Lab 4: Declarative Queries and Schema Evolution

**Goal:** Introduce the high-level Table API for declarative stream processing and demonstrate its resilience to schema evolution in real-time data sources.

# 

# INDEX

[**Purpose of this Lab**](#_nan8bid3y4rn) **3**

[**Prerequisites**](#_yn07etxa6vi4) **3**

[**Project Structure**](#_naid6ffvjiq5) **3**

[**Part 1: Project and Environment Setup**](#_lrn87yvz322) **4**

[**Part 2: Setting Up the Kafka Cluster**](#_r11pdn2umqad) **6**

[**Part 3: Developing the Flink Table API Application**](#_4y81gztlm2ca) **7**

[**Part 4: Executing the End-to-End Pipeline**](#_unok30eubgm2) **10**

[**Part 5: Verification**](#_5b2fog9fgtvc) **13**

[**Part 6: Cleanup**](#_3emsxnmm11dx) **14**

[**Part 7: Next Steps**](#_ywqmv4sq65eh) **15**

# 

# Purpose of this Lab

This lab transitions from the imperative DataStream API to the declarative, higher-level Table API. Instead of defining *how* to process data step-by-step, you will define *what* result you want using SQL-like expressions.

You will re-implement the aggregation from Lab 3 (counting events per product) using the concise Table API. The main experiment will involve modifying the Kafka producer to add a new field (price) to the JSON events while the Flink job is running. You'll observe how the job handles this change seamlessly, showcasing its robustness for production environments. By completing this lab, you will:

* **Use the StreamTableEnvironment:** Set up and use the unified entry point for Table and SQL API programming.
* **Convert a DataStream to a Table:** Turn a data stream into a dynamic table to perform declarative queries.
* **Define Declarative Queries:** Build a data processing pipeline using method chaining and Table API expressions.
* **Convert a Table back to a DataStream:** Transform the results of a table query back into a data stream.
* **Handle Schema Evolution:** Observe how Flink's Table API can gracefully handle new fields added to a source schema without a job restart.

# Prerequisites

This lab assumes you have successfully completed Labs 1, 2, and 3 and are using an **Ubuntu** environment. Your Flink cluster should already have the Kafka connector JAR in its lib directory from the previous lab.

# Project Structure

By the end of this lab, your new project directory will be structured as follows:

|  |
| --- |
| ~/flink-lab-4/ ├── venv/ # The isolated Python virtual environment ├── docker-compose.yaml # Defines our Kafka service ├── producer.py # The script to generate mock data with schema evolution └── table\_api\_queries.py # The Flink Table API job script |

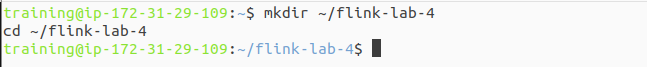
# Part 1: Project and Environment Setup

**Step 1: Create Project Directory and Virtual Environment**

We'll create a new, separate directory for this lab.

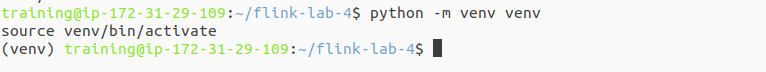
Create and navigate to the new lab directory

|  |
| --- |
| mkdir ~/flink-lab-4 cd ~/flink-lab-4 |



Initialize and activate a Python virtual environment

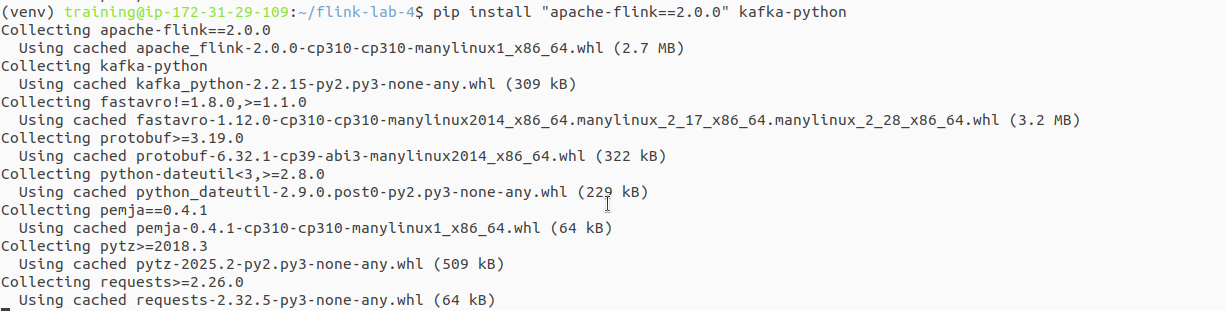
|  |
| --- |
| python -m venv venv source venv/bin/activate |



