# **Lab 7: Enriching Streams and Handling Data Skew**

**Goal:** Perform a real-time join between two Kafka streams to enrich event data. Deliberately introduce data skew to observe its impact on Flink's performance and discuss potential solutions.

## **INDEX**

**Purpose of this Lab 3**

**Prerequisites 3**

**Project Structure 3**

**Part 1: Project and Environment Setup 4**

Step 1: Create Project Directory and Virtual Environment 4

Step 2: Install Python Dependencies 5

Step 3: Configure Flink for the New Project 5

[Step 4: Increase The Parallelism Capacity](#_s9yxe6li1cjl) 6

**Part 2: Setting Up the Kafka Cluster 7**

Step 1: Define the Kafka Service 7

Step 2: Launch the Kafka Cluster 8

**Part 3: Developing the Flink Join Application 9**

Step 1: Implement the Data Producer 9

Step 2: Implement the Flink Stream Join Script 12

**Part 4: Execution and Observation (Balanced Load) 15**

Step 1: Verify Job Execution 17

Step 2: Inspect the Output 19

**Part 5: Introducing and Diagnosing Data Skew 19**

Step 1: Stop and Modify the Producer 19

Step 2: Restart the Producer 22

Step 3: Observe Performance Degradation in the Flink UI 22

**Part 6: Understanding the Problem and Potential Solutions 24**

Potential Solution: Broadcast Joins 24

**Part 7: Cleanup 24**

## **Purpose of this Lab**

This lab simulates a common e-commerce scenario: correlating product impressions with user clicks. You will build a pipeline that joins these two distinct event streams based on a common key and a time interval. By the end of this lab, you will:

* **Enrich Streams:** Implement a SQL-based interval join on two streams using the Flink Table API to combine related events.
* **Generate Complex Data:** Create a producer that sends two different types of JSON events to two separate Kafka topics.
* **Observe Performance:** Witness how a balanced workload performs smoothly within the Flink cluster.
* **Diagnose Bottlenecks:** Introduce data skew (a "hot key") and use the Flink Web UI to observe the resulting backpressure and uneven load distribution.
* **Understand Solutions:** Discuss why data skew is problematic and learn about alternative strategies like Broadcast Joins.

## **Prerequisites**

This lab assumes you have successfully completed Lab 1 and Lab 2 and are using the same Ubuntu VM environment. All necessary tools (Docker, Flink, Python) should already be configured.

## **Project Structure**

We will create a new, self-contained project directory for this lab:

|  |
| --- |
| ~/flink-lab-8/ ├── venv # The isolated Python virtual environment ├── docker-compose.yaml # Defines our Kafka service (copied from Lab 2) ├── producer.py # Script to generate impression and click events └── stream\_join.py # The Flink job script for the interval join |

**Part 1: Project and Environment Setup**

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

We'll start by creating a new directory and setting up a dedicated Python environment.

1. Create and navigate to the new lab directory:

|  |
| --- |
| mkdir ~/flink-lab-8 cd ~/flink-lab-8 |

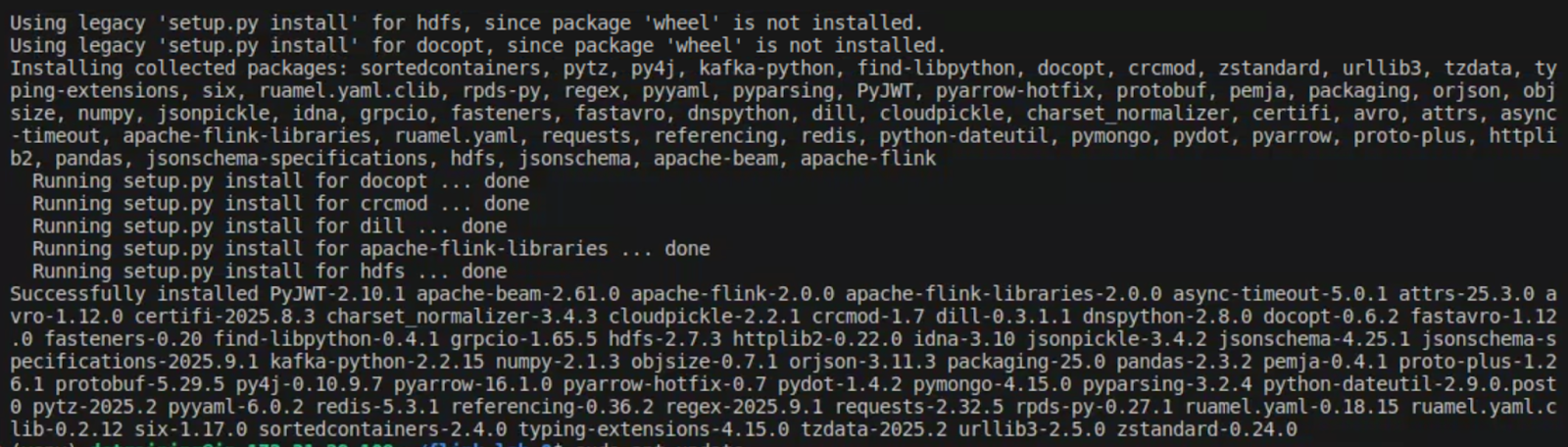
1. Initialize and activate a Python virtual environment:

