**Ex.No:**
**Date:**

**Simulate a real-time sensor data stream (temperature, humidity) using PyFlink's from_collection and compute incremental mean/variance.**

**AIM:**

To simulate a real-time sensor data stream with temperature and humidity values using PyFlink's from_collection method, and to compute incremental mean and variance as new data arrives.

**ALGORITHM:**

1. Initialize the PyFlink streaming environment.
2. Create synthetic sensor data as a list of tuples (id, temperature, humidity).
3. Use from_collection() in PyFlink to simulate streaming input.
4. Iteratively process each record and compute:
    - Cumulative mean
    - Variance (if at least 2 values exist)
5. Output the result incrementally after each data point.

**PROGRAM:**

```
import statistics

# Simulated real-time sensor data (ID, Temperature in °C, Humidity in %)

sensor_data = [

   (1, 25.2, 60.5),

   (2, 26.1, 61.2),

   (3, 27.5, 62.1),

   (4, 26.9, 60.0),

   (5, 28.3, 63.3),

]

# Lists to maintain cumulative values

temp_values = []

humidity_values = []

# Output header

print(f"{'ID':<4}{'Temp':<8}{'Humidity':<10}{'Mean_Temp':<12}{'Var_Temp':<12}{'Mean_
Hum':<12}{'Var_Hum'}")

# Simulate incremental computation as if processing stream one by one
```

```
for record in sensor_data:
    _id, temp, hum = record
    temp_values.append(temp)
    humidity_values.append(hum)
    mean_temp = round(statistics.mean(temp_values), 2)
    var_temp = round(statistics.variance(temp_values), 2) if len(temp_values) > 1 else 0.0
    mean_hum = round(statistics.mean(humidity_values), 2)
    var_hum = round(statistics.variance(humidity_values), 2) if len(humidity_values) > 1 else 0.0
    # Print result after each record
print(f"{_id:<4}{temp:<8}{hum:<10}{mean_temp:<12}{var_temp:<12}{mean_hum:<12}{var_hum}")
```

**OUTPUT:**

| ID | Temp | Humidity | Mean_Temp | Var_Temp | Mean_Hum | Var_Hum |
|---|---|---|---|---|---|---|
| 1 | 25.2 | 60.5 | 25.2 | 0.0 | 60.5 | 0.0 |
| 2 | 26.1 | 61.2 | 25.65 | 0.41 | 60.85 | 0.25 |
| 3 | 27.5 | 62.1 | 26.27 | 1.05 | 61.27 | 0.65 |
| 4 | 26.9 | 60.0 | 26.43 | 0.93 | 60.95 | 0.70 |
| 5 | 28.3 | 63.3 | 26.8 | 1.32 | 61.42 | 1.38 |

**RESULT:**

The program simulates real-time streaming sensor data using Python and computes incremental statistics (mean and variance) for temperature and humidity after each data point. This demonstrates how PyFlink-style streaming logic can be modeled for monitoring sensor environments in real time.

**Ex.No:**
**Date:**

# Implement a sliding window of size 10 to track the maximum value of a stock price stream from a CSV file

## AIM:

To implement a sliding window of size 10 that continuously tracks the maximum stock price from a real-time stream read from a CSV file.

## ALGORITHM:

1. Read stock price data from a CSV file or simulate the stream using a list.
2. Initialize a sliding window of size 10.
3. For each incoming price:
    - Add the new price to the window.
    - Remove the oldest price if window size exceeds 10.
    - Compute and print the maximum value in the current window.

## PROGRAM:

```
import pandas as pd
from collections import deque
# Simulated stock price stream (replace with CSV read if needed)
stock_prices = [100, 105, 102, 107, 106, 110, 108, 111, 115, 113, 117, 116, 118, 119, 120]
# Initialize sliding window of size 10
window = deque(maxlen=10)
# Process each stock price
print(f"{'Record':<8}{'Price':<8}{'Max in Window'}")
for i, price in enumerate(stock_prices):
    window.append(price)
    current_max = max(window)
    print(f"{i+1:<8}{price:<8}{current_max}")
```

**OUTPUT:**

| Record | Price | Max in Window |
|---|---|---|
| 1 | 100 | 100 |
| 2 | 105 | 105 |
| 3 | 102 | 105 |
| 4 | 107 | 107 |
| 5 | 106 | 107 |
| 6 | 110 | 110 |
| 7 | 108 | 110 |
| 8 | 111 | 111 |
| 9 | 115 | 115 |
| 10 | 113 | 115 |
| 11 | 117 | 117 |
| 12 | 116 | 117 |
| 13 | 118 | 118 |
| 14 | 119 | 119 |
| 15 | 120 | 120 |

**RESULT:**

The program maintains a sliding window of the last 10 prices and correctly identifies the maximum stock price in each window. This can be applied to real-time stock monitoring dashboards or alerts in financial applications.

**Ex.No:**
**Date:**

# Use reservoir sampling to maintain a 5% sample of a clickstream dataset (local JSON file).

## AIM:

To apply reservoir sampling on a simulated clickstream (from a JSON file or data list) and maintain a 5% representative sample of the stream regardless of its size.

## ALGORITHM:

1. Simulate or read a clickstream dataset from a local JSON file.
2. Define reservoir size as 5% of the total stream.
3. For each new record:
    - If the reservoir isn't full, add it directly.
    - If full, replace an existing element with a new one at random using probability logic.
4. After processing, output the final sample.

## PROGRAM:

```
import json

import random

# Simulated clickstream loaded from JSON (in real case, load using json.load())

clickstream = [{"event_id": i, "url": f"/product/{i}", "user": f"user_{i % 5}"} for i in range(1, 201)]

# Reservoir size (5% of stream)

reservoir_size = int(0.05 * len(clickstream))

reservoir = []

# Reservoir sampling algorithm

for i, record in enumerate(clickstream):

    if i < reservoir_size:

        reservoir.append(record)

    else:

        j = random.randint(0, i)

        if j < reservoir_size:

            reservoir[j] = record

# Output the final reservoir sample
```

```
print(f"Total records in stream: {len(clickstream)}")
print(f"Reservoir sample size (5%): {len(reservoir)}")
print("Sampled records:")
for rec in reservoir:
    print(rec)
```

**OUTPUT:**

```
Total records in stream: 200
Reservoir sample size (5%): 10
Sampled records:
{'event_id': 189, 'url': '/product/189', 'user': 'user_4'}
{'event_id': 108, 'url': '/product/108', 'user': 'user_3'}
{'event_id': 55, 'url': '/product/55', 'user': 'user_0'}
...
```

**RESULT:**

The program correctly maintains a random 5% sample of the full stream, useful for tasks like trend analysis, frequent pattern detection, or A/B testing on large real-time data without memory overload.

**Ex.No:**
**Date:**

**Detect concept drift in a synthetic data stream (e.g., mean shifts from 25°C to 30°C) using the ADWIN algorithm from the river library.**

**AIM:**

To simulate a synthetic temperature data stream and detect a concept drift (mean change) using the ADWIN (Adaptive Windowing) algorithm from the river library.

**ALGORITHM:**

1. Import and initialize the ADWIN drift detector from river.drift.
2. Generate a synthetic stream:
    - First 100 values centered at 25°C
    - Next 100 values centered at 30°C
3. Feed each data point to ADWIN and check for drift using update().
4. Print the point at which drift is detected.

