Lab 6: Aggregating with Tumbling Windows and Handling Late Data

**Goal:** Implement a fixed-time window aggregation using Flink SQL and directly observe how Flink's watermark mechanism handles late-arriving events to ensure data correctness.

# 

# INDEX

[**Purpose of this Lab**](#_6ogs1mpd0e52) **3**

[**Prerequisites**](#_xmmh3nn6kbk3) **4**

[**Project Structure**](#_ftrnnq4fc4k) **4**

[**Part 1: Project and Environment Setup**](#_29c507jd2ulu) **4**

[**Part 2: Setting Up the Kafka Cluster**](#_g73o7mmp98dp) **6**

[**Part 3: Developing the Flink Windowing Application**](#_q1xou7ia3cau) **7**

[**Part 4: Executing and Observing Late Data**](#_xbkveia3b5n0) **12**

[**Part 5: Verification**](#_r8mm95j8bwf6) **14**

[**Part 6: Cleanup**](#_gopfb0c7d6yc) **15**

[**Part 7: Next Steps**](#_tzxo9ur5b22w) **15**

# 

# Purpose of this Lab

This lab dives into one of the most fundamental concepts of stream processing: time-based windowing. You will analyze a stream of sensor data, grouping events into fixed 10-second windows (Tumbling Windows) to calculate an average.

The core of this lab is to understand how Flink uses **event time** and **watermarks** to create consistent, reproducible results, even when data arrives out-of-order or late. You will run a streaming job, observe windows being processed and finalized, and then deliberately introduce a late event to see how the system ignores it, thereby protecting the integrity of your analytics.

By completing this lab, you will:

**Master Event Time:** Explicitly define an event-time attribute and a watermark strategy in a Flink SQL DDL statement.

**Implement Tumbling Windows:** Use the TUMBLE function in a streaming SQL query to group data into fixed, non-overlapping time intervals.

**Understand Watermarks:** Visualize how watermarks trigger window computations and provide a guarantee of completeness.

**Handle Late Data:** Simulate the arrival of a late event and verify that Flink correctly drops it to prevent corrupting a finalized window calculation.

**Build a Complete SQL Pipeline:** Continue to practice defining sources, sinks, and transformations purely through declarative SQL.

# Prerequisites

This lab assumes you have successfully completed Labs 1 through 5 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-6/ ├── venv/ # The isolated Python virtual environment ├── docker-compose.yaml # Defines our Kafka service (reused) ├── producer.py # Script to generate mock sensor data, including a late event └── window\_aggregation.py # The Flink SQL job script for this lab |

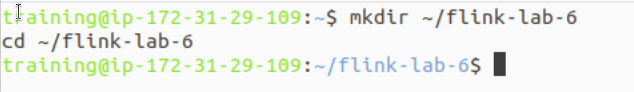
# Part 1: Project and Environment Setup

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

We'll create a new, separate directory for this lab to keep our projects organized.

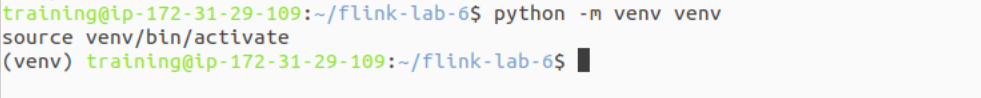
Create and navigate to the new lab directory:

|  |
| --- |
| mkdir ~/flink-lab-6 cd ~/flink-lab-6 |



Initialize and activate a Python virtual environment:

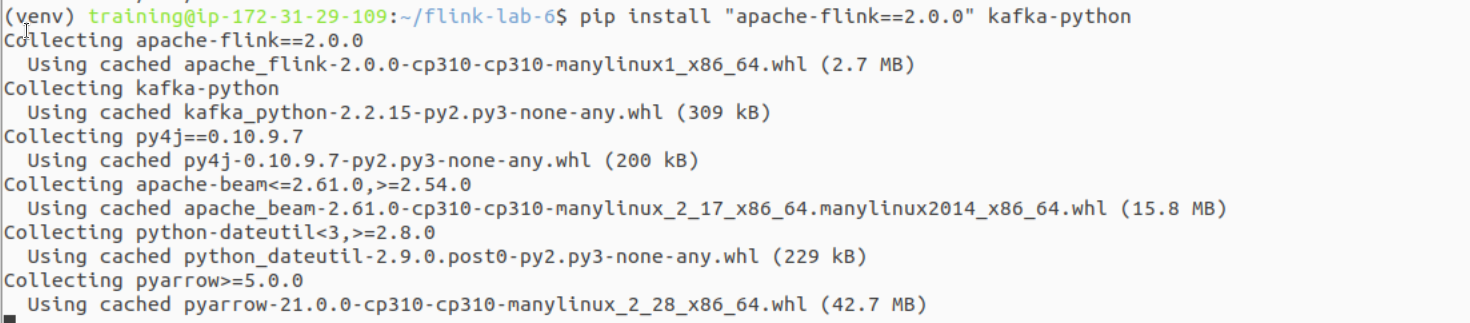
|  |
| --- |
| python -m venv venv source venv/bin/activate |