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

### **Step 2: Install Python Dependencies (if not done already)**

With the venv active, install apache-flink and kafka-python.

|  |
| --- |
| 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 Python executable in our new virtual environment.

1. Get the absolute path to the venv's Python executable:

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

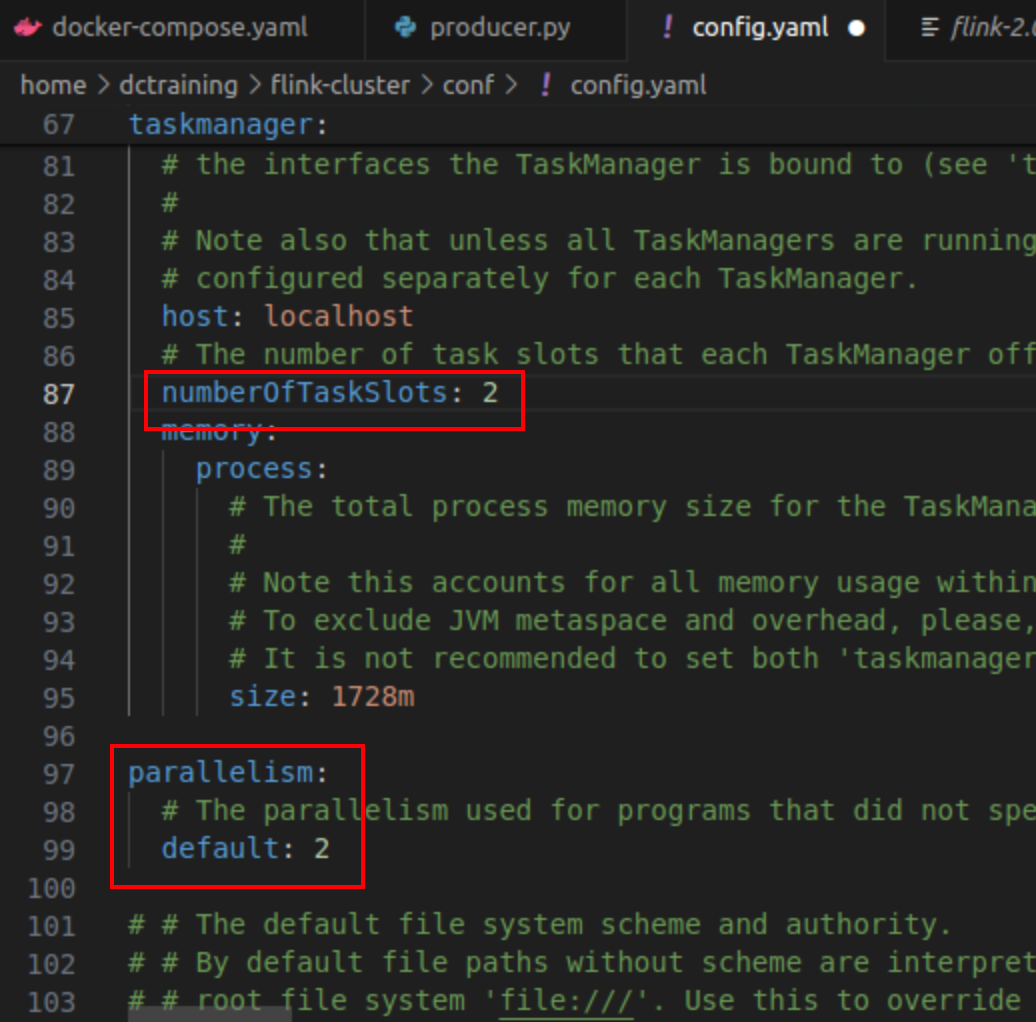
1. Update (or add) the python.executable path in config.yaml: This command is idempotent; it will update the line if it exists or add it if it doesn't.

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

### **Step 4: Increase The Parallelism Capacity**

Go to the config file inside **flink-cluster/conf** and change the default values for:

1. numberOfTaskSlots to 2
2. parallelism to 2



## **Part 2: Setting Up the Kafka Cluster**

We will use the exact same Docker-based Kafka setup as in Lab 2.