**Step 2: Install Python Dependencies**

With the venv active, install apache-flink and the Python client for Kafka.

|  |
| --- |
| pip install "apache-flink==2.0.0" kafka-python |



**Step 3: Configure Flink for the New Project**

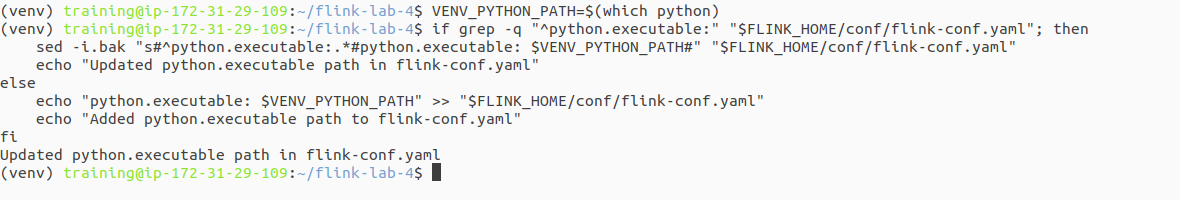
Since we created a new virtual environment, we must update Flink's configuration to point to the correct Python executable for this lab.

Get the absolute path to the Python executable in the new venv

|  |
| --- |
| VENV\_PYTHON\_PATH=$(which python) |

This command finds and replaces the 'python.executable' line, or adds it if not present.

|  |
| --- |
| if grep -q "^python.executable:" "$FLINK\_HOME/conf/flink-conf.yaml"; then  sed -i.bak "s#^python.executable:.\*#python.executable: $VENV\_PYTHON\_PATH#" "$FLINK\_HOME/conf/flink-conf.yaml"  echo "Updated python.executable path in flink-conf.yaml" else  echo "python.executable: $VENV\_PYTHON\_PATH" >> "$FLINK\_HOME/conf/flink-conf.yaml"  echo "Added python.executable path to flink-conf.yaml" fi |



# Part 2: Setting Up the Kafka Cluster

**Step 1: Define the Kafka Service**

Create a file named docker-compose.yaml in the ~/flink-lab-4 directory. This is identical to the file from the previous labs.

|  |
| --- |
| code docker-compose.yaml |

Add the following content to the file:

|  |
| --- |
| # docker-compose.yaml services:  zookeeper:  image: confluentinc/cp-zookeeper:7.3.2  container\_name: zookeeper  ports: ["2181:2181"]  environment:  ZOOKEEPER\_CLIENT\_PORT: 2181  ZOOKEEPER\_TICK\_TIME: 2000  kafka:  image: confluentinc/cp-kafka:7.3.2  container\_name: kafka  ports: ["9092:9092"]  depends\_on: [zookeeper]  environment:  KAFKA\_BROKER\_ID: 1  KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181  KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092  KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT  KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT  KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1 |

**Step 2: Launch the Kafka Cluster**

From the ~/flink-lab-4 directory, start the services.

|  |
| --- |
| docker compose up -d |

# Part 3: Developing the Flink Table API Application

**Step 1: Implement the Kafka Producer**

Create a file named producer.py. This version includes a toggle for schema evolution and our robust connection retry logic.

|  |
| --- |
| code producer.py |

Add the following code:

|  |
| --- |
| # producer.py import json import time import random from kafka import KafkaProducer from kafka.errors import NoBrokersAvailable  KAFKA\_TOPIC = 'clicks' KAFKA\_BROKERS = 'localhost:9092' SCHEMA\_EVOLUTION\_MODE = False # Set to True to add the 'price' field  def create\_producer():  """Creates a KafkaProducer with retry logic."""  retries = 10  while retries > 0:  try:  producer = KafkaProducer(  bootstrap\_servers=KAFKA\_BROKERS,  value\_serializer=lambda v: json.dumps(v).encode('utf-8')  )  print("Successfully connected to Kafka.")  return producer  except NoBrokersAvailable:  retries -= 1  print(f"Kafka not available, retrying in 5 seconds... ({retries} retries left)")  time.sleep(5)  raise RuntimeError("Failed to connect to Kafka after multiple retries.")  if \_\_name\_\_ == '\_\_main\_\_':  producer = create\_producer()    print("Producing mock click events... Press Ctrl+C to terminate.")  print(f"Schema Evolution Mode: {'ON' if SCHEMA\_EVOLUTION\_MODE else 'OFF'}")   user\_ids = [f'user\_{i}' for i in range(1, 11)]  product\_ids = [f'prod\_{i}' for i in range(1, 6)]  event\_types = ['page\_view', 'page\_view', 'page\_view', 'add\_to\_cart']    try:  while True:  event = {  'event\_type': random.choice(event\_types),  'user\_id': random.choice(user\_ids),  'product\_id': random.choice(product\_ids),  'timestamp': int(time.time() \* 1000)  }    # --- The Schema Evolution part ---  if SCHEMA\_EVOLUTION\_MODE:  event['price'] = round(random.uniform(5.0, 100.0), 2)  # -----------------------------------   producer.send(KAFKA\_TOPIC, value=event)  print(f"Sent event: {event}")  time.sleep(1)  except KeyboardInterrupt:  print("\nStopping producer.")  finally:  producer.flush()  producer.close() |