**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**

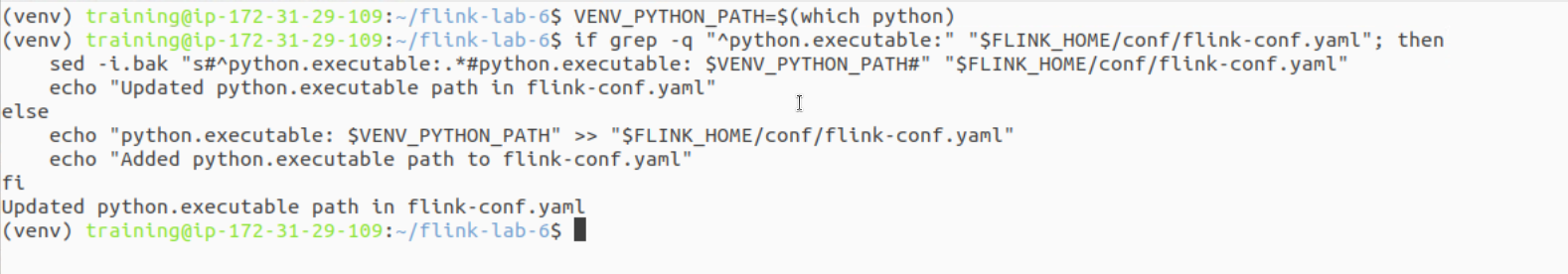
We must update Flink's configuration to point to the correct Python executable for this new lab's virtual environment.

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-6 directory. We will reuse the same configuration from Lab 5.

Create the file:

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

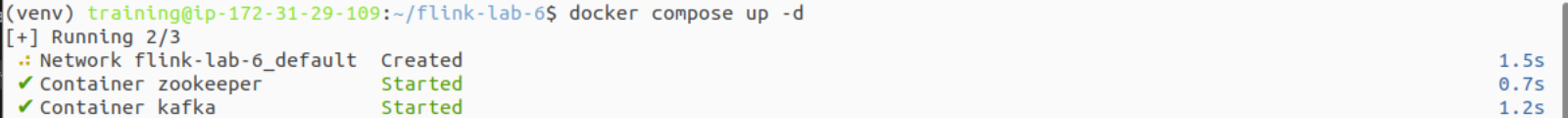
Add the following content to the file:

|  |
| --- |
| # 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-6 directory, start the services in detached mode.

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



### **Part 3: Developing the Flink Windowing Application**

**Step 1: Implement the Kafka Producer**

Create a file named producer.py. This script will generate mock sensor readings. It has a special feature: after running for a while, it will send a single "late" event with a timestamp from the past.

Create the file:

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

Add the following code:

|  |
| --- |
| # producer.py import json import time import random from datetime import datetime from kafka import KafkaProducer from kafka.admin import KafkaAdminClient, NewTopic from kafka.errors import TopicAlreadyExistsError, NoBrokersAvailable  KAFKA\_TOPIC = 'sensor\_readings' 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:  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()   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\_and\_topic()   print("Producing mock sensor events... Press Ctrl+C to terminate.")  sensor\_ids = [f'sensor\_{i}' for i in range(1, 6)]   try:  # Produce normal, timely events for ~45 seconds  print("\n--- Producing in-order events for 45 seconds ---")  start\_time = time.time()  while time.time() - start\_time < 45:  event\_time = int(time.time() \* 1000)  event = {  'sensor\_id': random.choice(sensor\_ids),  'temperature': round(random.uniform(15.0, 30.0), 2),  'event\_time': event\_time   }  producer.send(KAFKA\_TOPIC, value=event)  print(f"Sent event: {event} at {datetime.fromtimestamp(event\_time/1000)}")  time.sleep(random.uniform(0.5, 1.5))   # Now, send a deliberately late event  print("\n--- Sending a LATE event ---")  late\_event\_time = int((time.time() - 30) \* 1000) # 30 seconds in the past  late\_event = {  'sensor\_id': 'sensor\_late',  'temperature': 99.99,  'event\_time': late\_event\_time  }  producer.send(KAFKA\_TOPIC, value=late\_event)  print(f"Sent LATE event: {late\_event} for time {datetime.fromtimestamp(late\_event\_time/1000)}")   # Continue producing normal events  print("\n--- Resuming normal production ---")  while True:  event\_time = int(time.time() \* 1000)  event = {  'sensor\_id': random.choice(sensor\_ids),  'temperature': round(random.uniform(15.0, 30.0), 2),  'event\_time': event\_time  }  producer.send(KAFKA\_TOPIC, value=event)  print(f"Sent event: {event} at {datetime.fromtimestamp(event\_time/1000)}")  time.sleep(random.uniform(0.5, 1.5))   except KeyboardInterrupt:  print("\nStopping producer.")  finally:  producer.flush()  producer.close() |

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

Create the main Flink application file, window\_aggregation.py. This script defines the entire windowing logic.