### **Step 1: Define the Kafka Service**

1. Create a file named docker-compose.yaml in your ~/flink-lab-8 directory.

|  |
| --- |
| code ~/flink-lab-8/docker-compose.yaml |

1. Copy and paste the following content into the file. This configuration creates two Kafka topics required for this lab: impressions and clicks.

|  |
| --- |
| # 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  KAFKA\_CREATE\_TOPICS: "impressions:2:1,clicks:2:1" # Create our topics on startup |

### **Step 2: Launch the Kafka Cluster**

From the ~/flink-lab-8 directory, start the services in detached mode.

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

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

### **Step 1: Implement the Data Producer**

1. Create a file named producer.py. This script will generate both product impressions and user clicks or create one manually.

|  |
| --- |
| code ~/flink-lab-8/producer.py |

1. Add the following Python code. Pay close attention to the SKEW\_MODE flag, which we will use later.

|  |
| --- |
| # producer.py import json import time import random from kafka import KafkaProducer from kafka.errors import NoBrokersAvailable  # --- Configuration --- KAFKA\_BROKERS = 'localhost:9092' IMPRESSIONS\_TOPIC = 'impressions' CLICKS\_TOPIC = 'clicks'  # Set to True to enable data skew in the second part of the lab SKEW\_MODE = False SKEWED\_PRODUCT\_ID = 'prod\_1'  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.")  def get\_product():  """Returns a product\_id, introducing skew if SKEW\_MODE is True."""  product\_ids = [f'prod\_{i}' for i in range(1, 11)] # prod\_1 to prod\_10  if SKEW\_MODE and random.random() < 0.9: # 90% chance for the skewed product  return SKEWED\_PRODUCT\_ID  return random.choice(product\_ids)  if \_\_name\_\_ == '\_\_main\_\_':  producer = create\_producer()  user\_ids = [f'user\_{i}' for i in range(1, 6)]   print(f"Producer starting in SKEW\_MODE = {SKEW\_MODE}")  print("Producing events... Press Ctrl+C to terminate.")   try:  while True:  product\_id = get\_product()  user\_id = random.choice(user\_ids)  current\_time\_ms = int(time.time() \* 1000)   # 1. Generate an impression  impression\_event = {  'product\_id': product\_id,  'user\_id': user\_id,  'impression\_time': current\_time\_ms  }  producer.send(IMPRESSIONS\_TOPIC, value=impression\_event)  print(f"Sent Impression: {impression\_event}")   # 2. Occasionally generate a corresponding click  if random.random() < 0.4: # 40% chance of a click following an impression  click\_delay = random.randint(500, 4000) # 0.5 to 4 seconds later  click\_event = {  'product\_id': product\_id,  'user\_id': user\_id,  'click\_time': current\_time\_ms + click\_delay  }  time.sleep(click\_delay / 1000) # Simulate user delay  producer.send(CLICKS\_TOPIC, value=click\_event)  print(f"Sent Click: {click\_event}")   time.sleep(0.5) # Wait before next impression   except KeyboardInterrupt:  print("\nStopping producer.")  finally:  producer.flush()  producer.close() |

### **Step 2: Implement the Flink Stream Join Script**

1. Create the main Flink application file, stream\_join.py.

|  |
| --- |
| code ~/flink-lab-8/stream\_join.py |

1. This script defines the core logic. It uses the StreamTableEnvironment to convert the Kafka DataStreams into Tables, defines event time attributes and watermarks, and then performs the interval join using SQL.