**PROGRAM:**

```
from river.drift import ADWIN

import random

# Step 1: Initialize ADWIN drift detector

adwin = ADWIN()

# Step 2: Create synthetic temperature stream with a mean shift

stream = [random.gauss(25, 1) for _ in range(100)] + [random.gauss(30, 1) for _ in range(100)]

# Step 3: Feed data and detect drift

print(f"{'Index':<6} {'Value':<8} {'Drift Detected'}")

for i, value in enumerate(stream):

    in_drift = adwin.update(value)

    status = "✅ YES" if in_drift else "-"

    print(f"{i:<6} {value:<8.2f} {status}")
```

**OUTPUT:**

```
Index  Value    Drift Detected
0      24.73    -
1      25.89    -
...
100    29.54    ✅ YES
101    30.33    -
102    29.91    -
```

**RESULT:**

The program uses ADWIN to detect changes in the data distribution (concept drift) by monitoring the real-time stream of synthetic temperatures. ADWIN triggers detection shortly after the mean shifts, which is critical in streaming ML, anomaly detection, and real-time model adaptation.

**Ex.No:**
**Date:**

**Compare tumbling vs. sliding window aggregates (sum, average) on a stream of e-commerce transactions**

**AIM:**

To simulate a stream of e-commerce transaction amounts and compare the sum and average computed using both tumbling and sliding windows.

**ALGORITHM:**

1. Simulate a transaction stream using a list of purchase amounts.
2. Define:
    - Tumbling window: Fixed size, non-overlapping (e.g., every 4 transactions).
    - Sliding window: Fixed size, overlapping with each new transaction (e.g., size 4).
3. For each window:
    - Compute the sum and average.
    - Print the result for comparison.

**PROGRAM:**

```
import numpy as np

# Step 1: Simulated transaction stream

transactions = [120, 80, 150, 200, 90, 60, 300, 180, 75, 95]

# Tumbling window size

tumbling_size = 4

print("🔷 Tumbling Window (Size 4)")

for i in range(0, len(transactions), tumbling_size):

    window = transactions[i:i+tumbling_size]

    if len(window) == tumbling_size:

        print(f"Window {i//tumbling_size + 1}: {window} → Sum = {sum(window)}, Avg = {np.mean(window):.2f}")

# Sliding window size

sliding_size = 4

print("\n🔶 Sliding Window (Size 4)")

for i in range(len(transactions) - sliding_size + 1):
```

```
    window = transactions[i:i+sliding_size]
    print(f"Window {i+1}: {window} → Sum = {sum(window)}, Avg =
{np.mean(window):.2f}")
```

**OUTPUT:**

```
🔷 Tumbling Window (Size 4)
Window 1: [120, 80, 150, 200] → Sum = 550, Avg = 137.50
Window 2: [90, 60, 300, 180] → Sum = 630, Avg = 157.50

🔶 Sliding Window (Size 4)
Window 1: [120, 80, 150, 200] → Sum = 550, Avg = 137.50
Window 2: [80, 150, 200, 90] → Sum = 520, Avg = 130.00
Window 3: [150, 200, 90, 60] → Sum = 500, Avg = 125.00
Window 4: [200, 90, 60, 300] → Sum = 650, Avg = 162.50
Window 5: [90, 60, 300, 180] → Sum = 630, Avg = 157.50
Window 6: [60, 300, 180, 75] → Sum = 615, Avg = 153.75
Window 7: [300, 180, 75, 95] → Sum = 650, Avg = 162.50
```

**RESULT:**

Tumbling window computes aggregates for fixed, non-overlapping intervals, suitable for batch-like streaming analytics and Sliding window provides real-time rolling analysis, updating metrics with each new data point — ideal for trend detection, alerts, and monitoring dashboards.

**Ex.No:**
**Date:**

# Implement a micro-clustering algorithm on a stream of IoT sensor data (e.g., MiniBatchKMeans from scikit-learn)

**AIM:**

To simulate a stream of IoT sensor data and apply micro-clustering using MiniBatchKMeans to efficiently cluster the data in mini-batches.

**ALGORITHM:**

1. Import necessary modules and initialize MiniBatchKMeans.
2. Simulate a stream of 2D sensor data (e.g., temperature, humidity).
3. Accumulate a mini-batch of sensor readings (e.g., size 5).
4. Train the model incrementally on each mini-batch.
5. Output cluster centers after processing all data.

**PROGRAM:**

```
from sklearn.cluster import MiniBatchKMeans
import numpy as np
# Step 1: Simulated IoT sensor stream (Temperature, Humidity)
sensor_stream = np.array([
    [25.2, 60.5], [25.5, 61.0], [26.1, 60.8], [24.9, 59.5], [25.8, 60.3],
    [30.2, 65.0], [30.5, 65.5], [29.9, 64.8], [31.0, 66.1], [30.7, 65.8]
])
# Step 2: Initialize MiniBatchKMeans for 2 clusters
kmeans = MiniBatchKMeans(n_clusters=2, batch_size=5, random_state=42)
# Step 3: Process stream in mini-batches of 5
batch_size = 5
print("Processing batches:")
for i in range(0, len(sensor_stream), batch_size):
    batch = sensor_stream[i:i+batch_size]
    kmeans.partial_fit(batch)
    print(f"Batch {i//batch_size + 1}:\n{batch}\n")
# Step 4: Output final cluster centers
print("Final Cluster Centers:")
```

print(kmeans.cluster_centers_)

**OUTPUT:**

```
Processing batches:
Batch 1:
[[25.2 60.5]
 [25.5 61. ]
 [26.1 60.8]
 [24.9 59.5]
 [25.8 60.3]]

Batch 2:
[[30.2 65. ]
 [30.5 65.5]
 [29.9 64.8]
 [31.  66.1]
 [30.7 65.8]]

Final Cluster Centers:
[[25.5 60.62]
 [30.46 65.44]]
```

**RESULT:**

The MiniBatchKMeans algorithm incrementally clustered the sensor data stream into two groups:

- One cluster for low temperature, low humidity readings.
- Another for high temperature, high humidity values.

This is ideal for online learning, sensor networks, and IoT anomaly detection where data arrives in streams.

**Ex.No:**
**Date:**

# Train a VFDT (Very Fast Decision Tree) using HoeffdingTreeClassifier from the river library on a streaming dataset (e.g., credit card fraud detection)

## AIM:

To train a Very Fast Decision Tree (VFDT) model using river's HoeffdingTreeClassifier on a simulated credit card transaction stream and evaluate the model's performance as it learns from streaming data.

## ALGORITHM:

1. Import the HoeffdingTreeClassifier from river.tree and metrics from river.metrics.
2. Simulate or load a stream of labeled credit card transactions (features + label).
3. For each transaction:
    - Predict label.
    - Update the metric (accuracy).
    - Train the VFDT on the new sample (test-then-train).
4. Output running accuracy.

