running	**Execution proceeds in stages, running as needed**

Streaming:
We expect a stream processor to produce results quickly, with minimal end-to-end latency for the entire pipeline. For this to work, the entire pipeline must be left running continuously.

Batch:
By contrast, when Flink is running in batch mode, it can reduce its resource requirements by running the pipeline stages sequentially, rather than concurrently. If you think back to the earlier example where the first step was to filter out the orange events, in batch mode we could first remove all the orange events from the dataset, then push the filtered dataset through the next step in the pipeline, and so on.

Streaming vs. Batch

STREAMING	BATCH
Bounded or unbounded streams	Only bounded streams
Entire topology must always be running	Execution proceeds in stages, running as needed
Input must be processed as it arrives	**Input may be pre-sorted by time and key**

One of the available optimizations for batch processing is to pre-sort the input to make it easier to process. For example, databases sometimes implement joins by first sorting the input tables, after which the join is easier to do.

Flink can do the same thing, but only in batch mode.

When Flink is operating in streaming mode, it must accept each event as it arrives. Sometimes Flink is unable to immediately process an event, and may have to buffer it until other necessary data arrives. These buffers have to be stored in some sort of durable state store, so their contents aren't lost if the task manager fails and has to be restarted.

Streaming vs. Batch

STREAMING	BATCH
Bounded or unbounded streams	Only bounded streams
Entire topology must always be running	Execution proceeds in stages, running as needed
Input must be processed as it arrives	Input may be pre-sorted by time and key
Results are reported as they become ready	**Results are reported at the end of the job**

In streaming mode, no event can be assumed to be the last. Hence, rather than no output, results are produced incrementally, after every event, or periodically, based on timers.

In batch mode, processing can continue until the job is complete, at which point the final results can be produced.

Streaming vs. Batch

STREAMING	BATCH
Bounded or unbounded streams	Only bounded streams
Entire topology must always be running	Execution proceeds in stages, running as needed
Input must be processed as it arrives	Input may be pre-sorted by time and key
Results are reported as they become ready	Results are reported at the end of the job
Failure recovery resumes from a recent snapshot	**Failure recovery does a reset and full restart**

In streaming mode, Flink tries to minimize the downtime by resuming from a recent snapshot.

In batch mode, Flink can recover from failures by restarting the job from the beginning of the batch.

Set the Execution Mode

- Streaming mode is the default. To set streaming mode:

```
set 'execution.runtime-mode' = 'streaming';
```

- To set batch mode:

```
set 'execution.runtime-mode' = 'batch';
```

Execution Mode in Confluent Cloud

In Confluent Cloud, all queries are executed only in **Streaming execution mode**

Streaming mode supports both **bounded** and **unbounded** streams

---

Stream processing is more powerful than batch processing, and any use case that can be satisfied with batch processing could be handled by a stream processor instead. However, batch processing could be more efficient for certain queries.

On the other hand, some use cases are only possible with a stream processor. For example, any use case where low-latency is critical requirement, such as data center monitoring, or fraud detection. For use cases where you need an immediate response to every event, the latency from batch processing makes it a poor fit.

Example - Processing Bounded Data in Streaming & Batch Mode

Query:

```
select count(*) AS `count` from bounded_pageviews;
```

Result:

STREAMING	BATCH
op count ---------- ... ... -U 497 +U 498 -U 498 +U 499 -U 499 +U 500	count ----- 500

When operating in streaming mode, the Flink runtime can't rely on the stream to ever end, so it is instead continuously updating the result as it processes the input stream. It ultimately arrives at the same result as when running in batch mode, but the sink for this streaming counting job is seeing all of the incremental work done along the way by the SQL runtime.

02d: Flink SQL on Confluent Cloud

Description

Describing how Flink SQL looks like and is implemented in Confluent Cloud.