|  |
| --- |
| # stream\_join.py from pyflink.common import WatermarkStrategy from pyflink.common.typeinfo import Types from pyflink.datastream import StreamExecutionEnvironment from pyflink.datastream.connectors.kafka import KafkaSource, KafkaOffsetsInitializer from pyflink.datastream.formats.json import JsonRowDeserializationSchema from pyflink.table import StreamTableEnvironment, Schema, DataTypes  def main():  env = StreamExecutionEnvironment.get\_execution\_environment()  # Set parallelism to 2 to better observe skew  env.set\_parallelism(2)  t\_env = StreamTableEnvironment.create(env)   # --- Schema and Deserializer Setup ---  impressions\_type\_info = Types.ROW\_NAMED(  ["product\_id", "user\_id", "impression\_time"],  [Types.STRING(), Types.STRING(), Types.LONG()]  )  clicks\_type\_info = Types.ROW\_NAMED(  ["product\_id", "user\_id", "click\_time"],  [Types.STRING(), Types.STRING(), Types.LONG()]  )   impressions\_deserializer = JsonRowDeserializationSchema.builder() \  .type\_info(impressions\_type\_info).build()   clicks\_deserializer = JsonRowDeserializationSchema.builder() \  .type\_info(clicks\_type\_info).build()   # --- Kafka Sources ---  impressions\_source = KafkaSource.builder() \  .set\_bootstrap\_servers("localhost:9092") \  .set\_topics("impressions") \  .set\_group\_id("flink-join-group") \  .set\_starting\_offsets(KafkaOffsetsInitializer.latest()) \  .set\_value\_only\_deserializer(impressions\_deserializer) \  .build()   clicks\_source = KafkaSource.builder() \  .set\_bootstrap\_servers("localhost:9092") \  .set\_topics("clicks") \  .set\_group\_id("flink-join-group") \  .set\_starting\_offsets(KafkaOffsetsInitializer.latest()) \  .set\_value\_only\_deserializer(clicks\_deserializer) \  .build()   # --- Create DataStreams ---  impressions\_stream = env.from\_source(  source=impressions\_source,  watermark\_strategy=WatermarkStrategy.no\_watermarks(),  source\_name="impressions\_kafka\_source"  )  clicks\_stream = env.from\_source(  source=clicks\_source,  watermark\_strategy=WatermarkStrategy.no\_watermarks(),  source\_name="clicks\_kafka\_source"  )   # --- Convert to Tables and Define Event Time ---  impressions\_table = t\_env.from\_data\_stream(  impressions\_stream,  Schema.new\_builder()  .column("product\_id", DataTypes.STRING())  .column("user\_id", DataTypes.STRING())  .column("impression\_time", DataTypes.BIGINT())  .column\_by\_expression("impression\_ts", "TO\_TIMESTAMP\_LTZ(impression\_time, 3)")  .watermark("impression\_ts", "impression\_ts - INTERVAL '1' SECOND")  .build()  )  clicks\_table = t\_env.from\_data\_stream(  clicks\_stream,  Schema.new\_builder()  .column("product\_id", DataTypes.STRING())  .column("user\_id", DataTypes.STRING())  .column("click\_time", DataTypes.BIGINT())  .column\_by\_expression("click\_ts", "TO\_TIMESTAMP\_LTZ(click\_time, 3)")  .watermark("click\_ts", "click\_ts - INTERVAL '1' SECOND")  .build()  )   t\_env.create\_temporary\_view("impressions", impressions\_table)  t\_env.create\_temporary\_view("clicks", clicks\_table)   # --- Interval Join SQL ---  result\_table = t\_env.sql\_query("""  SELECT  c.product\_id,  c.user\_id,  c.click\_time,  i.impression\_time  FROM clicks AS c  JOIN impressions AS i  ON  c.product\_id = i.product\_id AND  c.user\_id = i.user\_id AND  i.impression\_ts BETWEEN c.click\_ts - INTERVAL '5' SECOND AND c.click\_ts  """)   # --- Convert result table to a DataStream and print to the log ---  t\_env.to\_data\_stream(result\_table).print()   # --- Execute the job ---  env.execute("kafka\_interval\_join")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

## **Part 4: Execution and Observation (Balanced Load)**

First, we'll run the pipeline with an evenly distributed data load. You will need three separate terminals for this.

**Terminal 1: Start/Restart the Flink Cluster**

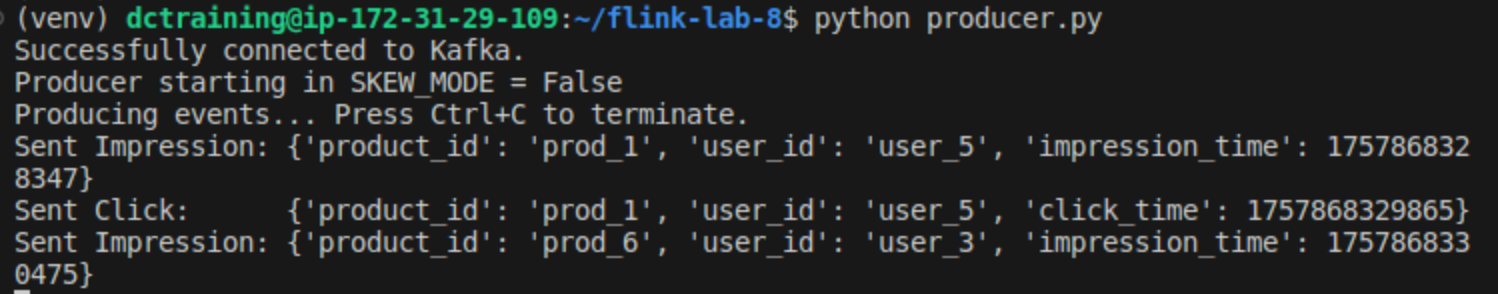
Ensure your cluster is running and configured for this lab's venv.

|  |
| --- |
| # If running, stop it first to apply the new python config stop-cluster.sh  # Start the cluster start-cluster.sh  # Check real-time logs tail -f $(ls -t $FLINK\_HOME/log/flink-\*-taskexecutor-\*.out | head -n 1) |

