Streaming Applications & Analytics

Assignment 1

Group Information:

Group Number: 8
Group Topic: Kafka Streaming SQL
Group Link: https://docs.confluent.io/platform/current/streams-ksql.html

1 Abstract:

Kafka Streams and ksqlDB enable developers to process and analyse streams of real-time data directly from Kafka. The combination of both provides a powerful toolset for developing highly scalable, low-latency, and fault-tolerant stream processing applications. Kafka Streams provides a Java-based API to build stream processing applications, whereas ksqlDB allows users to leverage SQL for stream processing, making it accessible to those familiar with relational databases. This dual approach supports a wide range of use cases, from data transformations and aggregations to machine learning model scoring in real time.

2 Need for Kafka Streams & ksqlDB:

Apache Kafka is a powerful event streaming platform, but it primarily acts as a message broker, designed for high-throughput, fault-tolerant, and scalable real-time data processing, which stores and distributes data streams. However, it does not process or transform data. This is where **Kafka Streams** and **ksqlDB** (formerly ksql) come into play.

Let us firstly understand Apache Kafka in detail!

Key components of Apache Kafka components:

1. **Producer:** Publishes messages (events) to Kafka topics.
2. **Consumer:** Reads messages from Kafka topics, either individually or as part of a consumer group. It uses a Subscription model.
3. **Broker:** A Kafka server acts as a distributed messaging system that stores streams of records in topics. It provides reliable and fault-tolerant storage and supports horizontal scaling.
4. **Topic:** A logical channel to which messages are sent by producers and read by consumers.
5. **Partition:** A topic is divided into partitions to enable parallel processing and scalability.
6. **ZooKeeper** (Legacy, now optional): Manages metadata and leader election. Kafka has an internal replacement now.

7. **The Log:** Immutably stores all the events across Kafka network. It uses some retention policy for storage (default 7 days). It stores directory name by topic name.
8. **Connectors:** We can use Producer & Consumer client API's to send & receive data from Kafka which involves lot of boilerplate coding. Hence, Kafka provides "***Kafka connects***" framework which does all the boilerplate code. Kafka Connect allows integration with external systems like databases, file systems, etc.

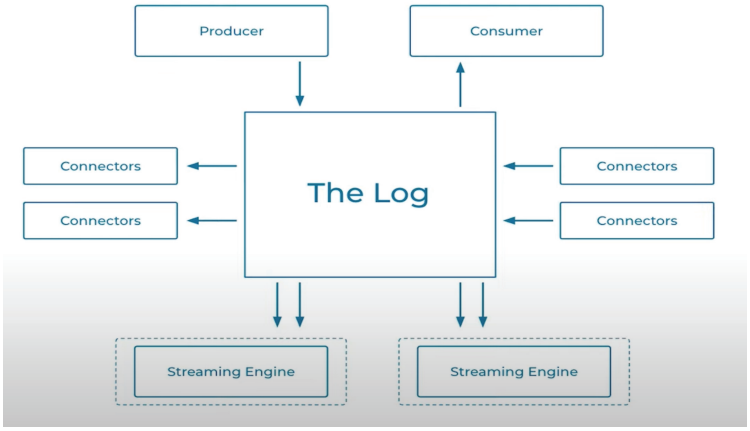


Fig 2.1 Key components of Apache Kafka

How Apache Kafka works:

1. Producers send data (events) to Kafka topics.
2. Data is stored in a durable and distributed manner across multiple brokers.
3. Consumers subscribe to topics and process data in real-time.
4. Kafka supports fault tolerance by replicating partitions across multiple brokers.

Why Kafka streams?

To perform something useful with the data that Apache Kafka provides, we need **Kafka Streams**! It is a streaming engine works on top of Apache Kafka, and part of Confluent platform. Here we focus on "*What to do?*" instead of "*How to do?*". Combining this with Confluent cloud, one can achieve serious stream processing power with very less code investments.

To give an example that how the boilerplate coding investments reduces when we use Kafka streams is depicted with an example below:

Let's consider below stream record:

```
{
        "reading_ts": "2020-02-14T12:19:27Z",
        "sensor_id": "aa-101",
        "production_line": "w01",
        "widget_type": "acme94",
        "temp_celcius": 23,
        "widget_weight_g": 100
}
```

Using Apache Kafka:

```
public static void main (String[] args) {
   try (Consumer<String, Widget > consumer = new KafkaConsumer<> (consumerProperties ());
   Producer<String, Widget> producer = new KafkaProducer<> (producerProperties ())) {
```

```
consumer.subscribe(Collections.singletonList("widgets"));
    while (true) {
        ConsumerRecords<String, Widget> records = consumer.poll(Duration.ofSeconds (5));
        for (ConsumerRecord<String, Wigdet> record : records) {
            Widget widget = record.value ();
            if (widget getColour (). equals("red") {
                ProducerRecord<String, Widget> producerRecord = new ProducerRecord<> ("widgets-red",
record.key(), widget);
                producer send (producerRecord, (metadata, exception) -> (
  ...} ) ;
```

Using Kafka Streams:

```
final StreamsBuilder builder = new StreamsBuilder ();
builder.stream("widgets", Consumed.with(stringSerde, widgetsserde))
   .filter ( (key, widget) -> widget.getcolour.equals("red"))
   .to("widgets-red", Produced.with(stringSerde, widgetsSerde));
```

Kafka vs. Kafka Streams vs. ksqlDB – When to Use What?

Feature	Apache Kafka	Kafka Streams	ksqlDB
Primary Use	Message brokering	Real-time stream processing in Java	Stream processing using SQL
Language	Any (clients in Java, Python, etc.)	Java	SQL
Processing Logic	None (just stores & transfers events)	Custom Java processing	SQL queries on streams
Stateful Processing	❌ No	✅ Yes	✅ Yes
Scalability	✅ High	✅ High	✅ High
Best For	Event-driven architectures	Complex, stateful applications	Quick, ad-hoc stream processing

Fig 2.2 When to use What?