Core Principals

Confluent Cloud provides a truly cloud-native experience for Flink

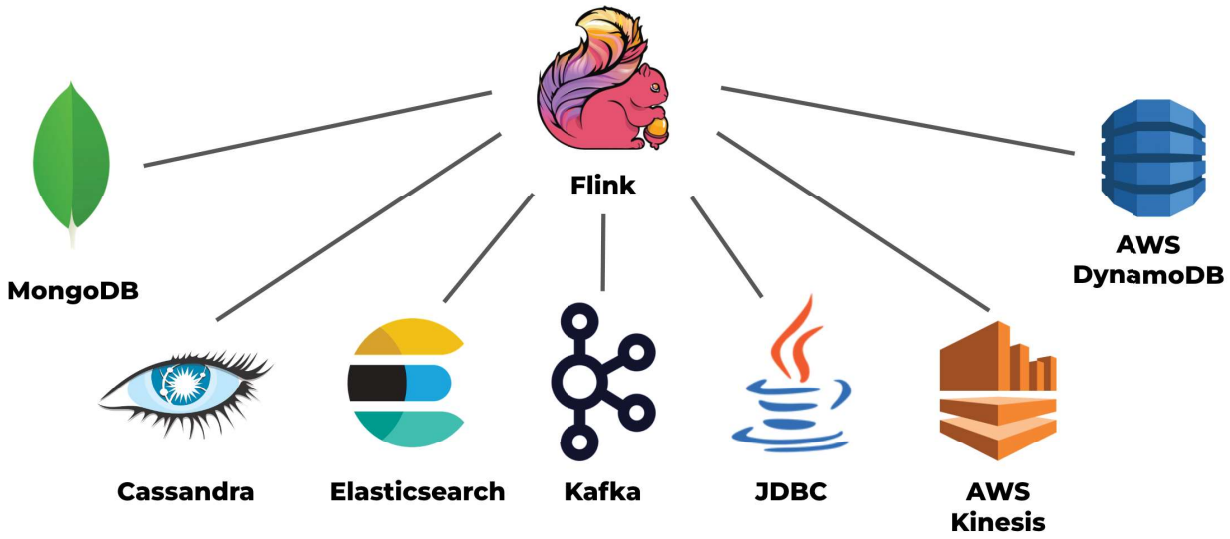
- Serverless
- Simple
- Independently Scalable from Kafka
- Secure

Confluent Cloud for Apache Flink®□ provides a truly cloud-native experience for Flink. This means you can fully focus on your business logic, encapsulated in Flink SQL statements, and Confluent Cloud takes care of what's needed to run them in a secure, resource-efficient and fault-tolerant manner. You don't need to know about or interact with Flink clusters, state backends, checkpointing, and all of the other aspects that are usually involved when operating a production-ready Flink deployment.

- **Serverless** can have many connotations. To Confluent, it has three primary dimensions:
  - Elastic autoscaling with scale-to-zero:
    On Confluent Cloud, Flink workloads scale automatically without the need for user intervention.
  - Evergreen runtime and APIs:
    The Flink runtime must always be up-to-date, providing you with the latest functionality. The APIs we expose should also be declarative.
  - Usage-based billing:
    You pay only for what you use, not what you provision.
- **Simple**:
  Managing apps on Apache Flink can also be challenging. Each application has to be independently sized and subsequently managed on an ongoing basis. Developers need to remain actively involved as the workload fluctuates, determining application scale and the required parallelism.
  On Confluent Cloud, you don't size apps when using Flink. Confluent takes care of managing resource assignments, parallelism, scale, high availability, etc. Our goal is for developers to focus on app development, not complex infrastructure-related tasks.
- **Scalable**:
  The scalability of a cloud-native service is closely tied to the separation of compute and storage. By separating these two components, you can scale each one independently, allowing for more efficient use of resources and better overall scalability.