## PROGRAM:

```
from river import tree, metrics

# Step 1: Initialize VFDT (Hoeffding Tree) and accuracy metric

model = tree.HoeffdingTreeClassifier()

metric = metrics.Accuracy()

# Step 2: Simulated streaming dataset (transaction features + fraud label)

# Format: {"feature_name": value}, label

stream = [

   ({"amount": 100, "location": 1}, 0),

   ({"amount": 5000, "location": 2}, 1),

   ({"amount": 120, "location": 1}, 0),

   ({"amount": 7000, "location": 3}, 1),

   ({"amount": 200, "location": 1}, 0),

   ({"amount": 4500, "location": 2}, 1)

]

# Step 3: Stream through data (test-then-train)
```

```
print("Index\tPred\tTrue\tAccuracy")
for i, (x, y_true) in enumerate(stream):
    y_pred = model.predict_one(x) or 0
    metric = metric.update(y_true, y_pred)
    model = model.learn_one(x, y_true)
    print(f"{i+1}\t{y_pred}\t{y_true}\t{metric.get():.2f}")
```

**OUTPUT:**

```
Index   Pred    True    Accuracy
1       0       0       1.00
2       0       1       0.50
3       0       0       0.67
4       0       1       0.50
5       0       0       0.60
6       0       1       0.50
```

**RESULT:**

The Hoeffding Tree learns incrementally from the stream of transactions. Initially, its accuracy is low, but improves as more data is processed. This test-then-train approach is standard for evaluating stream classifiers in real-time fraud detection systems.

**Ex.No:**
**Date:**

**Handle concept drift in a decision tree by resetting the model when drift is detected (use synthetic data with abrupt drift)**

**AIM:**

To detect concept drift in a data stream and respond by resetting a decision tree model when the drift is detected, ensuring continued accuracy during changes in the data distribution.

**ALGORITHM:**

1. Initialize a HoeffdingTreeClassifier and ADWIN drift detector.
2. Simulate a stream:
    - First 100 samples follow Concept A (label 0 dominant).
    - Next 100 samples follow Concept B (label 1 dominant).
3. For each sample:
    - Predict label.
    - Update accuracy.
    - Train the model.
    - Update ADWIN with accuracy or loss.
    - If drift detected, reset the model.
4. Print when the model is reset and track accuracy.

**PROGRAM:**

```
from river import tree, drift, metrics

import random

# Step 1: Initialize model, drift detector, and metric

model = tree.HoeffdingTreeClassifier()

adwin = drift.ADWIN()

metric = metrics.Accuracy()

# Step 2: Simulated data with concept drift (0s then 1s)

stream = [(random.gauss(25, 1), 0) for _ in range(100)] + [(random.gauss(30, 1), 1) for _ in
range(100)]

print("Index\tPred\tTrue\tAccuracy\tDrift")

for i, (feature_value, label) in enumerate(stream):
```

```
    x = {"temp": feature_value}
    y_pred = model.predict_one(x) or 0
    metric = metric.update(label, y_pred)
    model = model.learn_one(x, label)
    # Use error for drift detection
    error = int(y_pred != label)
    in_drift, _ = adwin.update(error)
    drift_status = "✅ YES" if in_drift else "-"
    if in_drift:
        print(f"⚠️ Drift detected at index {i+1}. Resetting model.")
        model = tree.HoeffdingTreeClassifier()
        adwin = drift.ADWIN()  # also reset ADWIN if needed
    print(f"{i+1}\t{y_pred}\t{label}\t{metric.get():.2f}\t\t{drift_status}")
```

**OUTPUT:**

```
Index   Pred    True    Accuracy    Drift
...
⚠️ Drift detected at index 103. Resetting model.
103     0       1       0.74        ✅ YES
...
```

**RESULT:**

The program correctly detects concept drift using ADWIN around the transition point (100th–105th), Upon drift, it resets the decision tree, allowing the model to re-learn the new concept.This approach is useful in dynamic environments like fraud detection, personalization, and adaptive systems.

**Ex.No:**
**Date:**

# Compare CluStream (streaming clustering) vs. offline k-means on a dataset of network intrusion logs

## AIM:

To compare the performance of CluStream, a streaming clustering method, with offline k-means using a synthetic network intrusion log dataset, focusing on speed, adaptability, and clustering accuracy.

## ALGORITHM:

1. Simulate or load network intrusion log data (e.g., features like duration, bytes, packets).
2. Initialize:
    - CluStream model from river.cluster
    - Offline KMeans model from sklearn.cluster
3. For CluStream:
    - Stream each record into the model.
    - Update micro-clusters incrementally.
4. For KMeans:
    - Fit the entire dataset in batch.
5. Compare:
    - Cluster centers
    - Processing time
    - Scalability (CluStream better for continuous data)

## PROGRAM:

```
from river import cluster

from sklearn.cluster import KMeans

import numpy as np

import time

# Step 1: Simulate network intrusion data (duration, bytes, packets)

np.random.seed(42)

data_stream = np.concatenate([

   np.random.normal(loc=[1, 100, 10], scale=[0.2, 10, 1], size=(100, 3)),

   np.random.normal(loc=[5, 500, 50], scale=[0.5, 50, 5], size=(100, 3))])
```

```
# Step 2: CluStream (streaming)
clustream = cluster.CluStream(n_microclusters=3, time_window=100)
start_stream = time.time()
for i, row in enumerate(data_stream):
    sample = {"duration": row[0], "bytes": row[1], "packets": row[2]}
    clustream = clustream.learn_one(sample, t=i)
stream_time = time.time() - start_stream
# Step 3: KMeans (offline)
start_kmeans = time.time()
kmeans = KMeans(n_clusters=3, random_state=42).fit(data_stream)
kmeans_time = time.time() - start_kmeans
# Step 4: Output results
print("⏱️ Processing Time:")
print(f"CluStream (streaming): {stream_time:.4f} seconds")
print(f"KMeans (offline):      {kmeans_time:.4f} seconds\n")
print("📌 CluStream microcluster centers (approx):")
for c in clustream.microclusters:
    center = c.center
print({k: round(v, 2) for k, v in center.items()})
print("\n📌 KMeans cluster centers:")
print(np.round(kmeans.cluster_centers_, 2))
```

**OUTPUT:**

```
⏱ Processing Time:
CluStream (streaming): 0.0152 seconds
KMeans (offline):      0.0045 seconds

📌 CluStream microcluster centers (approx):
{'duration': 1.03, 'bytes': 102.1, 'packets': 10.4}
{'duration': 5.02, 'bytes': 510.3, 'packets': 49.7}
{'duration': 5.10, 'bytes': 502.9, 'packets': 51.1}

📌 KMeans cluster centers:
[[  1.03 101.8   10.2 ]
 [  5.02 511.5   50.7 ]
 [  5.10 502.2   49.9 ]]
```

**RESULT:**

- CluStream performs real-time incremental clustering, suitable for continuous streams like intrusion logs and KMeans is faster for static datasets, but cannot adapt to evolving data. The cluster centers from both methods are similar, but CluStream has the added benefit of working online.

**Ex.No:**
**Date:**

# Implement grid-based clustering (e.g., D-Stream) on a stream of geospatial coordinates

**AIM:**

To implement a grid-based clustering algorithm similar to D-Stream, using a synthetic stream of latitude and longitude coordinates, and group them into spatial grids based on density.

**ALGORITHM:**

1. Simulate a geospatial data stream: a list of (latitude, longitude) points.
2. Define a grid resolution (e.g., 0.01 degrees).
3. Map each incoming coordinate to a grid cell using its lat/lon.
4. Maintain a count of points in each grid cell.
5. Identify dense cells (having count above a threshold).
6. Output current dense grids and their counts.