2.1 Overview of Kafka Streams and ksqlDB

Kafka Streams and KSQL are part of **Confluent's Kafka Platform** designed to provide stream processing capabilities on top of Apache Kafka. Both are used for building real-time, distributed data applications.

- **Kafka Streams** Kafka Streams runs as a lightweight Java library that allows for stream processing applications to consume, process, and produce data from/to Kafka topics. It handles fault tolerance, state management, and scalability automatically.
- **Kafka Streaming SQL (ksqlDB)** is a powerful, real-time stream processing platform which has SQL-like interface for stream processing in Kafka, allowing users to interact with Kafka streams through simple SQL queries without needing to write complex code in Java or Scala. It integrates seamlessly with Kafka, providing capabilities to filter, transform, and enrich data streams in real time.

3 Architecture Details

Apart from Kafka components, Kafka streams & ksqlDB maintains "***State Stores***", which are local stores that maintain the intermediate results of stream processing, enabling efficient, fault-tolerant stateful processing. ***Kafka Connect*** provides source and sink connectors that can be used in conjunction with Kafka Streams and ksqlDB to integrate other systems.

3.1 Kappa Architecture Perspective

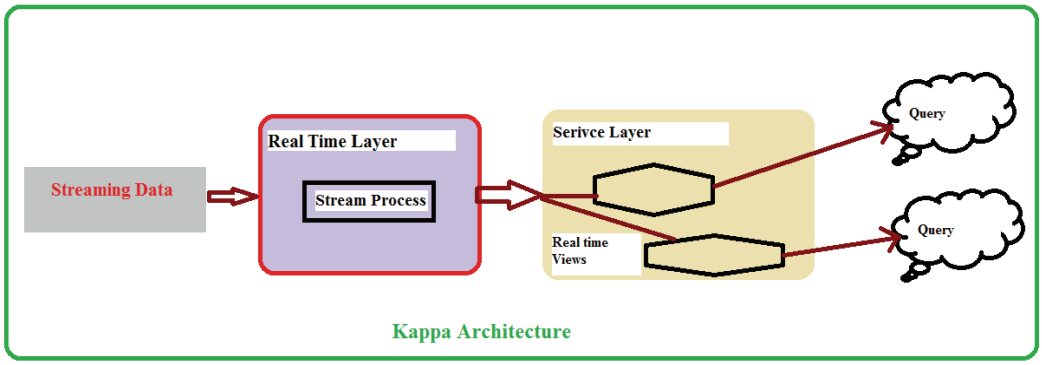


Fig 3.1 Kappa Architecture

Kappa Architecture is a streaming-first data processing model designed to handle both real-time and historical data using a single stream processing pipeline. Unlike Lambda Architecture, which separates batch and stream processing, Kappa simplifies the system by relying only on stream processing for all workloads.

Kappa Architecture Components Using Kafka:

1. Data Ingestion (Event Logs):
    a. Kafka acts as the central log storage where all raw data events are written.
    b. Producers (databases, microservices, IoT devices, logs) continuously stream data into Kafka topics.
    c. Kafka topics retain all data, allowing reprocessing if needed.
2. Stream Processing (Transformation & Analytics):
    a. Kafka Streams (Java-based processing):
        i. Performs real-time transformations, filtering, aggregations, and joins.
        ii. Enables stateful processing using local state stores.
        iii. Ideal for developer-controlled, complex event processing.
    b. ksqlDB (SQL-based processing):
        i. Enables SQL queries on streaming data without Java coding.
        ii. Supports continuous queries, aggregations, windowing, and materialized views.
        iii. Ideal for real-time dashboards, alerts, and quick transformations.
3. Serving Layer (Consumption & Storage):
    a. Applications (APIs, microservices, analytics systems) consume processed data in real-time.
    b. Materialized views in ksqlDB store the latest transformed results for fast querying.

c. Kafka topics can be connected to external databases, search engines, or BI tools.

3.2 Kafka Architecture Perspective

The architecture of Kafka Streaming SQL involves several core components:

1. **ksqlDB Server:** A stateless, distributed server that executes SQL queries on Kafka streams. It leverages Kafka Streams, an underlying library for stream processing.
2. **Kafka Topics:** Data is ingested and processed via Kafka topics, which are the fundamental unit of storage in Kafka.
3. **Streams:** Continuous data flows (stream of records) that are processed in real-time.
4. **Tables:** A table represents the latest state of a stream, which is continuously updated as new data arrives.
5. **ksqlDB Client:** This is the interface (CLI or REST API) through which users can submit SQL queries, define streams/tables, and interact with the ksqlDB server.

3.2.1 Architecture of Kafka Streams for Confluent Platform

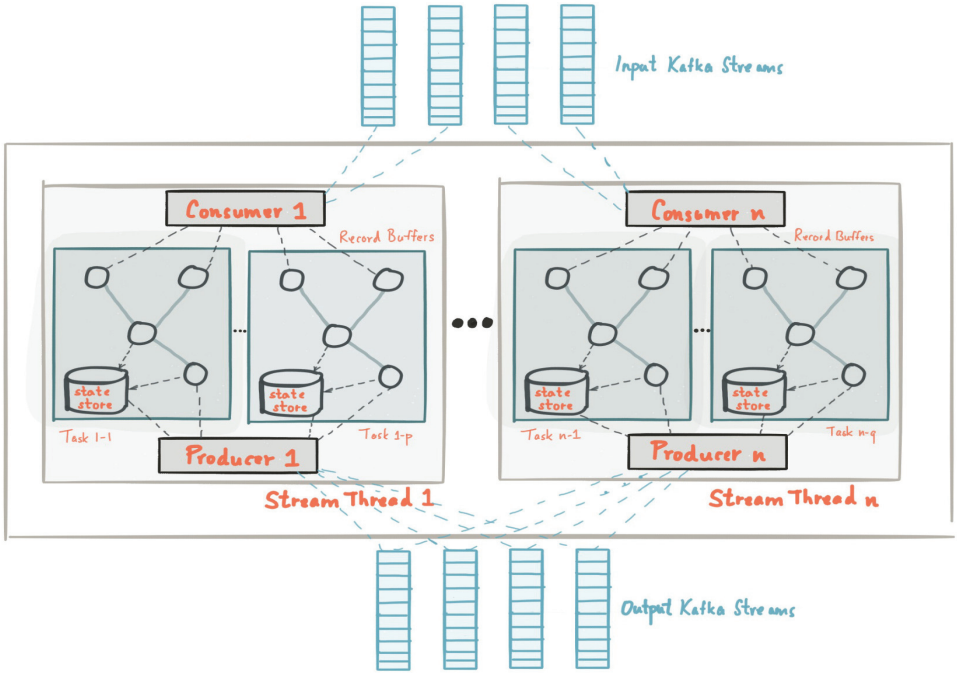


Fig 3.2.1.1 Kafka Streams Architecture for Confluent Platform

Kafka Streams simplifies application development by building on the Apache Kafka producer and consumer APIs, and leveraging the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity.