**Terminal 2: Launch the Data Producer (No Skew)**

Navigate to your lab directory, activate the environment, and start the producer. Ensure SKEW\_MODE is False.

|  |
| --- |
| cd ~/flink-lab-8 source venv/bin/activate python producer.py |

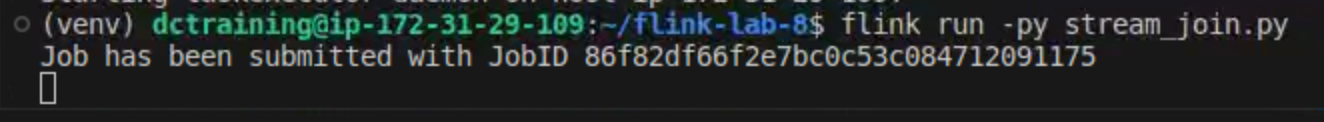


You should see a balanced stream of impressions and click events for different products.

**Terminal 3: Submit the Flink Application**

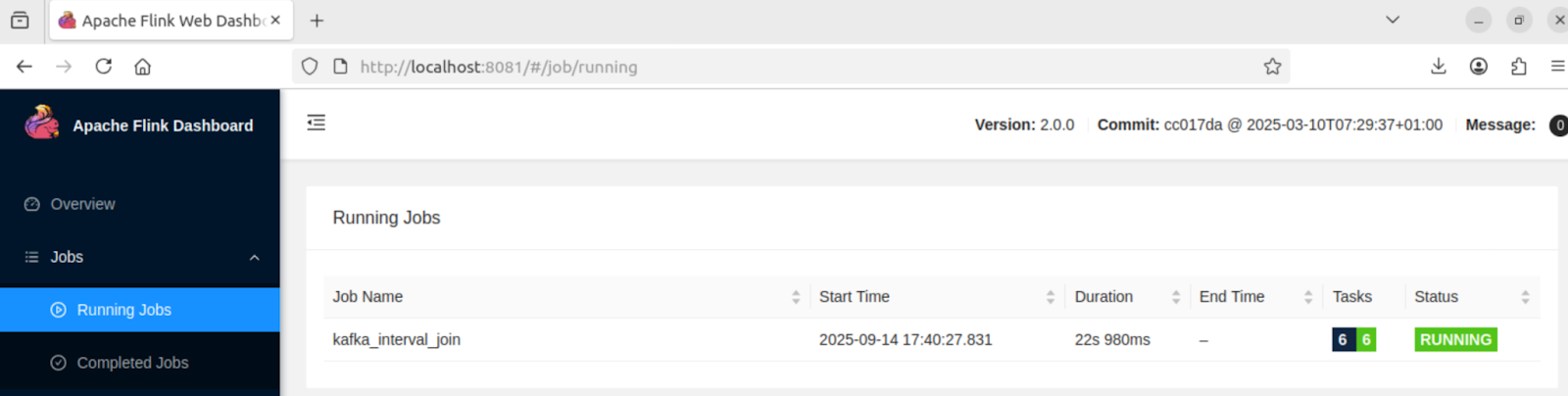
Submit the join job to the cluster.

|  |
| --- |
| cd ~/flink-lab-8 source venv/bin/activate flink run -py stream\_join.py |