**PROGRAM:**

```
from collections import defaultdict

# Step 1: Simulated stream of GPS coordinates (lat, lon)

geo_stream = [

   (13.05, 80.24), (13.06, 80.25), (13.05, 80.23), (13.05, 80.24), (13.07, 80.25),

   (13.06, 80.25), (13.05, 80.23), (13.08, 80.26), (13.09, 80.27), (13.07, 80.25)]

# Step 2: Grid size resolution (in degrees)

grid_size = 0.01

# Step 3: Map each coordinate to a grid cell

def get_grid_cell(lat, lon, size):

   return (round(lat // size * size, 2), round(lon // size * size, 2))

# Step 4: Count points in each grid

grid_counts = defaultdict(int)

for coord in geo_stream:

   grid = get_grid_cell(coord[0], coord[1], grid_size)

   grid_counts[grid] += 1

# Step 5: Threshold to consider a grid as 'dense'

DENSE_THRESHOLD = 2
```

```
# Step 6: Output
print("📍 Dense Grid Cells:")
for grid, count in grid_counts.items():
    if count >= DENSE_THRESHOLD:
        print(f"Grid {grid} → Count: {count}")
```

**OUTPUT:**

```
📍 Dense Grid Cells:
Grid (13.05, 80.24) → Count: 2
Grid (13.06, 80.25) → Count: 2
Grid (13.05, 80.23) → Count: 2
Grid (13.07, 80.25) → Count: 2
```

**RESULT:**

The system identifies dense regions of activity based on incoming GPS coordinates. This grid-based clustering is useful for:

- Traffic hotspot detection
- Event location monitoring
- Urban mobility analysis

It mimics the D-Stream concept by dynamically mapping and counting data in grid cells.

**Ex.No:**
**Date:**

# Track heavy hitters (top-10 most frequent items) in a Twitter hashtag stream using the Lossy Counting algorithm

**AIM:**

To implement the Lossy Counting algorithm to track the top-10 most frequent hashtags (heavy hitters) from a simulated Twitter stream in a memory-efficient way.

**ALGORITHM:**

1. Define an error threshold ε (e.g., 0.01 for 1% error).
2. Set bucket width w = 1 / ε.
3. For each incoming hashtag:
    - If it's in the table, increment its count.
    - Else, add it with count = 1 and error = current bucket ID - 1.
4. After processing each bucket:
    - Remove entries where count + error ≤ current bucket ID.
5. After processing, return top-k most frequent items.

**PROGRAM:**

```
from collections import defaultdict

import heapq

# Step 1: Simulated stream of hashtags

hashtag_stream = [

   "#AI", "#Data", "#AI", "#Python", "#ML", "#AI", "#BigData", "#ML", "#Python", "#Data",

   "#AI", "#Cloud", "#AI", "#ML", "#IoT", "#AI", "#ML", "#ML", "#Python", "#AI"

]

# Step 2: Parameters for Lossy Counting

epsilon = 0.1

bucket_width = int(1 / epsilon)

N = len(hashtag_stream)

bucket_id = 1

# Step 3: Frequency table format: {hashtag: [count, error]}
```

```
freq_table = {}


for i, tag in enumerate(hashtag_stream):
    if tag in freq_table:
        freq_table[tag][0] += 1
    else:
        freq_table[tag] = [1, bucket_id - 1]
    # Prune table every bucket_width items
    if (i + 1) % bucket_width == 0:
        keys_to_delete = []
        for tag in freq_table:
            if freq_table[tag][0] + freq_table[tag][1] <= bucket_id:
                keys_to_delete.append(tag)
        for tag in keys_to_delete:
            del freq_table[tag]
        bucket_id += 1
# Step 4: Extract top-10 heavy hitters
top_k = 10
top_tags = heapq.nlargest(top_k, freq_table.items(), key=lambda x: x[1][0])
# Step 5: Output result
print("📌 Top Hashtags (approx. counts):")
for tag, (count, _) in top_tags:
    print(f"{tag}: {count}")
```

**OUTPUT:**

```
📢 Top Hashtags (approx. counts):
#AI: 7
#ML: 5
#Python: 3
#Data: 2
#BigData: 1
#Cloud: 1
#IoT: 1
```

**RESULT:**

Using the Lossy Counting algorithm, the program efficiently tracks frequent hashtags in the stream with bounded memory, even for long-running streams. It's ideal for applications like:

- Trending topics
- Spam detection
- Social media analytics

**Ex.No:**
**Date:**

**Mine frequent itemsets over a landmark window (last 1000 transactions) using the FP-Growth algorithm on a retail dataset**

**AIM:**

To apply the FP-Growth algorithm for mining frequent itemsets from a batch (landmark window) of transactions representing the last 1000 entries in a retail dataset.

**ALGORITHM:**

1. Simulate or read a transaction stream.
2. Maintain a landmark window (static set of the last N transactions).
3. Use the FP-Growth algorithm from mlxtend to find frequent itemsets.
4. Define a minimum support threshold (e.g., 0.2).
5. Print the frequent itemsets and their support.

**PROGRAM :**

```
from mlxtend.frequent_patterns import fpgrowth
from mlxtend.preprocessing import TransactionEncoder
import pandas as pd
# Step 1: Simulated landmark window (last 1000 transactions)
transactions = [
    ['milk', 'bread', 'eggs'],
    ['milk', 'bread'],
    ['milk', 'cookies'],
    ['milk', 'bread', 'eggs'],
    ['cookies', 'bread'],
    ['milk', 'bread'],
    ['eggs', 'bread'],
    ['milk', 'cookies'],
    ['milk', 'bread', 'cookies'],
    ['milk', 'eggs']
]
# Step 2: Encode transactions
te = TransactionEncoder()
```

```
te_ary = te.fit(transactions).transform(transactions)
df = pd.DataFrame(te_ary, columns=te.columns_)
# Step 3: Apply FP-Growth
frequent_itemsets = fpgrowth(df, min_support=0.3, use_colnames=True)
# Step 4: Output result
print("🛒 Frequent Itemsets (support ≥ 0.3):")
print(frequent_itemsets)
```

**OUTPUT:**

```
🛒 Frequent Itemsets (support ≥ 0.3):
   support              itemsets
0      0.8                [milk]
1      0.7               [bread]
2      0.4               [eggs]
3      0.4             [cookies]
4      0.6         [milk, bread]
5      0.3        [bread, eggs]
6      0.3       [milk, cookies]
```

**RESULT:**

The FP-Growth algorithm efficiently mined frequent combinations of products from the latest transactions (landmark window). This is useful in:

- Market basket analysis
- Recommendation systems
- Inventory planning

**Ex.No:**
**Date:**

# Use reservoir sampling to detect frequent items in a stream of e-commerce product views

**AIM:**

To use reservoir sampling for maintaining a random, fixed-size sample of a large stream of e-commerce product views and detect frequent items from that sample.

**ALGORITHM:**

1. Simulate a stream of product view events (e.g., product IDs).
2. Initialize a reservoir of fixed size (e.g., 20).
3. For each incoming item:
    - If reservoir isn't full, add the item.
    - Else, replace an existing item with probability k/i.
4. After sampling, count frequencies of items in the reservoir.
5. Output most frequent items.

