Lab 5: Real-Time, Fault-Tolerant SQL Analytics

**Goal:** Use the Flink SQL Client for declarative stream processing and demonstrate Flink’s fault tolerance by recovering a stateful aggregation job from a checkpoint.

# 

# INDEX

[**INDEX**](#_tn1ddx1m2hf) **2**

[**Purpose of this Lab**](#_b5mgg8whp214) **3**

[**Prerequisites**](#_6tarwini418u) **3**

[**Project Structure**](#_m88tkg5a9xk1) **3**

[**Part 1: Project and Environment Setup**](#_8vgt9tn63r8x) **4**

[**Part 2: Setting Up the Kafka Cluster**](#_g111eritt2rn) **6**

[**Part 3: Developing the Fault-Tolerant Flink SQL Application**](#_mdi8kd1a58vy) **7**

[**Part 4: Executing and Simulating Failure**](#_io1nl2dkoruk) **11**

[**Part 5: Verification**](#_z9d9reyhbviz) **14**

[**Part 6: Cleanup**](#_7hk6grwocsh7) **14**

[**Part 7: Next Steps**](#_nj5qv32u9j7m) **15**

# 

# Purpose of this Lab

This lab moves from the programmatic Table API to purely declarative Flink SQL. You will define your entire pipeline—source, sink, and transformation—using SQL Data Definition Language (DDL) and Data Manipulation Language (DML), an approach highly accessible to anyone familiar with SQL.

You will write a Flink SQL query to calculate the number of user logins per minute. The most critical part of this lab is demonstrating Flink's fault tolerance. You will enable checkpointing, run the job, simulate a failure by canceling it, and then observe how Flink automatically restores its state from the last checkpoint, ensuring exactly-once processing without data loss. By completing this lab, you will:

* **Enable Checkpointing:** Configure a Flink job for stateful fault tolerance.
* **Define Connectors with SQL DDL:** Create source and sink tables on external systems like Kafka using CREATE TABLE statements.
* **Write Streaming SQL Queries:** Use SQL with time-windowed aggregations (TUMBLE) to perform analytics on unbounded data.
* **Execute a SQL-only Pipeline:** Run a Flink job composed entirely of SQL statements.
* **Demonstrate Recovery:** Simulate a job failure and verify that the application recovers its state and resumes correctly.

# Prerequisites

This lab assumes you have successfully completed Labs 1 through 4 and are using an **Ubuntu** environment. Your Flink cluster should already have the Kafka connector JAR in its lib directory.

# 

# Project Structure

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

|  |
| --- |
| ~/flink-lab-5/ ├── venv/ # The isolated Python virtual environment ├── docker-compose.yaml # Defines our Kafka service ├── producer.py # The script to generate mock login data └── fault\_tolerant\_sql.py # The Flink SQL 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.

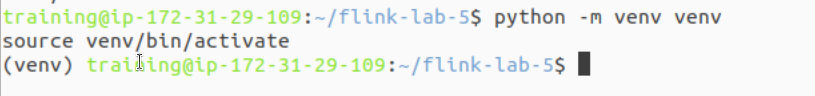
Create and navigate to the new lab directory

|  |
| --- |
| mkdir ~/flink-lab-5 cd ~/flink-lab-5 |



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**

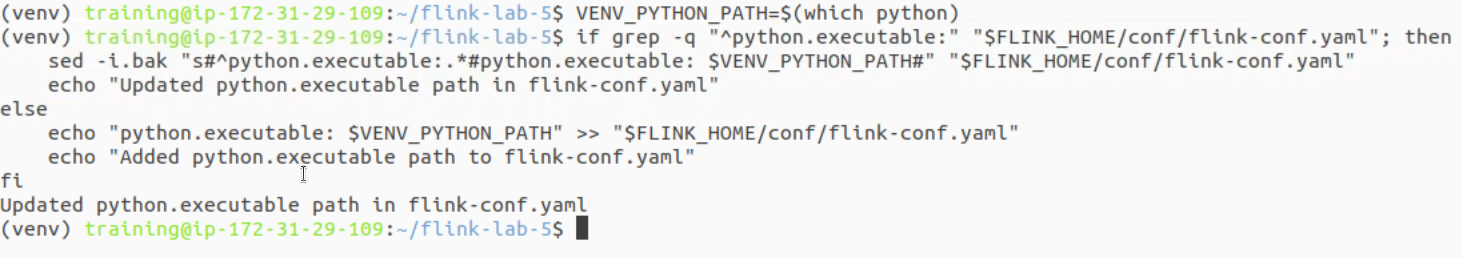
Since we created a new virtual environment, we must update Flink's configuration to point to the correct Python executable for this lab.