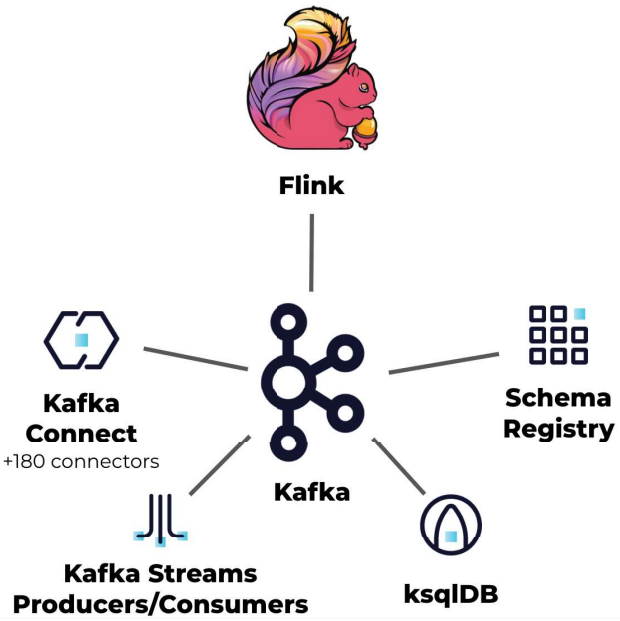
- **Security**:
  Open source Apache Flink does not have a security model built into its framework. This is a critical dimension that organizations have to address when deploying Flink. By contrast, Confluent Cloud offers a robust and secure suite of capabilities to control, manage, and govern access to data.