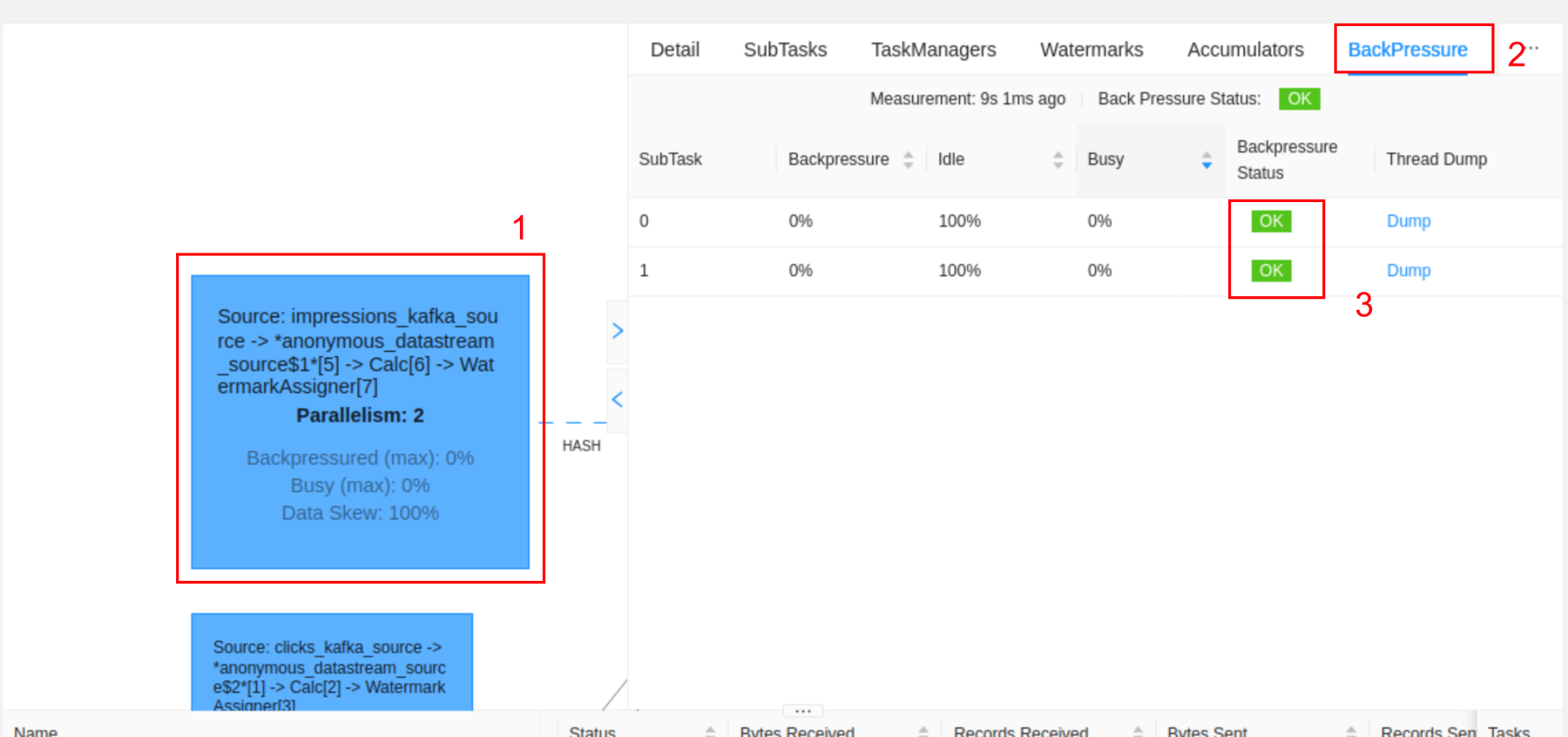
### **Step 1: Verify Job Execution**

1. Go to the Flink UI at http://localhost:8081. You should see the kafka\_interval\_join job running.



1. Click on the job to see the execution graph. Notice the two sources feeding into the join operator.
2. Observe the "Records Sent" and "Records Received" counters. They should be increasing steadily and be roughly balanced between the two parallel tasks.



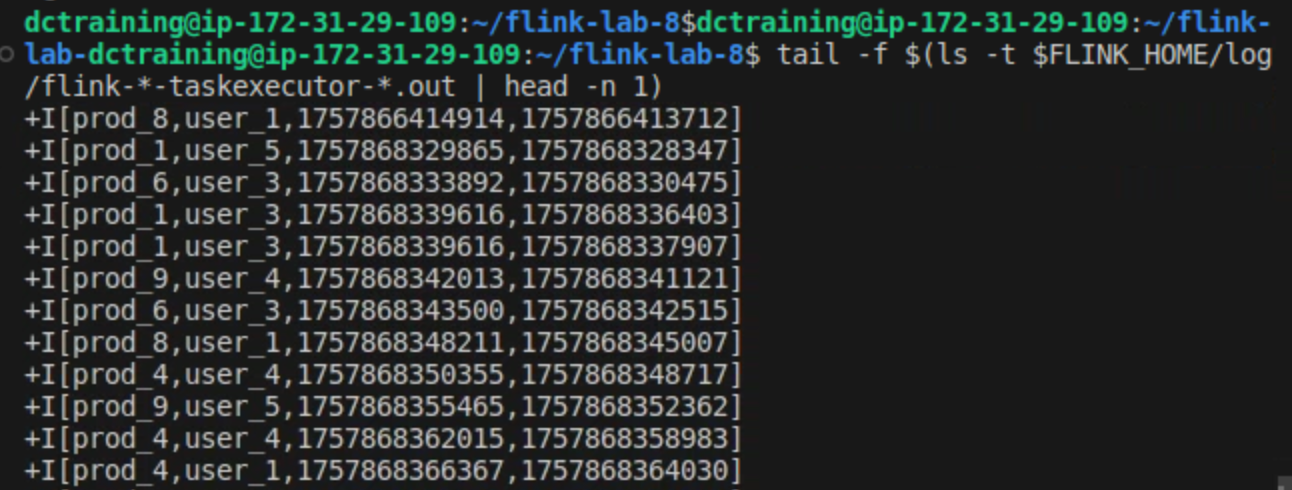


### **Step 2: Inspect the Output**

View the TaskManager logs to see the output of the successful joins.