Here is the anatomy of an application that uses the Kafka Streams API. It provides a logical view of a Kafka Streams application that contains multiple stream threads, that each contain multiple stream tasks.

Processor Topology:

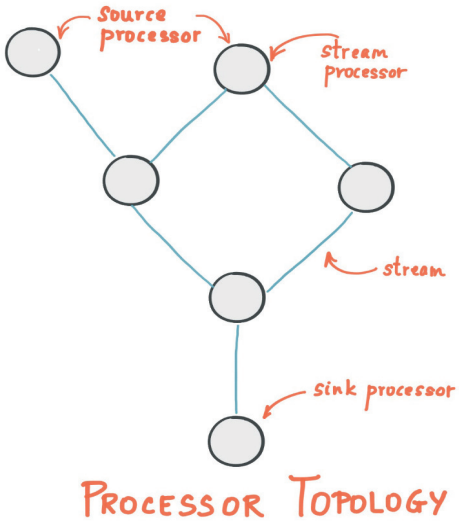
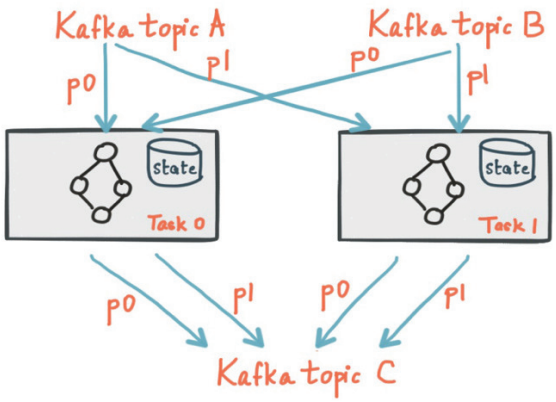


Fig 3.2.1.2 Processor Topology

A processor topology defines the stream processing computational logic for your application, i.e., how input data is transformed into output data. A topology is a graph of stream processors (nodes) that are connected by streams (edges) or shared state stores. There are two special processors in the topology:

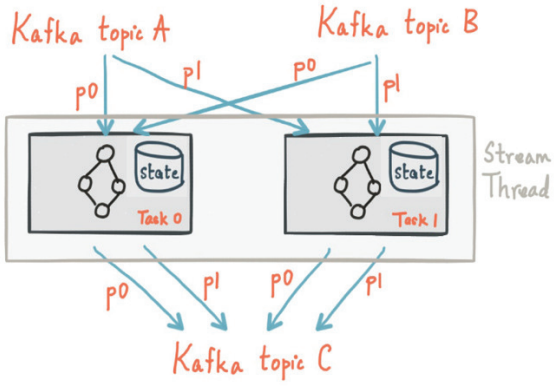
1. **Source Processor:** A source processor is a special type of stream processor that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forward them to its down-stream processors.
2. **Sink Processor:** A sink processor is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.

Parallelism model:



Two tasks each assigned with one partition of the input streams.

Fig 3.2.1.3 Sub-topologies (also called sub-graphs)



One stream thread running two stream tasks.

Fig 3.2.1.4 Threading model

Above self-explanatory figures shows how the Kafka Topic partitions are fed to its downstream Processor Topology (each maintaining its own state stores) & in-turn received by another Topic.

3.2.2 Architecture of ksqlDB for Confluent Platform

ksqlDB has these main components:

1. **ksqlDB engine:** processes SQL statements and queries
2. **REST interface:** enables client access to the engine
3. **ksqlDB CLI:** console that provides a command-line interface (CLI) to the engine
4. **ksqlDB UI:** enables developing ksqlDB applications in Confluent Control Center and Confluent Cloud

The REST server interface enables communicating with the ksqlDB engine from the CLI, Confluent Control Center, or from any other REST client. ksqlDB queries are executed by the ksqlDB server, which pushes the processing to Kafka Streams, allowing real-time stream processing across distributed systems.

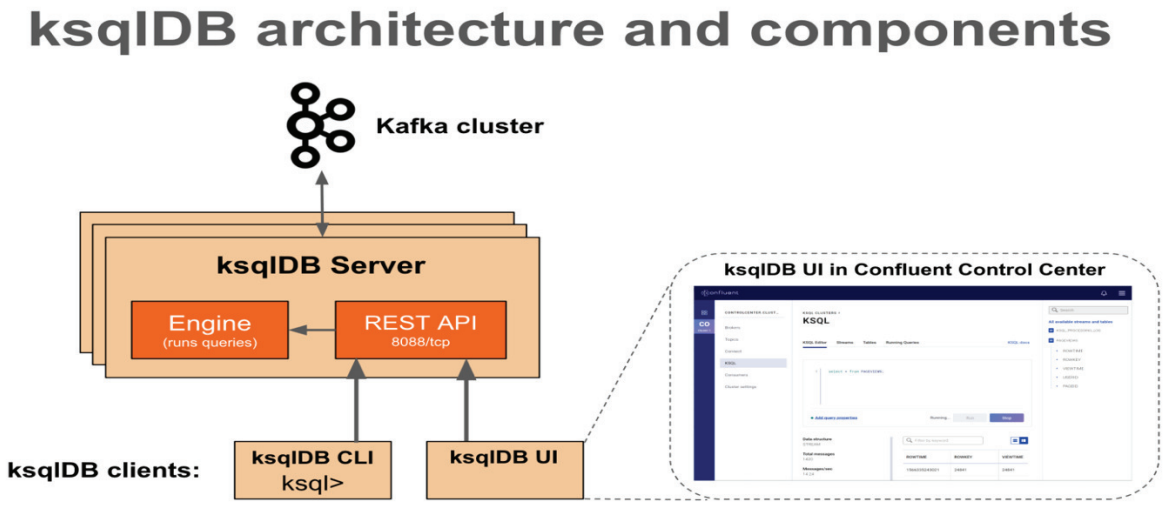


Fig 3.2.2.1 Architecture of ksqlDB

There are two modes of ksqlDB:

1. Interactive mode
2. Headless mode

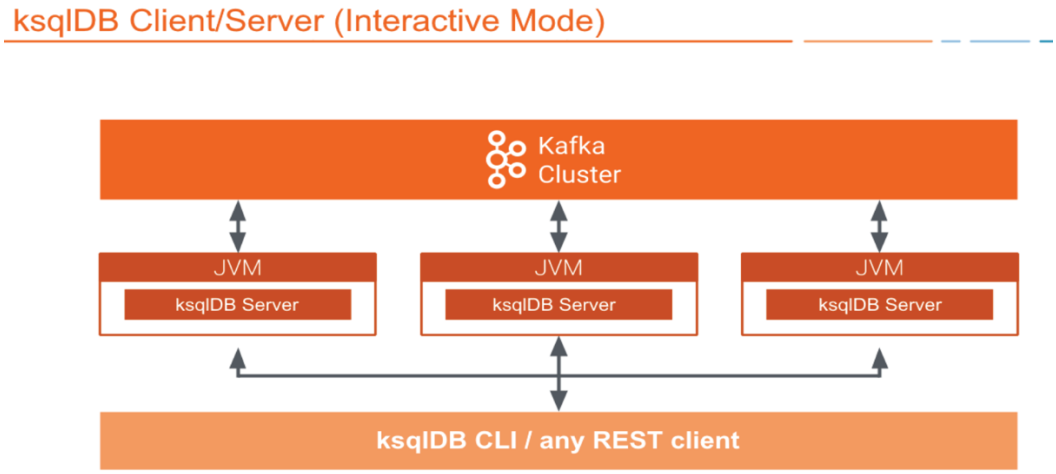


Fig 3.2.2.2 Interactive Mode

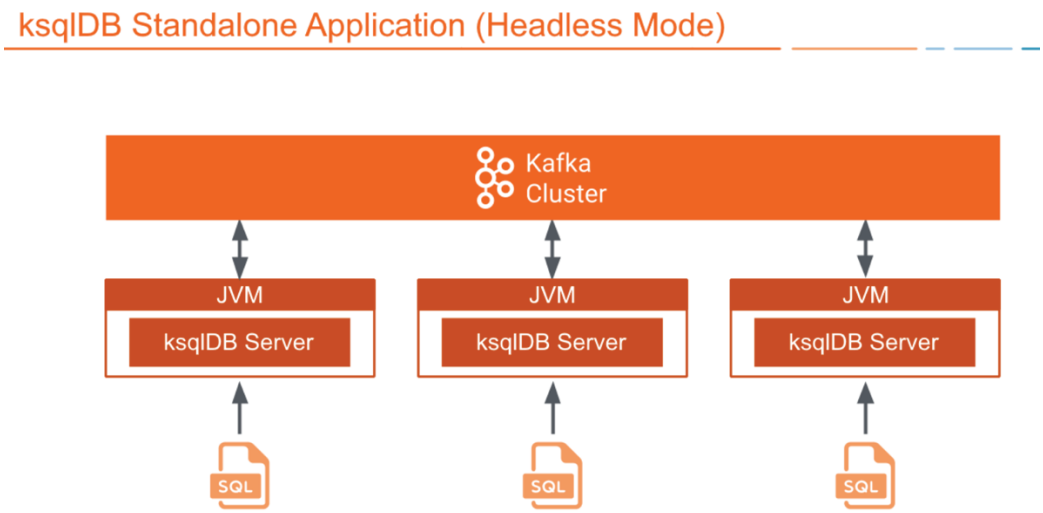


Fig 3.2.2.3 Headless Mode

Feature	Interactive Mode	Headless Mode
Execution Type	Manual (via CLI/API)	Automatic (via script)
Persistence	No (queries lost on restart)	Yes (queries run automatically)
Best For	Development, testing, debugging	Production workloads
Flexibility	High (modifications on the fly)	Low (requires restart for changes)
Startup Process	Requires manual query execution	Executes predefined queries

Fig 3.2.2.4 Interactive vs Headless modes of ksqlDB

For most real-world applications, developers start in Interactive Mode to experiment, then deploy in Headless Mode for production.

4 Use Cases Where Kafka Streaming SQL is Suitable

1. **Real-Time Data Processing**: Kafka Streams and KSQL excel at processing and analysing data in real-time. This makes them ideal for applications like real-time analytics, fraud detection, and monitoring.
2. **Event-Driven Architectures**: Streaming SQL can be used to react to changes in data as events. Use cases include order processing, inventory updates, and system alerts.
3. **Real-Time ETL Pipelines**: Kafka Streams and KSQL can be used to extract, transform, and load (ETL) data in real time between systems. For example, transforming raw logs into structured data or aggregating metrics on the fly.
4. **Aggregation and Windowing**: Real-time aggregations (such as count, sum, and average) over data streams can be performed. Windowing features allow for operations like "last-minute purchases" or "5-minute average temperature" to be calculated efficiently.
5. **IoT Applications**: In Internet of Things (IoT) scenarios, where streams of sensor data need to be processed in real-time, Kafka Streams and KSQL are well-suited for operations like anomaly detection or aggregating data over time windows.
6. **Data Integration with External Systems**: Kafka Connect and KSQL can be used to pull data from external sources like databases, apply transformations in-flight, and send

the results to Kafka topics, or push them to other data systems like databases or data lakes.