**Step 2: Implement the Flink Table API Script**

Create the main Flink application file, table\_api\_queries.py.

|  |
| --- |
| code table\_api\_queries.py |

Add the following code:

|  |
| --- |
| # table\_api\_queries.py from pyflink.datastream import StreamExecutionEnvironment from pyflink.table import StreamTableEnvironment from pyflink.table.expressions import lit from pyflink.datastream.connectors.kafka import KafkaSource, KafkaOffsetsInitializer from pyflink.datastream.formats.json import JsonRowDeserializationSchema from pyflink.common.watermark\_strategy import WatermarkStrategy from pyflink.common import Types  def main():  # 1. Set up the execution environments  env = StreamExecutionEnvironment.get\_execution\_environment()  table\_env = StreamTableEnvironment.create(stream\_execution\_environment=env)   # 2. Define the source schema (without the 'price' field)  type\_info = Types.ROW\_NAMED(  ["event\_type", "user\_id", "product\_id", "timestamp"],  [Types.STRING(), Types.STRING(), Types.STRING(), Types.LONG()]  )   json\_deserializer = JsonRowDeserializationSchema.builder() \  .type\_info(type\_info).build()   # 3. Create the Kafka source  kafka\_source = KafkaSource.builder() \  .set\_bootstrap\_servers('localhost:9092') \  .set\_topics('clicks') \  .set\_group\_id('flink-table-api-group') \  .set\_starting\_offsets(KafkaOffsetsInitializer.latest()) \  .set\_value\_only\_deserializer(json\_deserializer) \  .build()   data\_stream = env.from\_source(  source=kafka\_source,  watermark\_strategy=WatermarkStrategy.no\_watermarks(),  source\_name="kafka\_source"  )   # 4. Convert the DataStream to a Table  source\_table = table\_env.from\_data\_stream(data\_stream)   # 5. Define and execute the Table API query  aggregation\_result = source\_table \  .filter(source\_table.event\_type == 'add\_to\_cart') \  .group\_by(source\_table.product\_id) \  .select(source\_table.product\_id, lit(1).count.alias('add\_to\_cart\_count'))   # 6. Convert the result Table back to a DataStream to print it  result\_stream = table\_env.to\_changelog\_stream(aggregation\_result)  result\_stream.print()   # 7. Execute the Flink job  env.execute("table\_api\_schema\_evolution")  if \_\_name\_\_ == '\_\_main\_\_':  main() |

# Part 4: Executing the End-to-End Pipeline

This part is interactive and requires three separate terminal windows.

**Terminal 1: Start the Flink Cluster**

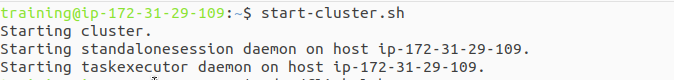
If your Flink cluster is not already running, restart it.