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

You will see output lines representing the joined rows, like: **+I[prod\_5, user\_2, 1726336205123, 1726336202345].**



## **Part 5: Introducing and Diagnosing Data Skew**

Now, we will intentionally create a performance problem.

### **Step 1: Stop and Modify the Producer**

1. Go to **Terminal 2** (where producer.py is running) and stop it by pressing Ctrl+C.
2. Open the producer.py script: code ~/flink-lab-8/producer.py.
3. Paste the following code

|  |
| --- |
| # producer.py import json import time import random from kafka import KafkaProducer from kafka.errors import NoBrokersAvailable  # --- Configuration --- KAFKA\_BROKERS = 'localhost:9092' IMPRESSIONS\_TOPIC = 'impressions' CLICKS\_TOPIC = 'clicks' SKEW\_MODE = True SKEWED\_PRODUCT\_ID = 'prod\_1'  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.")  def get\_product():  """Returns a product\_id, introducing skew if SKEW\_MODE is True."""  product\_ids = [f'prod\_{i}' for i in range(1, 11)] # prod\_1 to prod\_10  if SKEW\_MODE and random.random() < 0.99:  return SKEWED\_PRODUCT\_ID  return random.choice(product\_ids)  if \_\_name\_\_ == '\_\_main\_\_':  producer = create\_producer()  user\_ids = [f'user\_{i}' for i in range(1, 6)]   # --- NEW: Added a conditional message ---  print(f"Producer starting in SKEW\_MODE = {SKEW\_MODE}")  if SKEW\_MODE:  print("Producing SKEWED events at maximum speed... Press Ctrl+C to terminate.")  else:  print("Producing BALANCED events at maximum speed... Press Ctrl+C to terminate.")   try:  while True:  product\_id = get\_product()  user\_id = random.choice(user\_ids)  current\_time\_ms = int(time.time() \* 1000)   # 1. Generate an impression  impression\_event = {  'product\_id': product\_id,  'user\_id': user\_id,  'impression\_time': current\_time\_ms  }  producer.send(IMPRESSIONS\_TOPIC, value=impression\_event)  # --- NEW: Print the event that was sent ---  print(f"Sent Impression: {impression\_event}")    # 2. Occasionally generate a corresponding click  if random.random() < 0.8:  click\_delay = random.randint(500, 4000)  click\_event = {  'product\_id': product\_id,  'user\_id': user\_id,  'click\_time': current\_time\_ms + click\_delay  }  producer.send(CLICKS\_TOPIC, value=click\_event)  # --- NEW: Print the event that was sent ---  print(f"Sent Click: {click\_event}")    except KeyboardInterrupt:  print("\nStopping producer.")  finally:  producer.flush()  producer.close() |

1. Save the file.

### **Step 2: Restart the Producer**

In **Terminal 2**, restart the producer with the new skewed configuration. The Flink job is still running and will immediately start processing the new data pattern.

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

You will now see that the vast majority of events are for product\_id: 'prod\_1'.