4.1 Use Case for Demo

Let's consider a real-time fraud detection system on financial transaction for demo.
Fraud Detection: Set up a stream of transaction data where you detect fraudulent activity based on transaction volume or velocity. Use ksqlDB to filter and alert on high-risk transactions. (Transaction amount > 3000 are considered fraudulent in this example)

5 Language/Programming/Implementation Details

We used python scripts to run producer & consumer part. Screenshots for the same are attached with required output.
Producer script: is used to send the transactions to Kafka network.
Consumer script: is used to capture the fraud alerts.

Update below fields in kafka/etc/server.properties file:
listeners=PLAINTEXT://localhost:9092
advertised.listeners=PLAINTEXT://localhost:9092

```
santhbha@SANTHBHA-M-DJG2 kafka %
santhbha@SANTHBHA-M-DJG2 kafka % confluent local services start
The local commands are intended for a single-node development environment only, NOT for production usage.
As of Confluent Platform 8.0, Java 8 will no longer be supported.

Using CONFLUENT_CURRENT: /var/folders/6c/77pyy9515q38xq7by48bcj400000gn/T/confluent.093356
ZooKeeper is [UP]
Kafka is [UP]
Schema Registry is [UP]
Kafka REST is [UP]
Connect is [UP]
ksqlDB Server is [UP]
Control Center is [UP]
santhbha@SANTHBHA-M-DJG2 kafka %
```