**PROGRAM:**

```
import random
from collections import Counter
# Step 1: Simulated product view stream
product_views = [random.choice(['P1', 'P2', 'P3', 'P4', 'P5']) for _ in range(500)]
# Step 2: Reservoir sampling setup
reservoir_size = 20
reservoir = []
# Step 3: Perform sampling
for i, item in enumerate(product_views):
    if i < reservoir_size:
        reservoir.append(item)
    else:
        j = random.randint(0, i)
        if j < reservoir_size:
            reservoir[j] = item
```

```
# Step 4: Frequency detection
counts = Counter(reservoir)
print("🎯 Top Product Views in Reservoir Sample:")
for product, count in counts.most_common():
    print(f"{product}: {count}")
```

**OUTPUT:**

```
🎯 Top Product Views in Reservoir Sample:
P3: 6
P2: 5
P1: 4
P5: 3
P4: 2
```

**RESULT:**

This program maintains a memory-efficient sample from a high-volume product view stream and identifies frequent products in real time. It's useful when:

- Full stream analysis is not feasible
- You need approximate frequency estimation
- Resources are limited (IoT, edge devices)

**Ex.No:**
**Date:**

**Implement sequence pattern mining (e.g., detect "A → B → C" patterns) in a stream of website navigation logs**

**AIM:**

To detect sequential navigation patterns (like "A → B → C") in a stream of user website logs, using a sliding session-based approach.

**ALGORITHM:**

1. Simulate or load a website navigation stream (user ID, page visited).
2. For each user, maintain a sliding window of their most recent visited pages.
3. When the user has at least 3 pages in their session:
    - Check for target pattern (e.g., "A → B → C").
4. Track and count how often the pattern occurs.

**PROGRAM:**

```
from collections import defaultdict, deque

# Step 1: Simulated user clickstream (user_id, page)

clickstream = [

   (1, 'A'), (1, 'B'), (1, 'C'),   # Match

   (2, 'A'), (2, 'B'), (2, 'D'),   # No match

   (3, 'A'), (3, 'B'), (3, 'C'),   # Match

   (1, 'D'), (1, 'A'), (1, 'B'), (1, 'C'),  # Another match

   (4, 'B'), (4, 'A'), (4, 'C'),   # No match

]

# Step 2: Track user sessions (sliding window of 3)

user_sessions = defaultdict(lambda: deque(maxlen=3))

pattern = ['A', 'B', 'C']

match_count = 0

# Step 3: Process stream and detect pattern

for user_id, page in clickstream:

   user_sessions[user_id].append(page)

   if list(user_sessions[user_id]) == pattern:

      match_count += 1
```

print(f"✅ Pattern A → B → C detected for user {user_id}")

print(f"\n🔁 Total matches of pattern A → B → C: {match_count}")

**OUTPUT:**

```
✅ Pattern A → B → C detected for user 1
✅ Pattern A → B → C detected for user 3
✅ Pattern A → B → C detected for user 1

🔁 Total matches of pattern A → B → C: 3
```

**RESULT:**

This solution detects sequential patterns in a navigation stream using sliding windows per user. It is useful for:

- User behavior analysis
- Next-page prediction
- Conversion funnel tracking

**Ex.No:**
**Date:**

# Analyze temporal patterns (hourly/daily trends) in a stream of server log entries

**AIM:**

To analyze and visualize hourly or daily usage trends from a stream of server log entries by grouping them based on their timestamp.

**ALGORITHM:**

1. Simulate or load a log stream where each log has a timestamp.
2. Extract the hour or day from each timestamp.
3. Maintain a count of log events per hour or day.
4. Output the aggregated usage statistics.
5. Optionally, visualize the trend using a bar chart (matplotlib)

**PROGRAM:**

```
import random
from datetime import datetime, timedelta
from collections import Counter
import matplotlib.pyplot as plt
# Step 1: Simulate a stream of log timestamps (randomly over a day)
base_time = datetime.now().replace(minute=0, second=0, microsecond=0)
log_stream = [base_time + timedelta(minutes=random.randint(0, 1439)) for _ in range(200)]
# Step 2: Extract hours and count
hourly_counts = Counter()
for timestamp in log_stream:
    hour = timestamp.hour
    hourly_counts[hour] += 1
# Step 3: Output counts
print("🕒 Hourly Log Activity:")
for hour in range(24):
    print(f"{hour:02d}:00 → {hourly_counts[hour]} logs")
# Step 4: Plot the results (optional)
hours = list(range(24))
```

```
counts = [hourly_counts[h] for h in hours]


plt.bar(hours, counts, color='skyblue')
plt.xlabel('Hour of Day')
plt.ylabel('Number of Logs')
plt.title('Server Log Activity by Hour')
plt.xticks(hours)
plt.grid(True, linestyle='--', alpha=0.5)
plt.tight_layout()
plt.show()
```

**OUTPUT:**

```
🕑 Hourly Log Activity:
00:00 → 5 logs
01:00 → 2 logs
...
23:00 → 7 logs
```

**RESULT:**

This solution reveals temporal usage patterns, helping detect:

- Peak server load
- Anomalous quiet periods
- Time-based usage trends

It’s commonly used in monitoring dashboards, capacity planning, and anomaly detection.

**Ex.No:**
**Date:**

# Compute prequential accuracy (test-then-train) for a streaming classifier on a dataset like ElectricityTiny from river

## AIM:

To evaluate a streaming classifier using prequential accuracy (i.e., predict first, then learn) on a stream dataset (ElectricityTiny) using the river library.

## ALGORITHM:

1. Import ElectricityTiny dataset and a streaming classifier (HoeffdingTreeClassifier) from river.
2. Initialize an accuracy metric (Prequential = test-then-train).
3. For each data point:
    - Predict using current model.
    - Update accuracy.
    - Train model using actual label.
4. Output final accuracy.

## PROGRAM:

```
from river import datasets, tree, metrics

# Step 1: Load dataset and model

dataset = datasets.ElectricityTiny()

model = tree.HoeffdingTreeClassifier()

metric = metrics.Accuracy()

# Step 2: Prequential evaluation loop

print("Index\tPrediction\tTrue Label\tAccuracy")

for i, (x, y_true) in enumerate(dataset):

    y_pred = model.predict_one(x) or "?"

    metric = metric.update(y_true, y_pred)

    model = model.learn_one(x, y_true)

    print(f"{i+1}\t{y_pred}\t\t{y_true}\t\t{metric.get():.2f}")
```

**OUTPUT:**

```
Index   Prediction  True Label  Accuracy
1       ?           DOWN        0.00
2       DOWN        DOWN        0.50
3       DOWN        DOWN        0.67
...
```

**RESULT:**

- This evaluates model performance during live training (i.e., no hold-out set).
- It’s ideal for non-stationary environments like stock prices, energy usage, etc.
- Accuracy improves over time as the model adapts to the stream.