Flink's Datastream Connectors

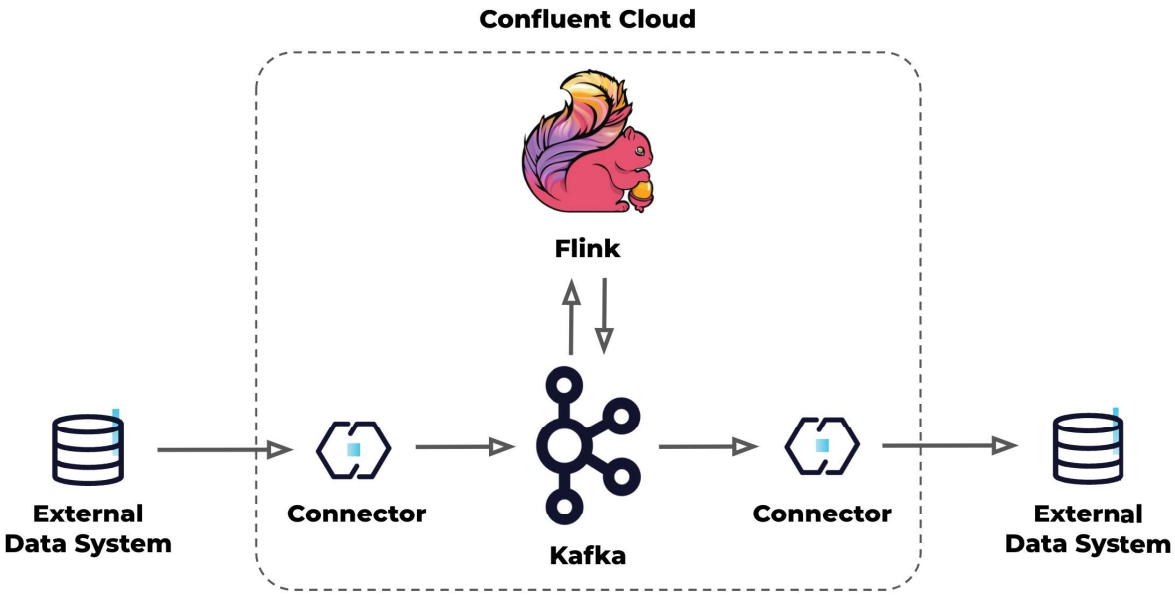


Flink on Confluent Cloud

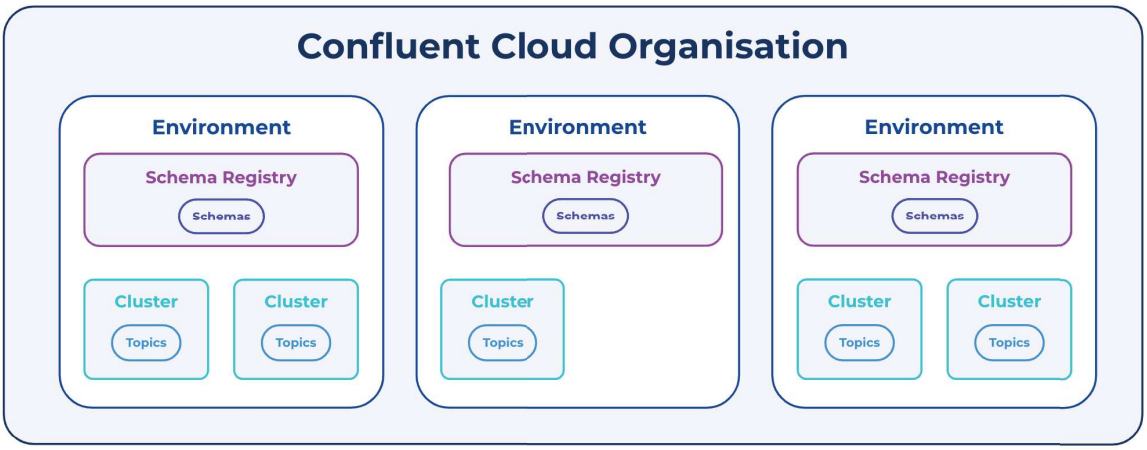
Flink on Confluent Cloud



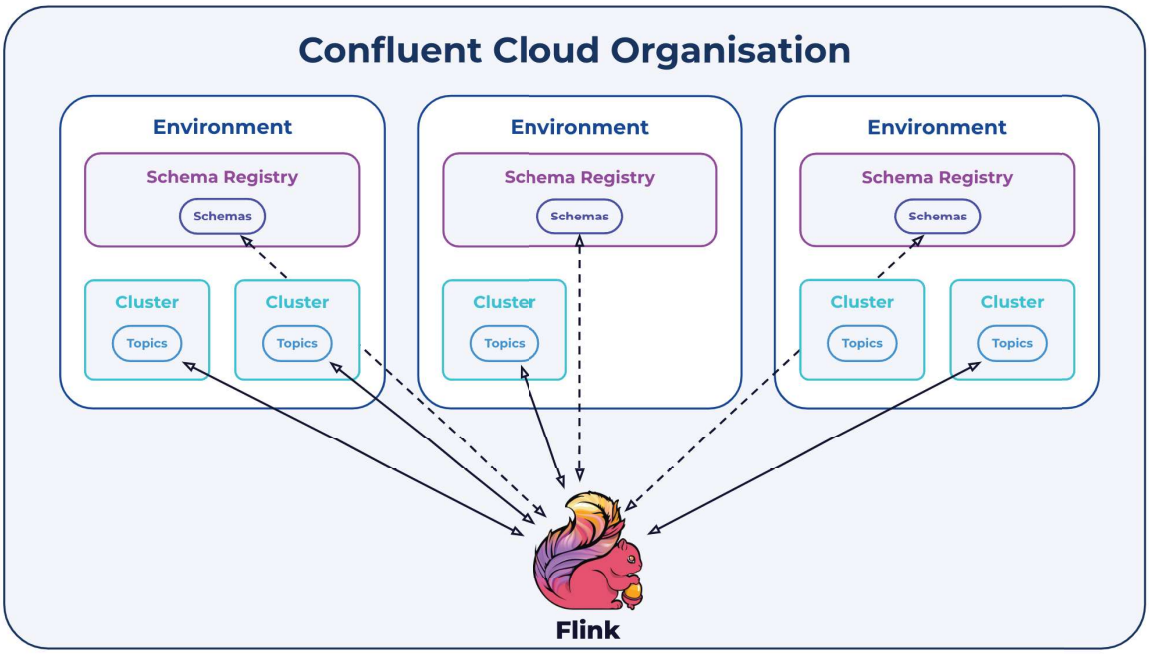
ETL with Confluent Cloud



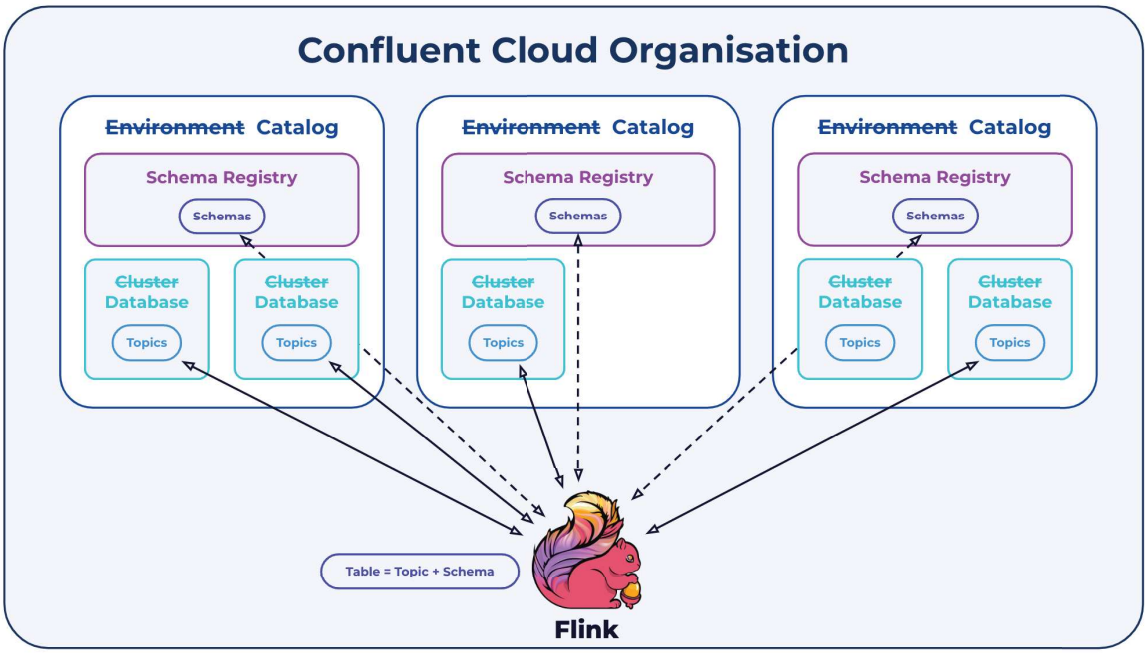
Confluent Cloud Structure



Confluent Cloud Structure & Flink



Confluent Cloud Structure & Flink (Flink Syntax)



Apache Kafka® topics and schemas are always in sync with Flink, simplifying how you can process your data. Any topic created in Kafka is visible directly as a table in Flink, and any table created in Flink is visible as a topic in Kafka. Effectively, Flink provides a SQL interface on top of Confluent Cloud.

Because Flink follows the SQL standard, the terminology is slightly different from Kafka. The following table shows the mapping between Kafka and Flink terminology.

As a result, when you start using Flink, you can directly access all of the environments, clusters, and topics that you already have in Confluent Cloud, without any additional metadata creation.

Compared with Open Source Flink, the main difference is that the DDLs related to catalogs, databases, and tables act on physical objects and not only on metadata. For example, when you create a table in Flink, the corresponding topic and schema are created immediately in Confluent Cloud.

Kafka	Flink	Notes
Environment	Catalog	Flink can query and join data that are in any environments/catalogs
Cluster	Database	Flink can query and join data that are in different clusters/databases

Kafka	Flink	Notes
Topic + Schema	Table	Kafka topics and Flink tables are always in sync. You never need to declare tables manually for existing topics. Creating a table in Flink creates a topic and the associated schema.