Fig 5.1 Starting Confluent services

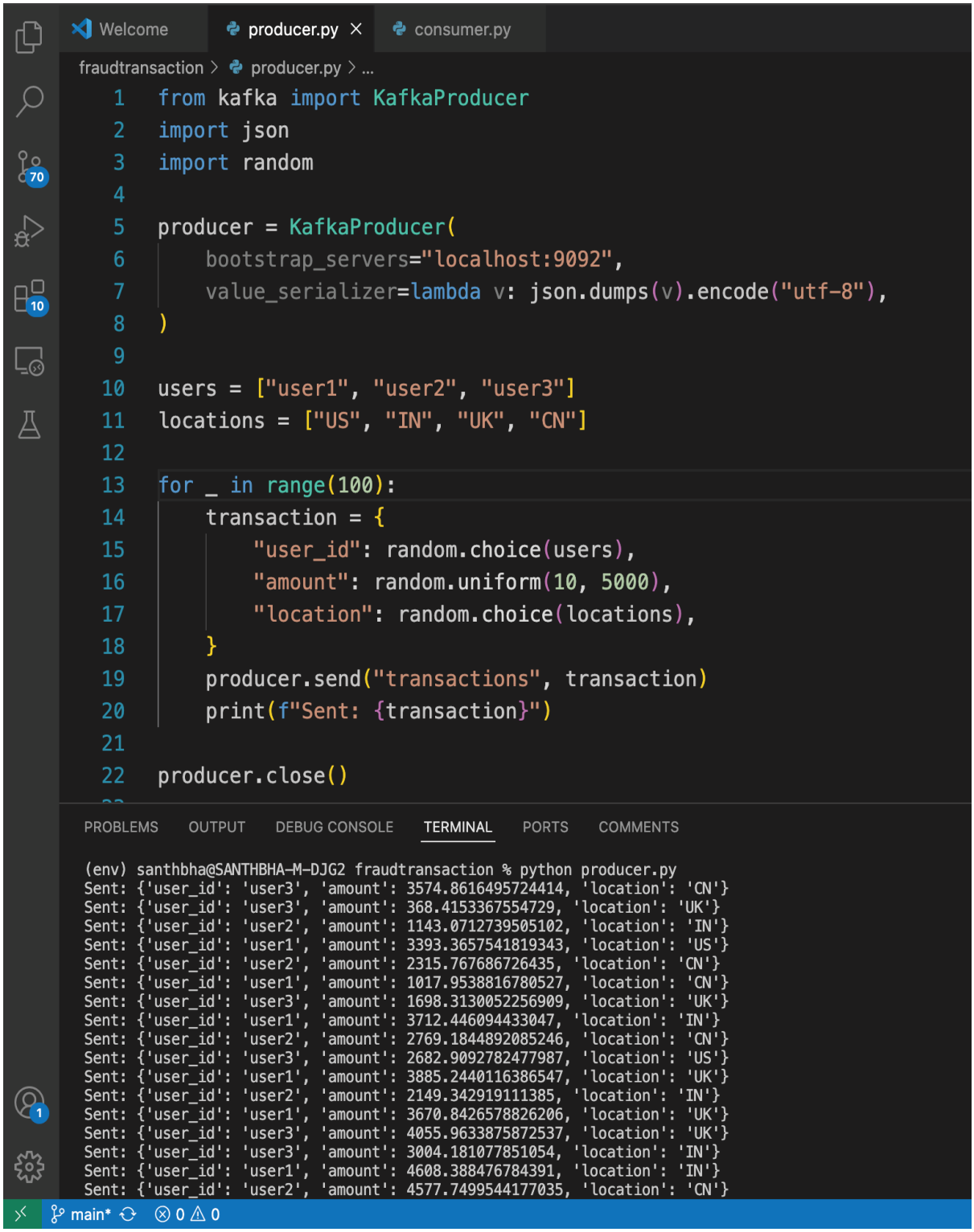


Fig 5.2 producer.py scripts & output

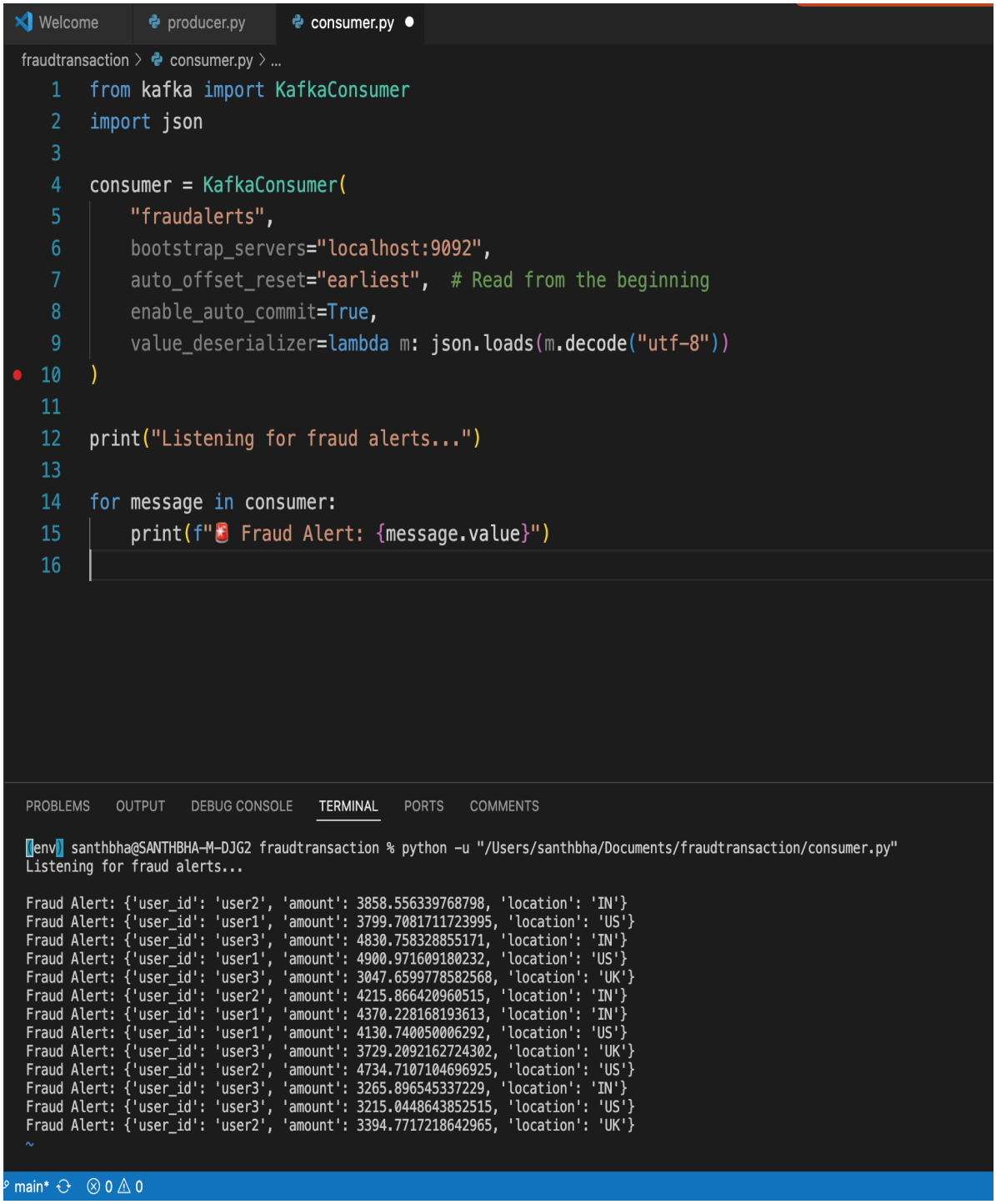


Fig 5.3 consumer.py scripts & output

6 Use case if any you are considering to illustrate as demo

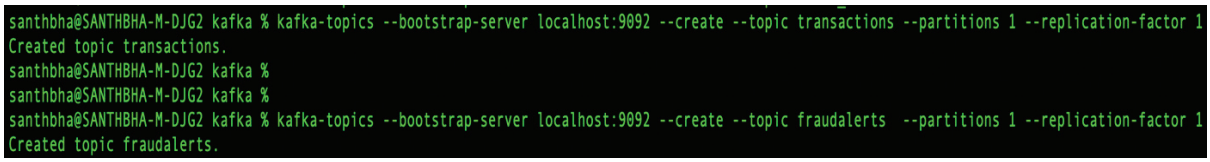


Fig 6.1 Creating Kafka Topics

```
ksql> CREATE STREAM transactions_streams (
>     user_id STRING,
>     amount DOUBLE,
>     location STRING
>) WITH (
>     KAFKA_TOPIC='transactions',
>     VALUE_FORMAT='JSON'
>);

 Message
----------------
 Stream created
----------------
ksql>
ksql>
ksql> CREATE STREAM fraudalert_stream AS
>SELECT user_id, amount, location
>FROM transactions_streams
>WHERE amount > 3000;

 Message
---------------------------------------------------
 Created query with ID CSAS_FRAUDALERT_STREAM_47
```

Fig 6.2 Creating Kafka Streams

```
santhbha@SANTHBHA-M-DJG2 kafka % kafka-topics --bootstrap-server localhost:9092 --list  | grep -v -e con -e __ -e _d -e _sc -e de
FRAUDALERT_STREAM
fraud_alerts
fraudalert_stream
fraudalerts
transactions
santhbha@SANTHBHA-M-DJG2 kafka %
santhbha@SANTHBHA-M-DJG2 kafka %
```