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-5 directory.

|  |
| --- |
| 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-5 directory, start the services.

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

# Part 3: Developing the Fault-Tolerant Flink SQL Application

**Step 1: Implement the Kafka Producer**

Create a file named producer.py. This version will create the Kafka topic automatically and includes 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.admin import KafkaAdminClient, NewTopic from kafka.errors import TopicAlreadyExistsError, NoBrokersAvailable  KAFKA\_TOPIC = 'logins' KAFKA\_BROKERS = 'localhost:9092'  def create\_producer\_and\_topic():  """Creates Kafka topic and returns a producer with retry logic."""  retries = 10  while retries > 0:  try:  # First, try to create the topic  admin\_client = KafkaAdminClient(bootstrap\_servers=KAFKA\_BROKERS)  try:  topic\_list = [NewTopic(name=KAFKA\_TOPIC, num\_partitions=1, replication\_factor=1)]  admin\_client.create\_topics(new\_topics=topic\_list, validate\_only=False)  print(f"Topic '{KAFKA\_TOPIC}' created successfully.")  except TopicAlreadyExistsError:  print(f"Topic '{KAFKA\_TOPIC}' already exists.")  finally:  admin\_client.close()   # Now, create the producer  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)  except Exception as e:  print(f"An unexpected error occurred: {e}")  retries -= 1  time.sleep(5)   raise RuntimeError("Failed to connect to Kafka and create topic after multiple retries.")  if \_\_name\_\_ == '\_\_main\_\_':  producer = create\_producer\_and\_topic()    print("Producing mock login events... Press Ctrl+C to terminate.")  user\_ids = [f'user\_{i}' for i in range(1, 21)]    try:  while True:  event = {  'user\_id': random.choice(user\_ids),  'login\_time': int(time.time() \* 1000)  }  producer.send(KAFKA\_TOPIC, value=event)  print(f"Sent event: {event}")  time.sleep(random.uniform(0.5, 2.0)) # Send events at a variable rate  except KeyboardInterrupt:  print("\nStopping producer.")  finally:  producer.flush()  producer.close() |

**Step 2: Implement the Flink SQL Script**

Create the main Flink application file, fault\_tolerant\_sql.py.

|  |
| --- |
| code fault\_tolerant\_sql.py |

Add the following code:

|  |
| --- |
| # fault\_tolerant\_sql.py from pyflink.datastream import StreamExecutionEnvironment from pyflink.table import StreamTableEnvironment  def main():  # 1. Set up the execution environments  env = StreamExecutionEnvironment.get\_execution\_environment()  table\_env = StreamTableEnvironment.create(stream\_execution\_environment=env)   # 2. IMPORTANT: Enable checkpointing for fault tolerance  # Checkpoint every 5 seconds (5000 ms)  env.enable\_checkpointing(5000)   # 3. Create a source table from the 'logins' Kafka topic  table\_env.execute\_sql("""  CREATE TABLE logins (  user\_id STRING,  login\_time BIGINT,  -- Define the login\_time as the event-time attribute  ts AS TO\_TIMESTAMP\_LTZ(login\_time, 3),  -- Define a watermark strategy for event-time processing  WATERMARK FOR ts AS ts - INTERVAL '5' SECOND  ) WITH (  'connector' = 'kafka',  'topic' = 'logins',  'properties.bootstrap.servers' = 'localhost:9092',  'properties.group.id' = 'flink-sql-group',  'scan.startup.mode' = 'latest-offset',  'format' = 'json'  )  """)   # 4. Create a sink table to print results to the console  table\_env.execute\_sql("""  CREATE TABLE print\_sink (  window\_end TIMESTAMP(3),  login\_count BIGINT  ) WITH (  'connector' = 'print'  )  """)   # 5. Define and execute the main analytical query  table\_env.execute\_sql("""  INSERT INTO print\_sink  SELECT  TUMBLE\_END(ts, INTERVAL '1' MINUTE) as window\_end,  COUNT(user\_id) as login\_count  FROM logins  GROUP BY TUMBLE(ts, INTERVAL '1' MINUTE)  """)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