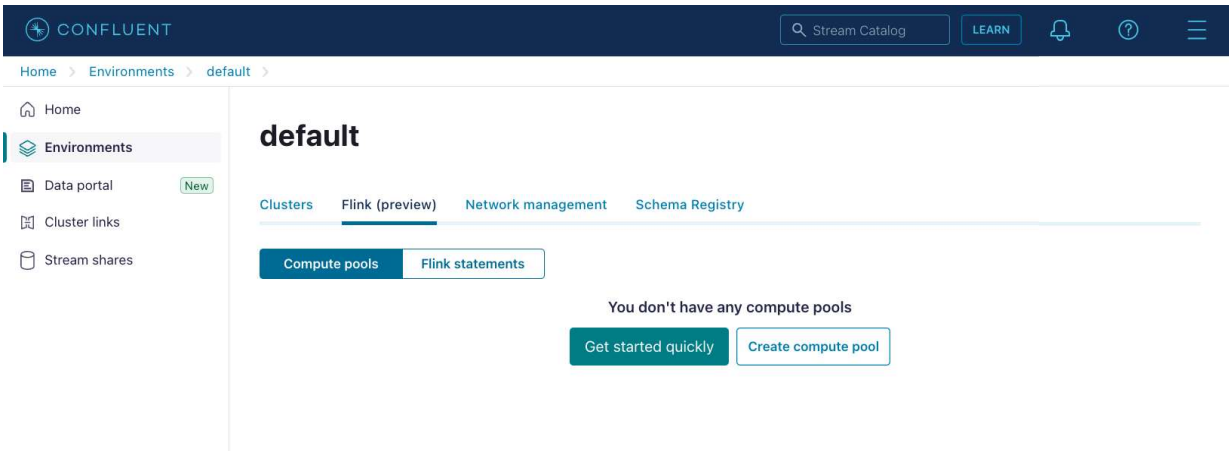
Compute Pools

Set of compute resources to run your Flink SQL statements:

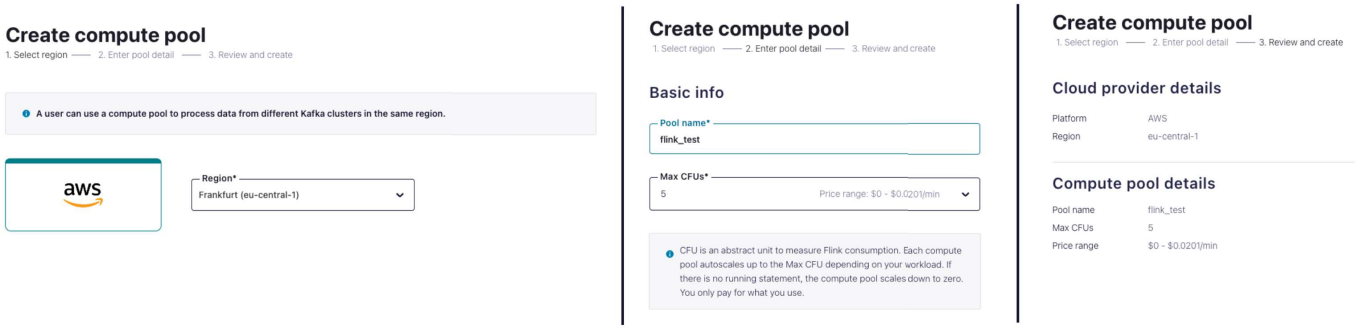
- Provisioned in a specific region (only read and write Kafka topics in the same region)
- Measured in CFUs (5, 10, 20, 30, 40, 50)
- Expand and shrink automatically (from zero to the maximum CFUs configured)
- Workload isolation between compute pools (statements in different compute pools are isolated from each other)

- **Compute pools and isolation**
  All statements using the same compute pool compete for resources. Although Confluent Cloud's Autopilot aims to provide each statement with the resources it needs, this might not always be possible, in particular, when the maximum resources of the compute pool are exhausted.
  To avoid situations in which statements with different latency and availability requirements compete for resources, Confluent recommends using separate compute pools for different use cases, for example, ad-hoc exploration vs. mission-critical, long-running queries. Because statements may affect each other, Confluent recommends sharing compute pools only between statements with comparable requirements.