If running, stop it first

|  |
| --- |
| stop-cluster.sh |

Start the cluster

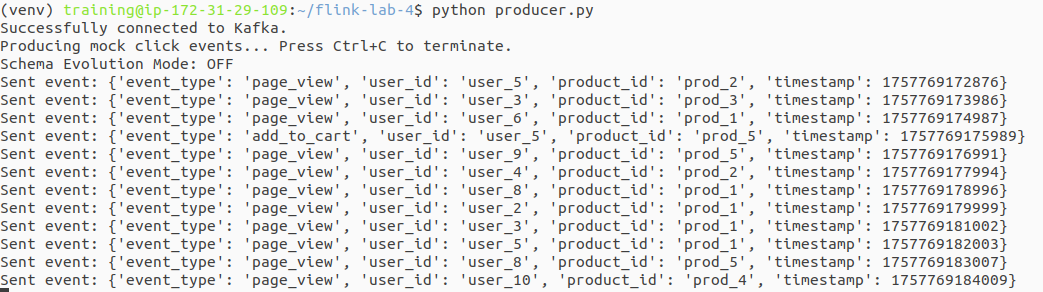
|  |
| --- |
| start-cluster.sh |



**Terminal 2: Launch the Data Producer (Phase 1)**

Navigate to your lab directory, activate the environment, and start the producer script.

|  |
| --- |
| cd ~/flink-lab-4 source venv/bin/activate python producer.py |



The producer will start, printing Schema Evolution Mode: OFF.

**Terminal 3: Submit the Flink Application**

Submit the job with our simple command.

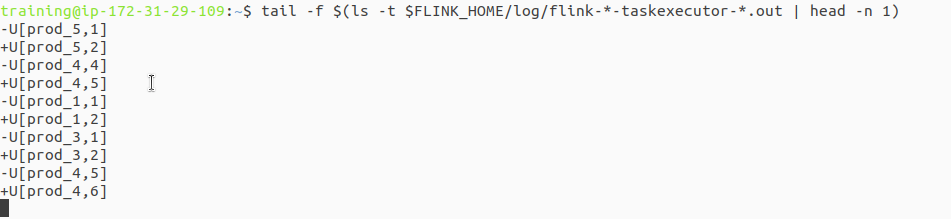
|  |
| --- |
| cd ~/flink-lab-4 source venv/bin/activate flink run -py table\_api\_queries.py |



**Phase 2: Observe Initial Output**

With the job running, let it process events for about a minute. Check the TaskManager logs to see the output.

|  |
| --- |
| tail -f $(ls -t $FLINK\_HOME/log/flink-\*-taskexecutor-\*.out | head -n 1) |



You will see a changelog stream representing the continuous query result. +I means a new insert, -U is an update of the old value, and +U is an update of the new value.

|  |
| --- |
| (+I,prod\_2,1) (-U,prod\_2,1) (+U,prod\_2,2) (+I,prod\_5,1) |

This is the "before" state of our experiment.

**Phase 3: Introduce Schema Change**

Now we will modify the data source *without stopping the Flink job*.

**Stop the producer** in Terminal 2 by pressing Ctrl+C.

**Do not stop the Flink job in Terminal 3.**

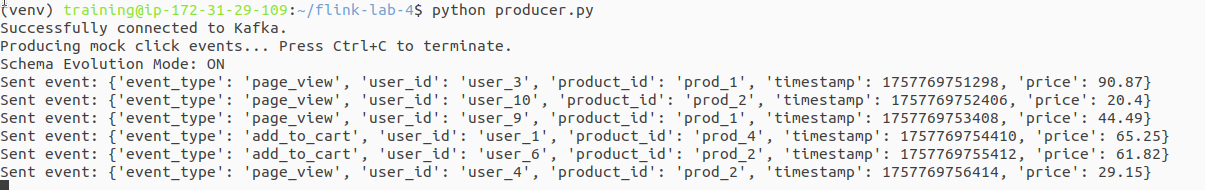
Open producer.py in your editor.

Change the SCHEMA\_EVOLUTION\_MODE flag to True.

|  |
| --- |
| SCHEMA\_EVOLUTION\_MODE = True # Set to True to add the 'price' field |

Save the file and restart the producer in Terminal 2:

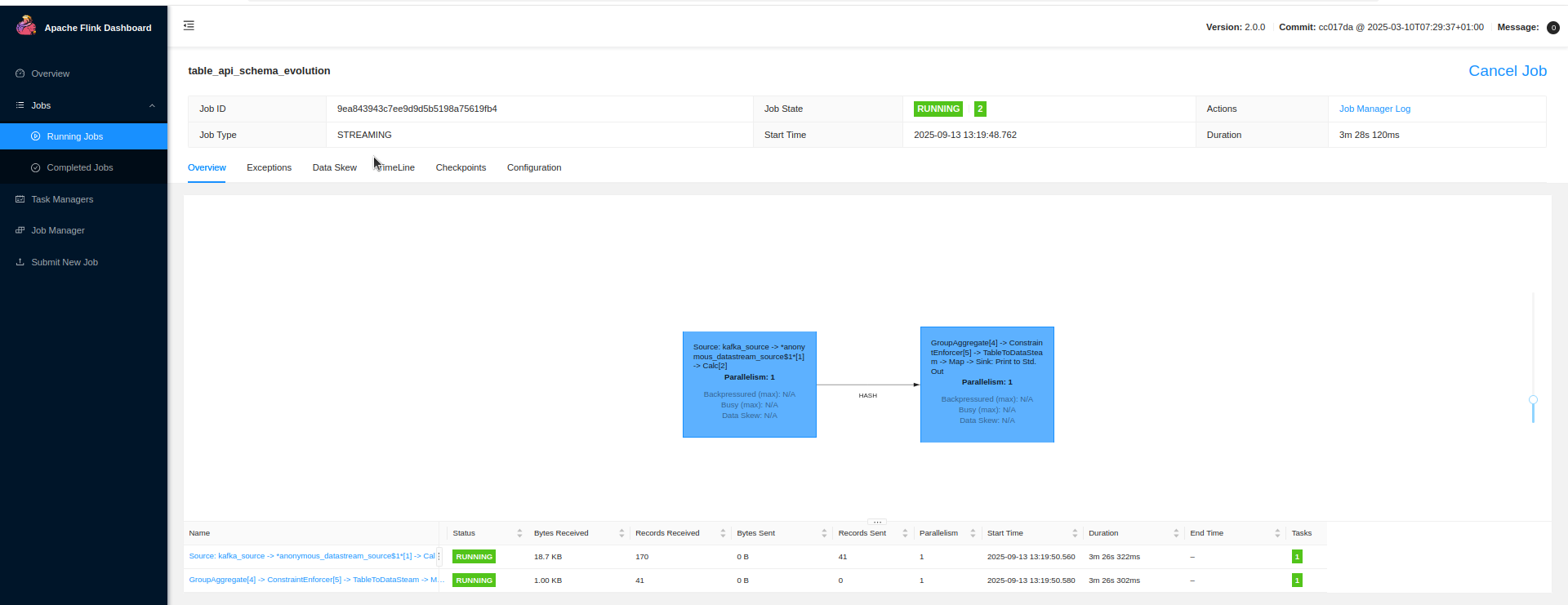
|  |
| --- |
| python producer.py |



The producer will now print Schema Evolution Mode: ON and send events that include the new price field.

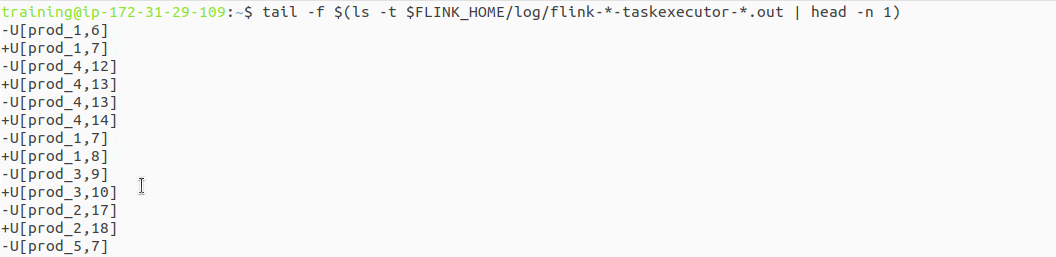
# Part 5: Verification

**Step 1: Verify Job Execution** Observe the Flink job in the UI and the logs. The job will continue running without any errors or interruptions.



**Step 2: Inspect the Output** View the TaskManager logs again. The output format will remain exactly the same. Flink's Table API, based on the schema defined in the job, has simply ignored the new price field it did not know about. This demonstrates its resilience to non-breaking schema changes.

|  |
| --- |
| tail -f $(ls -t $FLINK\_HOME/log/flink-\*-taskexecutor-\*.out | head -n 1) |



# Part 6: Cleanup

Once you have verified the behavior, shut down all the components.

1. **Stop the Flink job:** Press Ctrl+C in Terminal 3.
2. **Stop the producer:** Press Ctrl+C in Terminal 2.
3. **Stop the Flink cluster:** stop-cluster.sh
4. **Stop the Kafka cluster:** cd ~/flink-lab-4 && docker compose down

# Part 7: Next Steps

* **SQL Queries:** Try rewriting the Table API query using pure SQL with table\_env.execute\_sql().
* **Schema Evolution with New Fields:** Modify the Flink job to recognize and process the price field.
* **Joins:** Introduce a second data stream (e.g., product metadata) and join it with the clickstream.