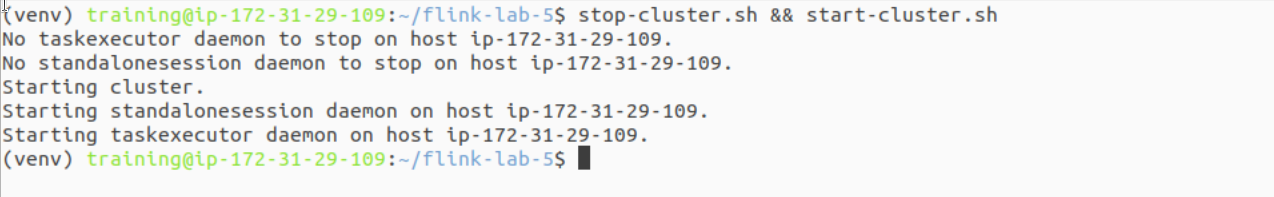
# Part 4: Executing and Simulating Failure

This part is interactive and requires three separate terminal windows.

**Phase 1: Start All Components**

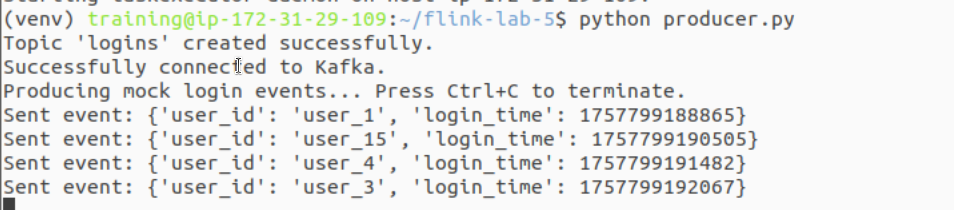
**Start Flink Cluster (Terminal 1):** If your Flink cluster is not already running, restart it.