Fig 6.3 Displaying created Kafka Topics & Streams

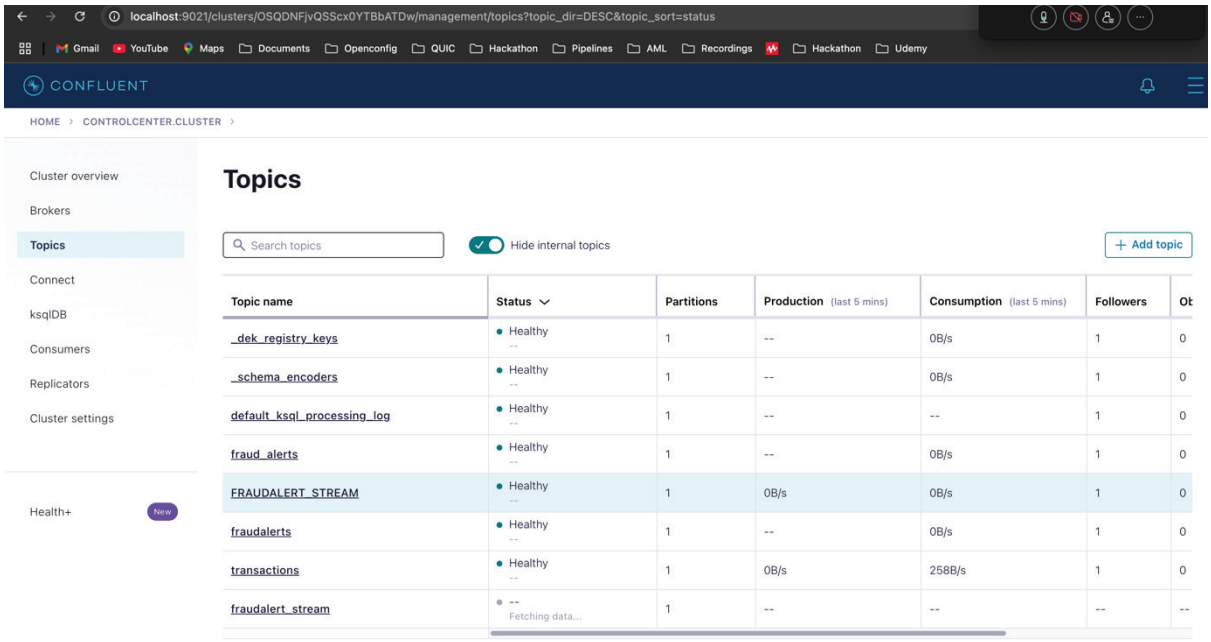


Fig 6.4 Kafka Topics & Streams in Confluent GUI

```
santhbha@SANTHBHA-M-DJG2 kafka % kafka-console-consumer --topic transactions --bootstrap-server loc
alhost:9092 --from-beginning --property print.key=true --property print.value=true

null    {"user_id": "user1", "amount": 612.6076277137471, "location": "CN"}
null    {"user_id": "user3", "amount": 31.589399153602486, "location": "CN"}
null    {"user_id": "user2", "amount": 3858.556339768798, "location": "CN"}
null    {"user_id": "user1", "amount": 2819.5072062358804, "location": "UK"}
null    {"user_id": "user2", "amount": 3799.7081711723995, "location": "IN"}
null    {"user_id": "user1", "amount": 4830.758328855171, "location": "UK"}
null    {"user_id": "user3", "amount": 1748.0579725075818, "location": "IN"}
null    {"user_id": "user2", "amount": 2908.8176845042676, "location": "CN"}
null    {"user_id": "user2", "amount": 4900.971609180232, "location": "IN"}
null    {"user_id": "user2", "amount": 3047.6599778582568, "location": "UK"}
null    {"user_id": "user2", "amount": 2868.5756995833904, "location": "US"}
null    {"user_id": "user1", "amount": 295.4469140649998, "location": "UK"}
null    {"user_id": "user1", "amount": 1798.1154040651136, "location": "US"}
null    {"user_id": "user3", "amount": 2838.3748587445507, "location": "IN"}
null    {"user_id": "user1", "amount": 1588.9450475344172, "location": "US"}
null    {"user_id": "user1", "amount": 4215.866420960515, "location": "UK"}
null    {"user_id": "user3", "amount": 4370.228168193613, "location": "CN"}
null    {"user_id": "user3", "amount": 2091.409961971488, "location": "CN"}
null    {"user_id": "user2", "amount": 1894.7732116025823, "location": "UK"}
null    {"user_id": "user2", "amount": 145.18374083320774, "location": "UK"}
null    {"user_id": "user2", "amount": 4130.740050006292, "location": "IN"}
null    {"user_id": "user1", "amount": 3729.2092162724302, "location": "IN"}
null    {"user_id": "user2", "amount": 4734.7107104696925, "location": "CN"}
null    {"user_id": "user1", "amount": 2691.8083593827128, "location": "CN"}
null    {"user_id": "user3", "amount": 2857.3047687083313, "location": "UK"}
null    {"user_id": "user1", "amount": 3265.896545337229, "location": "IN"}
null    {"user_id": "user3", "amount": 1278.3447276482416, "location": "CN"}
null    {"user_id": "user3", "amount": 2284.9327606982697, "location": "IN"}
null    {"user_id": "user2", "amount": 3215.0448643852515, "location": "IN"}
```

Fig 6.5 Kafka transactions Topic contents

```
ksql> SELECT * FROM transactions_streams EMIT CHANGES;
+----------------------------------+----------------------------------+----------------------------------+
|USER_ID                           |AMOUNT                            |LOCATION                          |
+----------------------------------+----------------------------------+----------------------------------+
|user1                             |612.6076277137471                 |CN                                |
|user3                             |31.589399153602486                |CN                                |
|user2                             |3858.556339768798                 |CN                                |
|user1                             |2819.5072062358804                |UK                                |
|user2                             |3799.7081711723995                |IN                                |
|user1                             |4830.758328855171                 |UK                                |
|user3                             |1748.0579725075818                |IN                                |
|user2                             |2908.8176845042676                |CN                                |
|user2                             |4900.971609180232                 |IN                                |
|user2                             |3047.6599778582568                |UK                                |
|user2                             |2868.5756995833904                |US                                |
|user1                             |295.4469140649998                 |UK                                |
|user1                             |1798.1154040651136                |US                                |
|user3                             |2838.3748587445507                |IN                                |
|user1                             |1588.9450475344172                |US                                |
|user1                             |4215.866420960515                 |UK                                |
|user3                             |4370.228168193613                 |CN                                |
|user3                             |2091.409961971488                 |CN                                |
|user2                             |1894.7732116025823                |UK                                |
|user2                             |145.18374083320774                |UK                                |
|user2                             |4130.740050006292                 |IN                                |
|user1                             |3729.2092162724302                |IN                                |
|user2                             |4734.7107104696925                |CN                                |
|user1                             |2691.8083593827128                |CN                                |
|user3                             |2857.3047687083313                |UK                                |
|user1                             |3265.896545337229                 |IN                                |
|user3                             |1278.3447276482416                |CN                                |
|user3                             |2284.9327606982697                |IN                                |
|user2                             |3215.0448643852515                |IN                                |
```