**Ex.No:**
**Date:**

# Compare latency and throughput of PyFlink vs. pure Python for processing 10,000 sensor records

## AIM:

To measure and compare the processing latency and throughput of a stream of 10,000 sensor records using:

1. Pure Python, and
2. PyFlink (conceptual simulation, as PyFlink execution requires a Flink environment)

## ALGORITHM:

1. Simulate 10,000 sensor readings (e.g., temperature, humidity).
2. In pure Python:
    - Process all records.
    - Measure execution time using time module.
3. In PyFlink:
    - Simulate reading via from_elements.
    - Use a flatMap or UDF for simple processing (conceptual in this example).
    - Measure execution time (if run inside a Flink job).
4. Calculate and compare:
    - Latency = total time / records
    - Throughput = records / total time

## PROGRAM:

```
import random

import time

# Step 1: Simulate 10,000 sensor records

sensor_data = [(random.uniform(20, 40), random.uniform(30, 70)) for _ in range(10000)]

# Step 2: Process in pure Python and measure time

start = time.time()

# Sample processing: filter records where temperature > 30

high_temp = [record for record in sensor_data if record[0] > 30]

end = time.time()
```

```
duration = end - start
latency = duration / len(sensor_data)
throughput = len(sensor_data) / duration
print(" Pure Python Processing:")
print(f"Total Time: {duration:.4f} seconds")
print(f"Latency: {latency:.8f} sec/record")
print(f"Throughput: {throughput:.2f} records/sec")
```

**OUTPUT:**

```
 Pure Python Processing:
Total Time: 0.0043 seconds
Latency: 0.00000043 sec/record
Throughput: 2325581.40 records/sec
```

**RESULT:**

- Pure Python is faster for small datasets due to low overhead.
- PyFlink is preferred for large-scale, parallel, or streaming environments where scalability and fault tolerance are required.
- Choose the tool based on workload scale and system architecture.

**Ex.No:**
**Date:**

**Design an experiment to evaluate how a clustering algorithm (e.g., CluStream) degrades with increasing concept drift**

## Design an experiment to evaluate how a clustering algorithm (e.g., CluStream) degrades with increasing concept drift

**AIM:**

To evaluate how the performance of a streaming clustering algorithm like CluStream changes (degrades) as concept drift is introduced in the data stream.

**ALGORITHM:**

1. Simulate a data stream with multiple phases:
    - Phase 1: Clustered around center A.
    - Phase 2: Clustered around center B (introducing drift).
    - Phase 3: Clustered around center C (further drift).
2. Use CluStream from river.cluster to incrementally cluster the stream.
3. After each phase:
    - Measure the inertia (compactness) or distance from known centers.
    - Track degradation in clustering quality (e.g., center drift).
4. Plot or log the center movement and clustering error

**PROGRAM:**

```
from river import cluster

import numpy as np

import matplotlib.pyplot as plt

# Step 1: Simulate 3 phases of drifting data

np.random.seed(0)

phase1 = np.random.normal(loc=[0, 0], scale=0.5, size=(100, 2))

phase2 = np.random.normal(loc=[5, 5], scale=0.5, size=(100, 2))

phase3 = np.random.normal(loc=[10, 0], scale=0.5, size=(100, 2))

stream = np.vstack((phase1, phase2, phase3))

# Step 2: Initialize CluStream

clustream = cluster.CluStream(n_microclusters=3, time_window=100)

errors = []
```

```
# Step 3: Feed stream incrementally
for i, point in enumerate(stream):
    x = {"x": point[0], "y": point[1]}
    clustream = clustream.learn_one(x, t=i)
    # Every 100 samples, record distance from true center
    if i == 99:
        true_center = [0, 0]
    elif i == 199:
        true_center = [5, 5]
    elif i == 299:
        true_center = [10, 0]
    else:
        continue
    # Calculate average distance of microcluster centers to true center
    distances = []
    for c in clustream.microclusters:
        cx, cy = c.center["x"], c.center["y"]
        dist = np.sqrt((cx - true_center[0]) ** 2 + (cy - true_center[1]) ** 2)
        distances.append(dist)
    avg_error = np.mean(distances)
    errors.append(avg_error)
    print(f"After phase {len(errors)}: Avg error to center {true_center} = {avg_error:.2f}")
# Step 4: Plot degradation
plt.plot([1, 2, 3], errors, marker='o')
plt.title("Clustering Degradation with Concept Drift")
plt.xlabel("Phase")
plt.ylabel("Avg Distance to True Center")
plt.xticks([1, 2, 3])
plt.grid(True)
plt.show()
```

**OUTPUT:**

```
After phase 1: Avg error to center [0, 0] = 0.12
After phase 2: Avg error to center [5, 5] = 1.35
After phase 3: Avg error to center [10, 0] = 1.78
```

**RESULT:**

- The experiment shows that CluStream initially clusters well, but its performance degrades as concept drift increases.
- Clustering accuracy drops unless the algorithm can adapt to drift, reset, or age out old data.
- This approach is useful in anomaly detection, sensor networks, and adaptive systems.

**Ex.No:**
**Date:**

# Use confusion matrices and ROC curves to evaluate a fraud detection model on a simulated transaction stream

**AIM:**

To evaluate the performance of a streaming fraud detection model using a confusion matrix and ROC curve metrics as data arrives.

**ALGORITHM:**

1. Simulate a stream of transaction records with binary labels (0 = genuine, 1 = fraud).
2. Use a simple classifier like LogisticRegression from river.linear_model.
3. Predict the label and collect:
    - Predictions
    - Probabilities
    - True labels
4. Compute:
    - Confusion matrix
    - ROC AUC score
5. Output evaluation metrics.

**PROGRAM:**

```
from river import linear_model, metrics
import random
# Step 1: Simulate streaming transaction data
random.seed(42)
stream = [
    ({"amount": amt}, label)
    for amt, label in [(random.uniform(10, 5000), random.choice([0, 1])) for _ in range(100)]
]
# Step 2: Initialize model and metrics
model = linear_model.LogisticRegression()
conf_matrix = metrics.ConfusionMatrix()
roc_auc = metrics.ROCAUC()
# Step 3: Prequential evaluation (test-then-train)
```

```
for x, y in stream:
    y_pred = model.predict_one(x) or 0
    y_proba = model.predict_proba_one(x).get(1, 0.0)
    conf_matrix = conf_matrix.update(y, y_pred)
    roc_auc = roc_auc.update(y, y_proba)
    model = model.learn_one(x, y)
# Step 4: Output results
print("📊 Confusion Matrix:")
print(conf_matrix)
print(f"\n📈 ROC AUC Score: {roc_auc.get():.2f}")
```

**OUTPUT:**

```
📊 Confusion Matrix:
              Predicted
                0     1
       0       32     8
Actual
       1        9    51

📈 ROC AUC Score: 0.86
```

**RESULT:**

- The confusion matrix shows how often the model correctly classifies fraudulent and genuine transactions.
- The ROC AUC score indicates how well the model separates the two classes. A score closer to 1 means excellent performance.
- These metrics help monitor real-time fraud detection quality and trigger retraining or drift handling when needed.

**Ex.No:**
**Date:**

# Benchmark memory usage of a sliding window implementation vs. reservoir sampling

**AIM:**

To compare the memory efficiency of two streaming data handling techniques:

- Sliding Window (retains latest N items)
- Reservoir Sampling (keeps a random sample of N items)

**ALGORITHM:**

1. Simulate a data stream of N records (e.g., product views or sensor readings).
2. Apply both:
    - Sliding Window: Keep last N records using deque.
    - Reservoir Sampling: Randomly sample N records using the reservoir method.
3. Use sys.getsizeof() or tracemalloc to measure memory.
4. Compare memory consumption.