Compute Pools Creation - Confluent Cloud Console

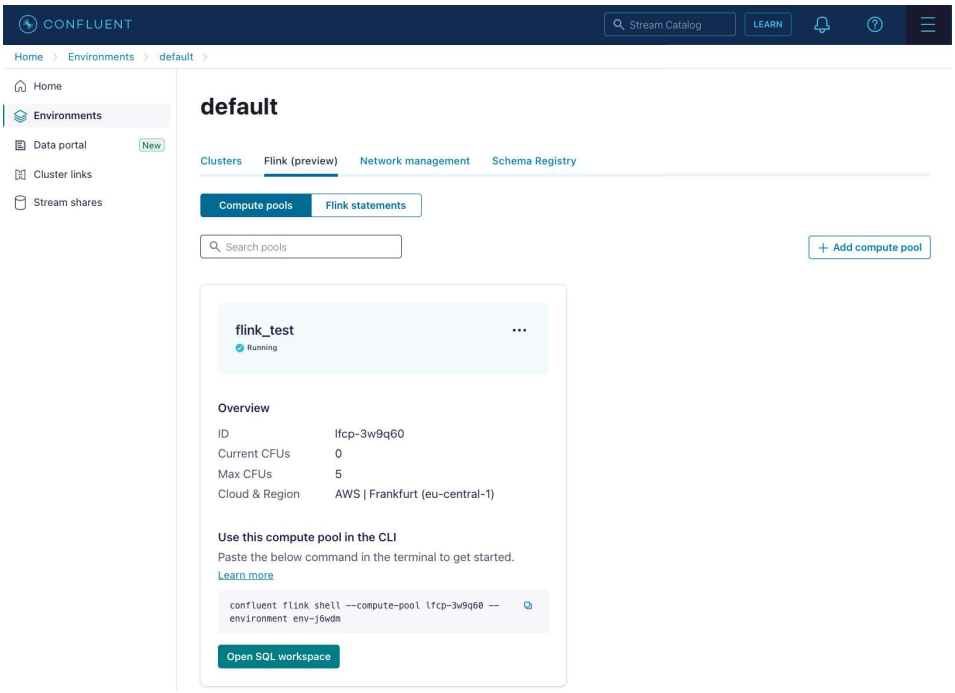


Compute Pools Configuration - Confluent Cloud Console

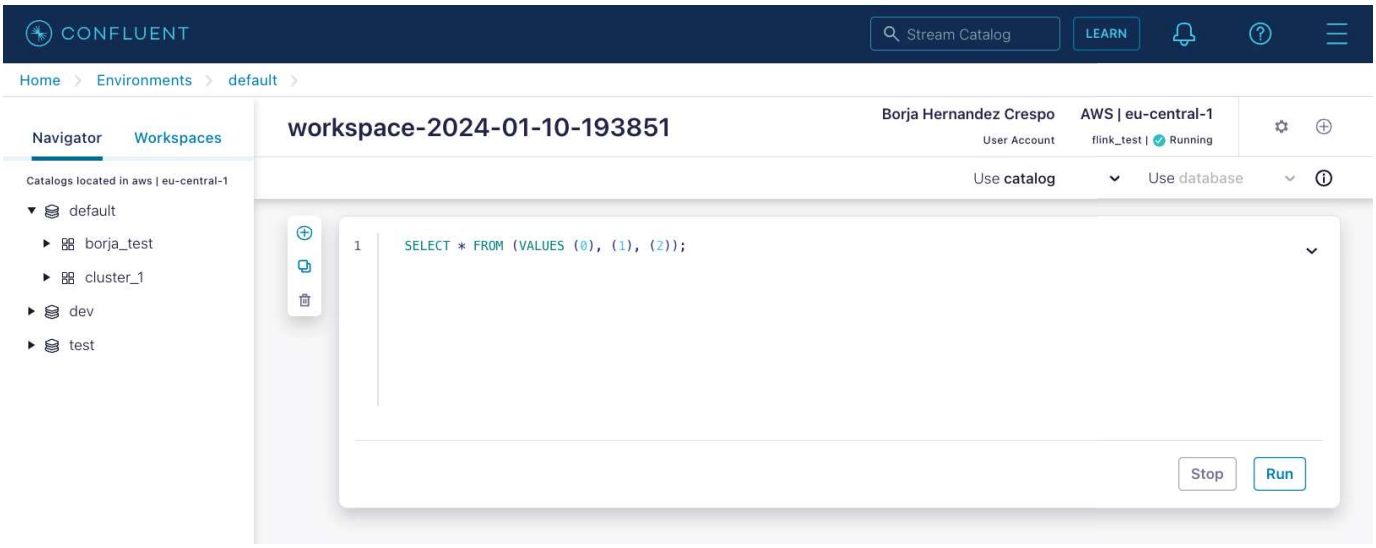


Compute Pools Running - Confluent Cloud Console

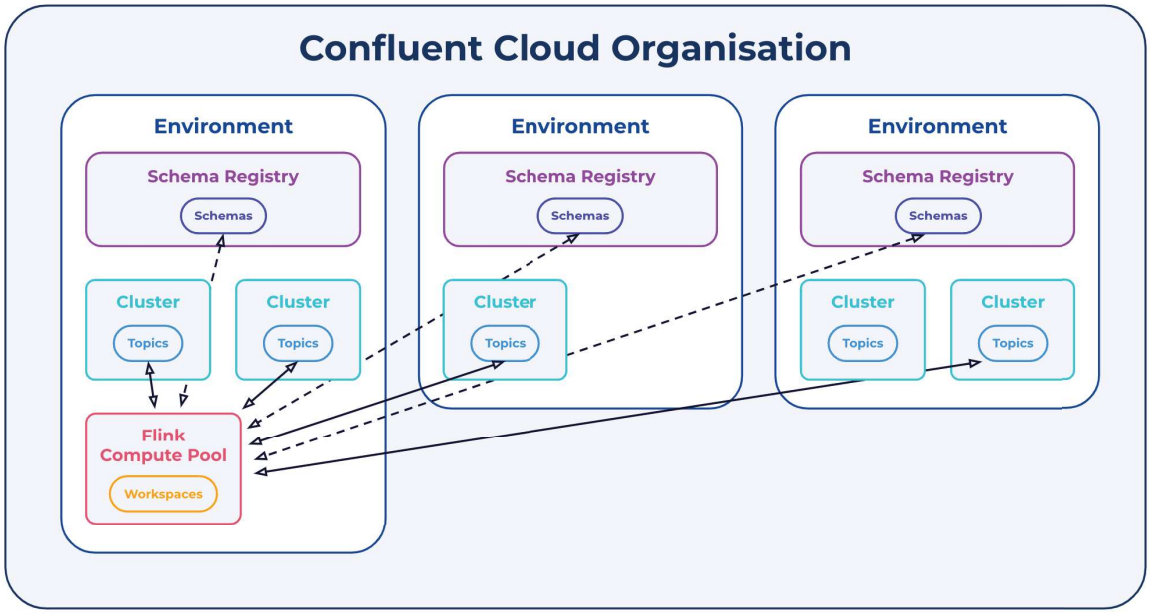
Now, you have a cloud-native production-ready Flink cluster



SQL Workspace - Confluent Cloud Console



Confluent Cloud Structure & Compute Pools



Compute pools have no inherent access to anything. Credentials are given only to the statements that run on them, with credentials of either the current user or a specified service account and can only access what Kafka data those credentials have access to. As long as the Flink Compute Pool and the accessed Kafka clusters are in the same cloud provider and region.

Flink SQL in Confluent Cloud uses TopicName strategy for the automatic creation of schemas based on the table structure in Schema Registry.

Running Flink SQL using Confluent CLI

To create a Flink compute pool:

```
confluent flink compute-pool create my-compute-pool --cloud aws --region us-west-2
--max-cfu 5
```

To start the Flink SQL shell:

```
confluent flink shell --compute-pool lfcp-xxxxxx --environment env-xxxxx

Warning: no service account provided. To ensure that your statements run continuously,
switch to using a service account instead of your user identity with `confluent iam
service-account use` or `--service-account`. Otherwise, statements will stop running
after 4 hours.
Welcome!
To exit, press Ctrl-Q or type "exit".

[Ctrl-Q] Quit [Ctrl-S] Toggle Smart Completion
>
```

The prefix of the compute pool ID `lfcp-xxxxxx` stands for "logical flink compute pool".

Lab Module 02: Working with Flink in Confluent Cloud

Please work on **Lab Module 02: Working with Flink in Confluent Cloud**.

Refer to the Exercise Guide.