|  |
| --- |
| stop-cluster.sh && start-cluster.sh |



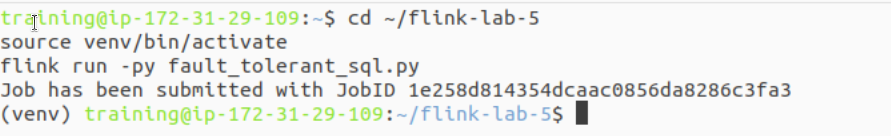
**Launch Data Producer (Terminal 2):**

|  |
| --- |
| cd ~/flink-lab-5 source venv/bin/activate python producer.py |



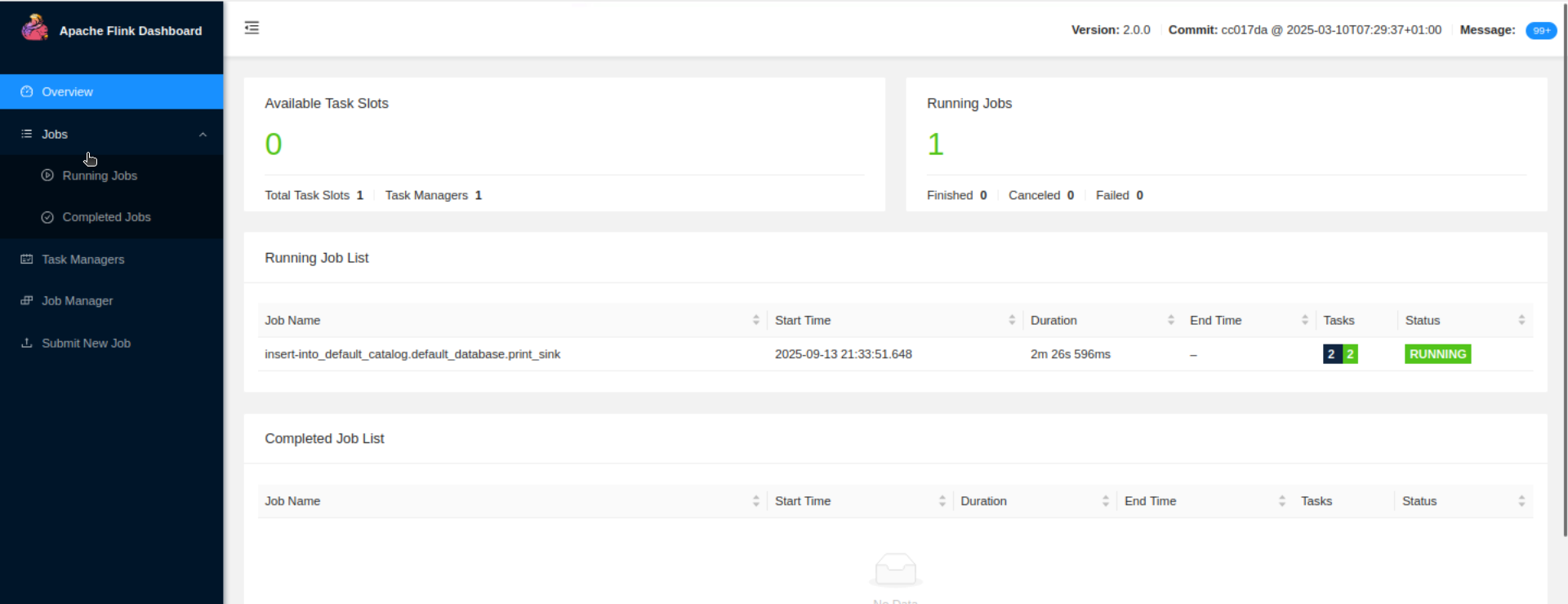
**Submit Flink Job (Terminal 3):**

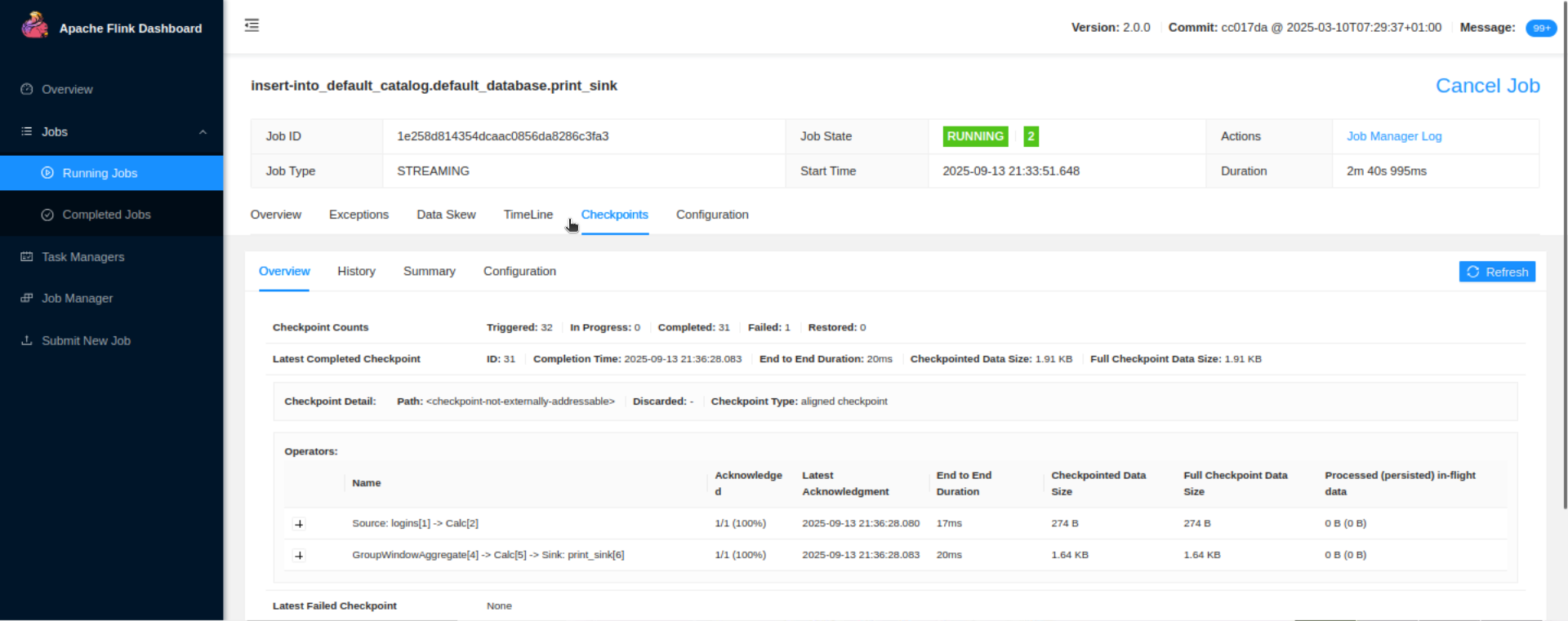
|  |
| --- |
| cd ~/flink-lab-5 source venv/bin/activate flink run -py fault\_tolerant\_sql.py |