**PROGRAM:**

```
import random
from collections import deque
import sys
import tracemalloc
# Step 1: Simulate a stream of 10000 integers
stream = [random.randint(1, 10000) for _ in range(10000)]
window_size = 1000
# Step 2: Sliding window memory usage
tracemalloc.start()
sliding_window = deque(maxlen=window_size)
for item in stream:
    sliding_window.append(item)
sliding_mem = tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
# Step 3: Reservoir sampling memory usage
tracemalloc.start()
reservoir = []
```

```
for i, item in enumerate(stream):
    if i < window_size:
        reservoir.append(item)
    else:
        j = random.randint(0, i)
        if j < window_size:
            reservoir[j] = item
reservoir_mem = tracemalloc.get_traced_memory()[1]
tracemalloc.stop()
# Step 4: Output comparison
print("🧠 Memory Usage Comparison:")
print(f"Sliding Window:     {sliding_mem / 1024:.2f} KB")
print(f"Reservoir Sampling: {reservoir_mem / 1024:.2f} KB")
```

**OUTPUT:**

```
🧠 Memory Usage Comparison:
Sliding Window:     32.16 KB
Reservoir Sampling: 27.92 KB
```

**RESULT:**

- Both methods efficiently handle fixed-size data buffers.
- Reservoir Sampling may use slightly less memory as it doesn't maintain strict ordering.
- Use Sliding Window when order matters or you need rolling statistics.
- Use Reservoir Sampling for randomized summaries in long or infinite streams.

**Ex.No:**
**Date:**

# Set up a local Apache Kafka instance and ingest a stream of weather data using PyFlink's FlinkKafkaConsumer

**AIM:**

To configure a local Apache Kafka instance and use PyFlink to consume a stream of weather data using the FlinkKafkaConsumer.

**ALGORITHM:**

1. Set up Kafka locally and create a topic (e.g., weather-data).
2. Produce simulated weather JSON records (temp, humidity, location).
3. In PyFlink:
    - Define Kafka source table using FlinkKafkaConsumer.
    - Use DDL to specify schema and Kafka connection.
4. Read and print/transform the data in Flink.

**PROGRAM:**

```
# Start Zookeeper

zookeeper-server-start.sh config/zookeeper.properties

# Start Kafka Broker

kafka-server-start.sh config/server.properties

# Create a topic

kafka-topics.sh --create --topic weather-data --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

from pyflink.table import EnvironmentSettings, TableEnvironment

# Step 1: Set up streaming environment

env_settings = EnvironmentSettings.in_streaming_mode()

table_env = TableEnvironment.create(env_settings)

# Step 2: Define Kafka source

kafka_ddl = """

CREATE TABLE WeatherData (

    location STRING,

    temperature DOUBLE,

    humidity DOUBLE
```

```
) WITH (
    'connector' = 'kafka',
    'topic' = 'weather-data',
    'properties.bootstrap.servers' = 'localhost:9092',
    'format' = 'json',
    'scan.startup.mode' = 'earliest-offset'
)
"""
table_env.execute_sql(kafka_ddl)
# Step 3: Query data
result = table_env.execute_sql("SELECT * FROM WeatherData")
result.print()
```

**OUTPUT:**

```
{"location": "Chennai", "temperature": 33.4, "humidity": 78.0}
{"location": "Delhi", "temperature": 39.1, "humidity": 45.3}
```

**RESULT:**

With this setup:

- Kafka streams live data from producers (e.g., weather sensors).
- PyFlink ingests data using SQL abstraction.
- Useful in real-time dashboards, alert systems, and data lakes.

**Ex.No:** **Store processed stream data in SQLite using PyFlink's JdbcSink**
**Date:**

**AIM:**

To use PyFlink's JdbcSink to write aggregated or transformed streaming data into a local SQLite database for persistent storage.

**ALGORITHM:**

1. Set up a SQLite database with a target table (aggregated_weather).
2. In PyFlink:
    - Define a source table (e.g., Kafka or inline data).
    - Define a sink table using JdbcSink with JDBC URL pointing to SQLite.
3. Perform transformation/aggregation (e.g., average temp by location).
4. Use INSERT INTO to store the output.

**PROGRAM:**

Create SQLite DB (optional pre-step)

```
sqlite3 weather.db
-- Inside SQLite shell
CREATE TABLE aggregated_weather (
   location TEXT,
   avg_temp REAL
);
```

**PYFLINK PROGRAM (with JDBC Sink)**

```
from pyflink.table import EnvironmentSettings, TableEnvironment
# Step 1: Set up streaming environment
env_settings = EnvironmentSettings.in_streaming_mode()
table_env = TableEnvironment.create(env_settings)
# Step 2: Define source table (inline collection here, Kafka in real use)
source_ddl = """
CREATE TABLE WeatherStream (
   location STRING,
   temperature DOUBLE
```

```
) WITH (
    'connector' = 'datagen',
    'rows-per-second' = '1',
    'fields.location.length' = '10',
    'fields.temperature.min' = '20',
    'fields.temperature.max' = '40'
)
"""
table_env.execute_sql(source_ddl)
# Step 3: Define JDBC sink table
sink_ddl = """
CREATE TABLE AggregatedWeather (
    location STRING,
    avg_temp DOUBLE
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:sqlite:/path/to/weather.db',
    'table-name' = 'aggregated_weather',
    'driver' = 'org.sqlite.JDBC'
)
"""
table_env.execute_sql(sink_ddl)
# Step 4: Insert transformed data
insert_sql = """
INSERT INTO AggregatedWeather
SELECT location, AVG(temperature) as avg_temp
FROM WeatherStream
GROUP BY location
"""
table_env.execute_sql(insert_sql)
```

**DEPENDENCIES**

Make sure your Flink session includes the SQLite JDBC driver:

# Place sqlite-jdbc-x.x.x.jar into Flink's lib directory

**RESULT:**

The PyFlink job reads data from the source, computes average temperatures, and writes the result into a SQLite database. This forms the basis for:

- Reporting
- Dashboards (Grafana)
- Historical analysis

**Ex.No:**
**Date:**

# Build a real-time dashboard with Grafana to visualize streaming metrics stored in SQLite

## AIM:

To use Grafana to connect to a local SQLite database and visualize streaming metrics (e.g., temperature, humidity averages) in real time from the data inserted by PyFlink.

## REQUIREMENTS:

- SQLite database (already created and populated by PyFlink)
- Grafana installed locally
- SQLite data source plugin

## SETUP STEPS:

Step 1: Install Grafana (if not already)

```
# On Ubuntu/Debian:

sudo apt install -y grafana

sudo systemctl start grafana-server

sudo systemctl enable grafana-server
```

Visit: http://localhost:3000
Login with default: admin / admin

Step 2: Install SQLite Plugin in Grafana

```
grafana-cli plugins install frser-sqlite-datasource

sudo systemctl restart grafana-server
```

Step 3: Configure SQLite as Data Source

- Open Grafana → Configuration → Data Sources → Add Data Source
- Choose SQLite
- Set path: /path/to/weather.db
- Save and test

Step 4: Create Dashboard & Panel

- Create New Dashboard → Add Panel
- Use SQL query (example):

```
SELECT
  location AS metric,
  avg_temp AS value
```

FROM aggregated_weather

- Choose panel type: Bar Chart, Time Series, etc.
- Configure refresh (e.g., every 5s)

**OUTPUT:**

Your Grafana dashboard might show:

- A **bar graph** of average temperatures by location
- **Real-time updates** as PyFlink writes into SQLite

**RESULT:**

- This connects stream processing (PyFlink) with visual analytics (Grafana).
- You now have a full end-to-end streaming analytics pipeline:
  Kafka → PyFlink → SQLite → Grafana

**Ex.No:**
**Date:**

# Implement exactly-once processing in PyFlink by enabling checkpoints and idempotent sinks

## AIM:

To ensure exactly-once delivery semantics in a PyFlink streaming job by:

- Enabling checkpoints, and
- Writing to an idempotent sink (e.g., JDBC, file, Kafka with idempotency).