Create the file:

|  |
| --- |
| code window\_aggregation.py |

Add the following code:

|  |
| --- |
| # window\_aggregation.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)  # Checkpointing is good practice for all stateful jobs  env.enable\_checkpointing(5000)   # 2. Create a source table from the 'sensor\_readings' Kafka topic  table\_env.execute\_sql("""  CREATE TABLE sensor\_readings (  sensor\_id STRING,  temperature DOUBLE,  event\_time BIGINT,  -- Define a computed column for the event-time attribute  ts AS TO\_TIMESTAMP\_LTZ(event\_time, 3),  -- Define a watermark strategy. Windows will be triggered based on this.  -- This means we expect events to be at most 5 seconds late.  WATERMARK FOR ts AS ts - INTERVAL '5' SECOND  ) WITH (  'connector' = 'kafka',  'topic' = 'sensor\_readings',  'properties.bootstrap.servers' = 'localhost:9092',  'properties.group.id' = 'flink-sql-group-windows',  'scan.startup.mode' = 'earliest-offset',  'format' = 'json'  )  """)   # 3. Create a sink table to print results to the console  table\_env.execute\_sql("""  CREATE TABLE print\_sink (  window\_end TIMESTAMP(3),  avg\_temp DOUBLE  ) WITH (  'connector' = 'print'  )  """)   # 4. Define and execute the windowed aggregation query  # This query calculates the average temperature over 10-second tumbling windows.  table\_env.execute\_sql("""  INSERT INTO print\_sink  SELECT  -- The end timestamp of the 10-second window  TUMBLE\_END(ts, INTERVAL '10' SECOND) as window\_end,  -- The average temperature for all events in that window  AVG(temperature) as avg\_temp  FROM sensor\_readings  GROUP BY  TUMBLE(ts, INTERVAL '10' SECOND)  """)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

### **Part 4: Executing and Observing Late Data**

This part is interactive and requires three separate terminal windows.