**Phase 2: Observe Normal Operation**

**Check Flink UI:** Go to http://localhost:8081. Click on your running job and go to the "Checkpoints" tab. You should see checkpoints being successfully completed every 5 seconds.

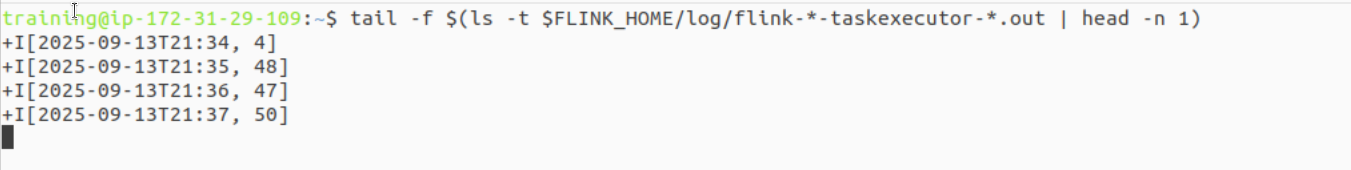
****

****

**Check Logs:** After a minute, view the TaskManager logs. You will see the output of the print sink with the per-minute login counts.

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

The output will look similar to this: +I[2025-09-13T13:45:00, 28]

****

**Phase 3: Simulate a Job Failure**

Let the job run for at least 2 minutes to build up some state.

In the Flink UI, copy the **Job ID** of the running job.

**Cancel the job** by pressing Ctrl+C in Terminal 3.

**Keep the producer in Terminal 2 running!** It will continue sending events to Kafka while the Flink job is down.

**Phase 4: Recover the Job**

Wait for about 30-60 seconds to allow more data to accumulate in Kafka.

In Terminal 3, **resubmit the exact same Flink job again**:

|  |
| --- |
| flink run -py fault\_tolerant\_sql.py |

# Part 5: Verification

Once the job restarts, monitor the TaskManager logs again. You will see the job immediately process the backlog of data that was produced while it was down.

Crucially, you will see that the counts for the time windows during the downtime are correct. Flink did not restart the counts from zero; it restored the state from the last checkpoint and continued processing from the saved Kafka offset. This demonstrates a successful, fault-tolerant recovery.

### 

### **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-5 && docker compose down

### **Part 7: Next Steps**

* **Different Checkpoint Backends:** Explore using a more robust checkpoint backend like a file system (HDFS, S3) instead of the JobManager's memory.
* **Savepoints:** Learn about Flink Savepoints for planned application upgrades and maintenance.
* **Exactly-Once Sinks:** Experiment with a transactional sink (like the Kafka sink) to achieve end-to-end exactly-once guarantees.