Fig 6.6 Displaying transactions_streams from producer.py in ksqlDB

```
ksql> SELECT * FROM fraudalert_stream;
+---------------------------------+---------------------------------+---------------------------------+
|USER_ID                          |AMOUNT                           |LOCATION                         |
+---------------------------------+---------------------------------+---------------------------------+
|user2                            |3858.556339768798                |CN                               |
|user2                            |3799.7081711723995               |IN                               |
|user1                            |4830.758328855171                |UK                               |
|user2                            |4900.971609180232                |IN                               |
|user2                            |3047.6599778582568               |UK                               |
|user1                            |4215.866420960515                |UK                               |
|user3                            |4370.228168193613                |CN                               |
|user2                            |4130.740050006292                |IN                               |
|user1                            |3729.2092162724302               |IN                               |
|user2                            |4734.7107104696925               |CN                               |
|user1                            |3265.896545337229                |IN                               |
|user2                            |3215.0448643852515               |IN                               |
|user3                            |3394.7717218642965               |UK                               |
|user3                            |3966.6473881841357               |CN                               |
|user2                            |4042.2937255225766               |IN                               |
|user1                            |3957.212583489849                |CN                               |
|user3                            |4694.3467551038875               |US                               |
|user1                            |4965.198434352817                |UK                               |
|user1                            |4570.63245738717                 |CN                               |
|user3                            |4986.8380923545255               |US                               |
|user2                            |4997.339666109605                |IN                               |
|user1                            |3233.8438721259936               |IN                               |
|user2                            |4700.376418908743                |UK                               |
|user2                            |3925.705580169537                |US                               |
|user2                            |3393.9161903035656               |CN                               |
|user1                            |4867.249573519803                |IN                               |
```

Fig 6.7 Processed fraudalent_stream in ksqlDB

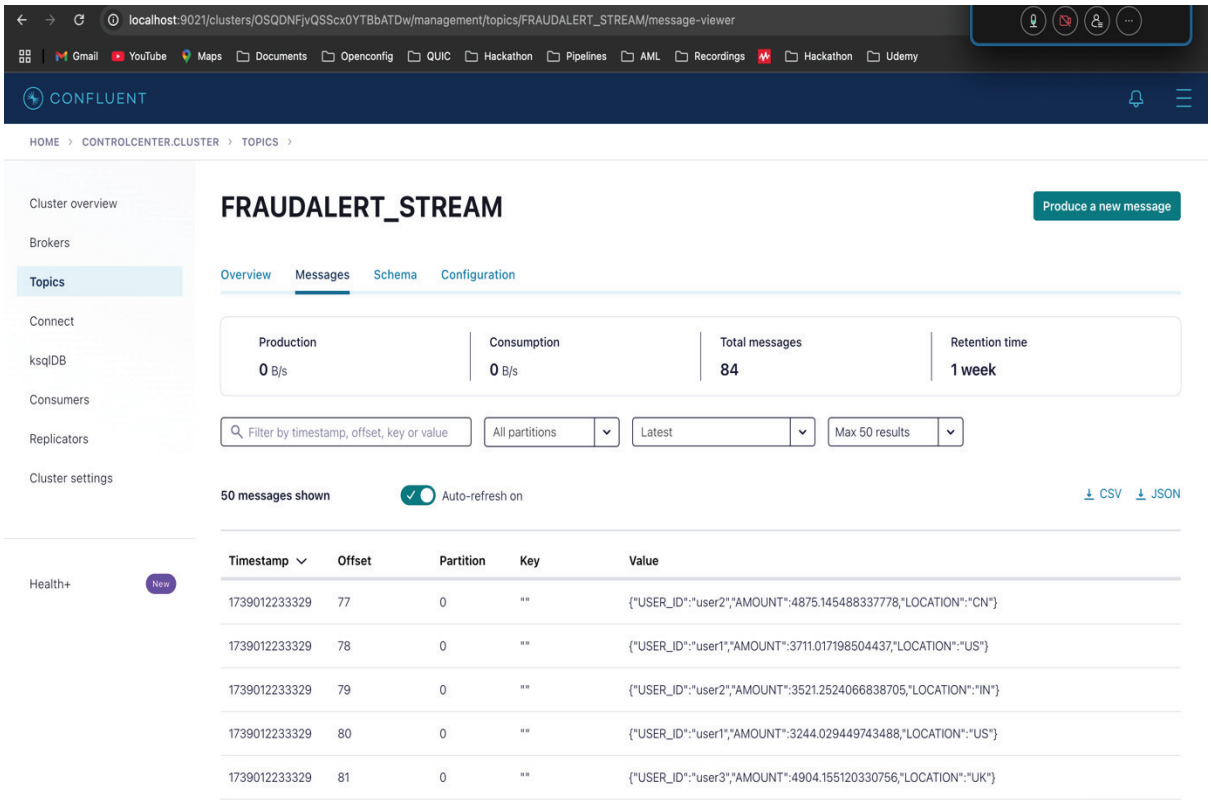


Fig 6.8 Processed Fraudulent Streams in Confluent GUI

7 Explicitly mention scripts/commands to be used as part of streaming integration with other platforms if applicable

Not applicable. All the required screenshots are attached in previous section itself.

8 Conclusions and inferences

Kafka Streams and ksqlDB extend full potential of Apache Kafka's capabilities by enabling real-time stream processing, making data transformation, enrichment, and analytics seamless without relying on external systems. Kafka Streams provides a powerful, stateful Java API for building scalable stream-processing applications, while ksqlDB simplifies stream processing with an easy-to-use SQL interface. Both technologies align perfectly with Kappa Architecture, eliminating batch-processing complexity and supporting event-driven applications. Their ability to handle continuous data processing, fault tolerance, and real-time analytics makes them ideal for modern use cases such as fraud detection, monitoring, IoT, and microservices. By leveraging Kafka Streams and ksqlDB, organizations can move towards a fully event-driven architecture, unlocking real-time insights and crucial step towards decision-making at scale.