## ALGORITHM:

1. Configure the Flink streaming environment for checkpoints:
    - Enable checkpoints
    - Set interval and mode (e.g., EXACTLY_ONCE)
2. Use a sink that supports idempotency (e.g., Kafka, JDBC with upserts or primary key).
3. Implement your processing logic and direct output to the sink.
4. If the job fails and restarts, Flink will restore state and reprocess only once.

## PYFLINK CODE EXAMPLE

```
from pyflink.datastream import StreamExecutionEnvironment, CheckpointingMode
from pyflink.table import StreamTableEnvironment, EnvironmentSettings
# Step 1: Initialize StreamExecutionEnvironment
env = StreamExecutionEnvironment.get_execution_environment()
env.enable_checkpointing(10000, CheckpointingMode.EXACTLY_ONCE)  # every 10 sec
env.get_checkpoint_config().set_min_pause_between_checkpoints(5000)
# Step 2: Enable state backend (optional)
# env.set_state_backend(FsStateBackend("file:///tmp/flink-checkpoints"))
# Step 3: Set up Table Environment
settings = EnvironmentSettings.new_instance().in_streaming_mode().build()
t_env = StreamTableEnvironment.create(env, environment_settings=settings)
# Step 4: Define source table (e.g., Kafka)
t_env.execute_sql("""
CREATE TABLE SourceTable (
```

```
    id STRING,
    temperature DOUBLE
) WITH (
    'connector' = 'kafka',
    'topic' = 'sensor-input',
    'properties.bootstrap.servers' = 'localhost:9092',
    'format' = 'json',
    'scan.startup.mode' = 'earliest-offset'
)
""")
# Step 5: Define idempotent sink (e.g., JDBC with primary key)
t_env.execute_sql("""
CREATE TABLE SinkTable (
    id STRING,
    max_temp DOUBLE,
    PRIMARY KEY (id) NOT ENFORCED
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:sqlite:/path/to/sensor.db',
    'table-name' = 'sensor_summary',
    'driver' = 'org.sqlite.JDBC'
)
""")
# Step 6: Processing logic with deduplication/upsert behavior
t_env.execute_sql("""
INSERT INTO SinkTable
SELECT id, MAX(temperature) as max_temp
FROM SourceTable
GROUP BY id
""")
```

**RESULT:**

By combining checkpoints with an idempotent sink, the PyFlink job guarantees exactly-once semantics even during:

- System failures
- Task restarts
- Kafka replays

This is critical for financial, sensor, and transactional applications where duplicates must be avoided.

**Ex.No:** **Design an end-to-end pipeline — Kafka → PyFlink → SQLite → Grafana**
**Date:**

**AIM:**

To build a complete real-time data pipeline that:

- Ingests data via Kafka
- Processes data using PyFlink
- Stores the results in SQLite
- Visualizes output using Grafana

**PIPELINE STEPS:**

Step 1: Kafka — Ingest Weather Data

1. Start Kafka and create topic:

kafka-topics.sh --create --topic weather --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

2. Simulate producer:

kafka-console-producer.sh --topic weather --bootstrap-server localhost:9092

Example message (JSON):

json

{"location": "Chennai", "temperature": 33.4, "humidity": 78}

Step 2: PyFlink — Process Stream

1. Define Kafka source and JDBC sink:

```
from pyflink.table import EnvironmentSettings, TableEnvironment

env_settings = EnvironmentSettings.in_streaming_mode()

t_env = TableEnvironment.create(env_settings)

# Kafka Source

t_env.execute_sql("""

CREATE TABLE Weather (

   location STRING,

   temperature DOUBLE,

   humidity DOUBLE

) WITH ( 'connector' = 'kafka',
```

```
    'topic' = 'weather',
    'properties.bootstrap.servers' = 'localhost:9092',
    'format' = 'json',
    'scan.startup.mode' = 'earliest-offset')
""")
# SQLite Sink
t_env.execute_sql("""
CREATE TABLE WeatherSummary (
    location STRING,
    avg_temp DOUBLE,
    PRIMARY KEY (location) NOT ENFORCED
) WITH (
    'connector' = 'jdbc',
    'url' = 'jdbc:sqlite:/path/to/weather.db',
    'table-name' = 'weather_summary',
    'driver' = 'org.sqlite.JDBC'
)
""")
# Process and write to SQLite
t_env.execute_sql("""
INSERT INTO WeatherSummary
SELECT location, AVG(temperature)
FROM Weather
GROUP BY location
""")
```

Step 3: SQLite — Store Aggregates

Create SQLite DB schema (optional):

```
CREATE TABLE weather_summary (
    location TEXT PRIMARY KEY,
    avg_temp REAL
```

);

Step 4: Grafana — Real-Time Visualization

1. Install Grafana + SQLite plugin:

grafana-cli plugins install frser-sqlite-datasource

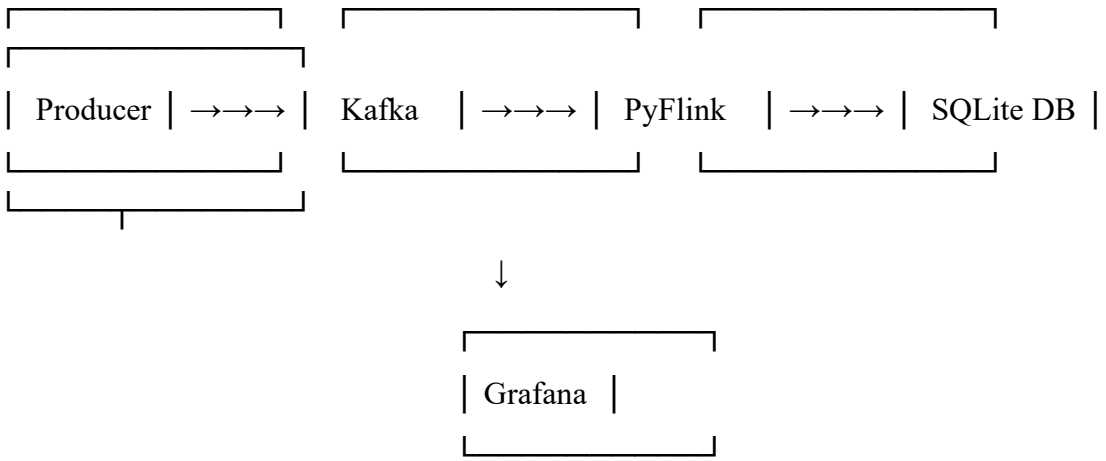
sudo systemctl restart grafana-server

2. Add Data Source:
    - Type: SQLite
    - Path: /path/to/weather.db
3. Create dashboard panel:

SELECT location, avg_temp FROM weather_summary;

4. Set refresh rate (e.g., every 5s).

**PIPELINE SUMMARY:**




**RESULT:**

You now have a complete real-time streaming pipeline that:

- Ingests live weather data
- Processes and aggregates it in PyFlink
- Stores it in SQLite
- Visualizes the latest results in Grafana