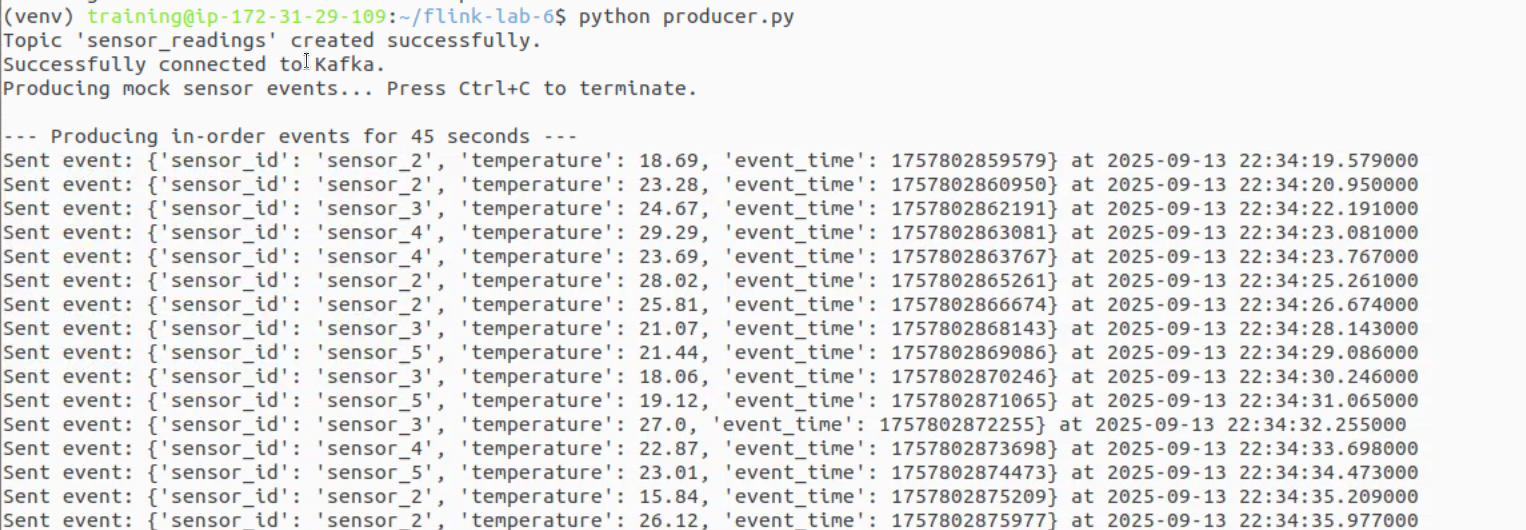
**Phase 1: Start All Components**

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

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

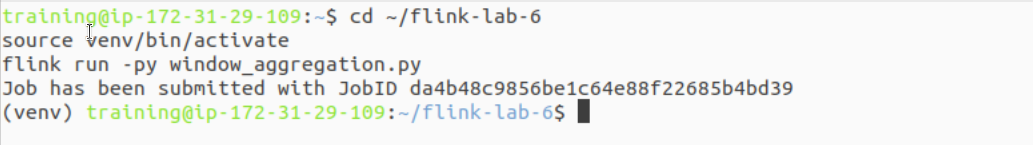
**Launch Data Producer (Terminal 2):**

|  |
| --- |
| cd ~/flink-lab-6 source venv/bin/activate python producer.py |



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

|  |
| --- |
| cd ~/flink-lab-6 source venv/bin/activate flink run -py window\_aggregation.py |



**Phase 2: Observe Normal Window Operation**

**Check Producer Output (Terminal 2):** Watch the producer send normal events. Note the timestamps.

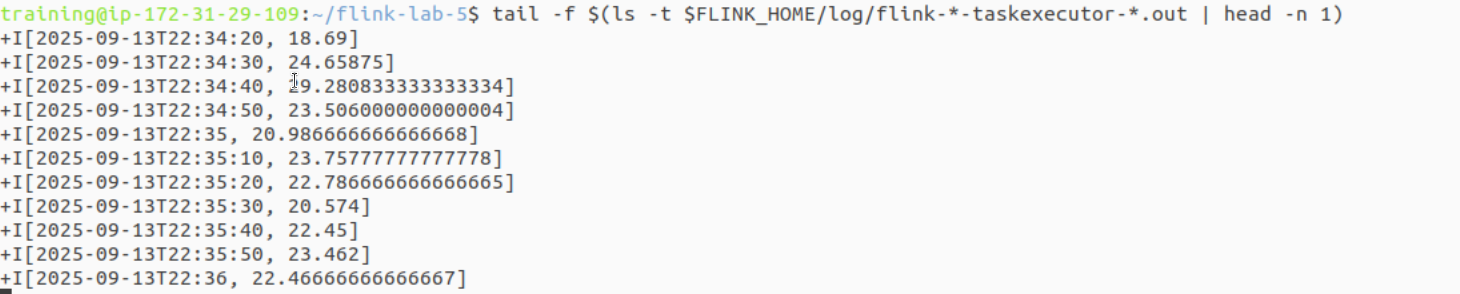
**Check Flink Logs (Terminal 3 or TM logs):** After 10-15 seconds, you will start to see the output from the print sink. The watermark (which is 5 seconds behind the latest event time) must pass the end of a window for that window's result to be calculated and emitted.

Find the TaskManager log file:

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

The output will look similar to this, showing the window's end time and the calculated average temperature:

|  |
| --- |
| +I[2025-09-14T03:15:20, 22.754]  +I[2025-09-14T03:15:30, 21.981]  +I[2025-09-14T03:15:40, 24.115] |



**Phase 3: Observe the Late Event**

After about 45 seconds, the producer in **Terminal 2** will print a message that it is sending a **LATE event**. The event has a timestamp that is 30 seconds in the past. This timestamp falls into a window that Flink has *already calculated and finalized*.

### **Part 5: Verification**

Monitor the TaskManager logs in **Terminal 3**.

You will see that the high temperature value (99.99) from the late event **does not** appear in the output. The average temperature for the old, closed window is not re-calculated or updated.

This is the key takeaway: because the watermark had already advanced far past the window containing the late event, Flink correctly dropped the event. This behavior is essential for ensuring that once a result is published, it is correct and final. You have successfully demonstrated how event time and watermarks provide correctness guarantees.

### **Part 6: Cleanup**

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

**Stop the Flink job:** Press Ctrl+C in Terminal 3.

**Stop the producer:** Press Ctrl+C in Terminal 2.

**Stop the Flink cluster:** stop-cluster.sh

**Stop the Kafka cluster:** cd ~/flink-lab-6 && docker compose down -v

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

**Allowed Lateness:** Modify the SQL query to include an allowedLateness period. This allows a window to accept late events for a specified duration after the watermark has passed, updating its result.

**Side Output for Late Data:** Explore the DataStream API's sideOutputLateData feature to capture events that are too late and route them to a separate stream or sink for auditing or reprocessing.

**Hopping and Sliding Windows:** Experiment with other window types, such as HOP (overlapping windows) and CUMULATE (expanding windows), to solve different analytical problems.