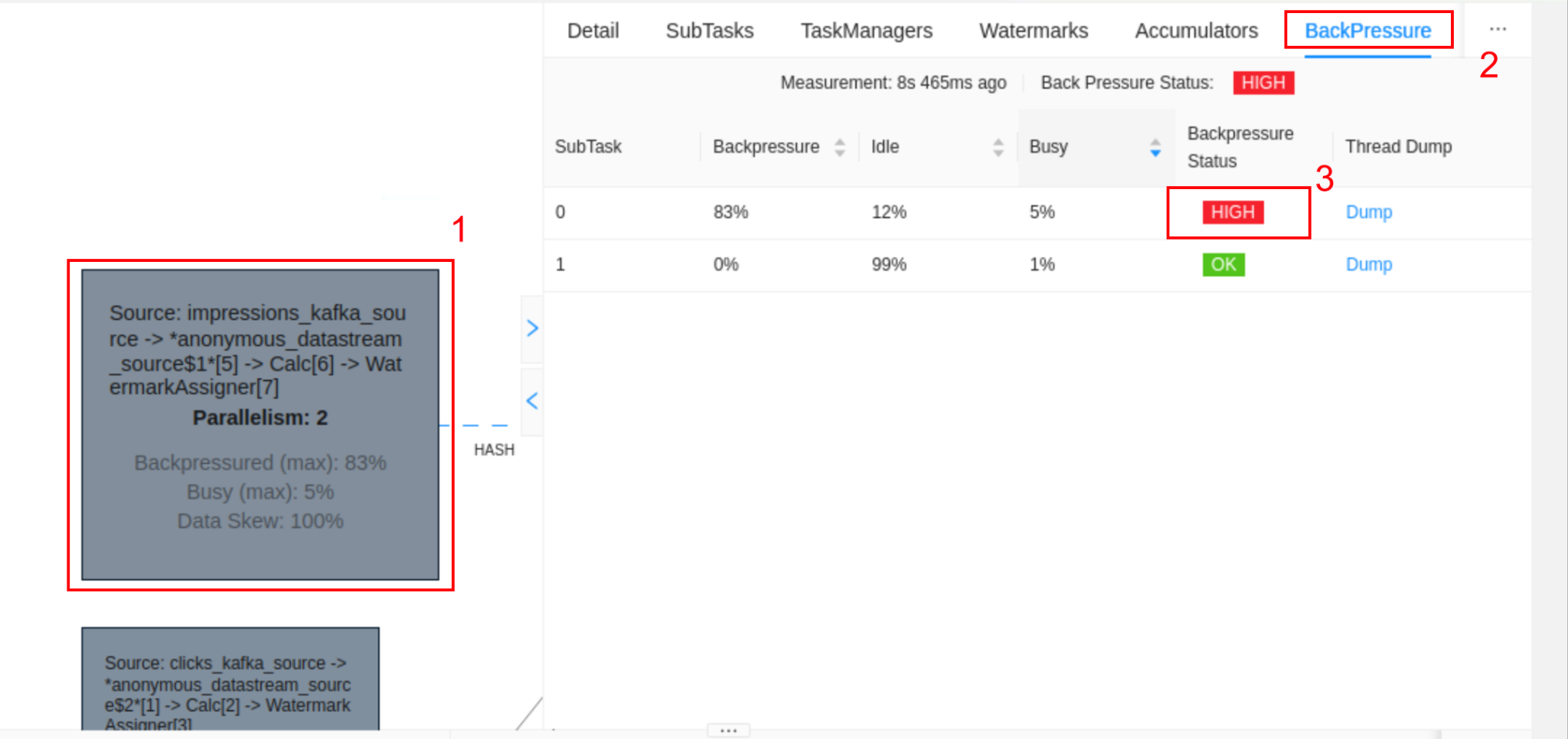


### **Step 3: Observe Performance Degradation in the Flink UI**

Go back to the Flink UI at http://localhost:8081 and refresh the page. After a minute, you will see clear signs of a problem.



* **Uneven Load:** Go back to the job's main "Overview" tab. Look at the "Records Sent" for the source operators. You will notice that one of the two parallel instances is processing significantly more records than the other. This is the core of the problem: Flink's SQL planner sends all records for 'prod\_1' to the same physical task to process the join, overwhelming it while leaving the other task underutilized.
* Go back to the Flink UI. After 60-90 seconds, you will see clear signs of a problem:
* **Uneven Load:** On the "Overview" tab, the "Records Sent" for one IntervalJoin task will be dramatically higher than the other.
* **High Backpressure:** On the "Back Pressure" tab, the status for the overloaded source task will turn to **"HIGH" (red)**.



## **Part 6: Understanding the Problem and Potential Solutions**

You have just witnessed how data skew can cripple a distributed processing pipeline. Even though you have multiple cores available, only one is doing the majority of the work for the "hot key."

**The Cause:** The JOIN ON c.product\_id = i.product\_id ... clause in the SQL query uses a hash of the join key (product\_id) to determine which parallel task instance will process the data. When one key ('prod\_1') appears 90% of the time, the hash is always the same, and the data is always routed to the same task, creating a bottleneck.

### **Potential Solution: Broadcast Joins**

In some scenarios, especially when joining a large stream against a small, slowly changing dataset (like enriching events with user metadata), a **Broadcast Join** is an excellent solution.

* **How it Works:** The small "lookup" stream is not partitioned. Instead, it is broadcasted (sent in its entirety) to every parallel instance of the main stream's processing operator. Each task instance holds a complete local copy of the broadcasted data. When an event from the large stream arrives, it can be joined locally without any network shuffling.
* **When to Use It:** This pattern is ideal when one side of the join is small enough to fit comfortably in the memory of each TaskManager.
* **Why it Solves Skew:** Since the main stream is never shuffled by the skewed key, the load remains perfectly balanced. The bottleneck is eliminated.

For a stream-stream join where both streams are large and skewed, more advanced techniques like two-stage aggregation (key salting) are required, but that is beyond the scope of this lab.

## **Part 7: Cleanup**

1. **Stop the producer:** Press Ctrl+C in Terminal 2.
2. **Stop the Flink job:** Cancel it from the Flink Web UI.
3. **Stop the Flink cluster:** stop-cluster.sh
4. **Stop the Kafka cluster:** cd ~/flink-lab-8 && docker compose down
5. **Deactivate the environment:** deactivate