Mid-Term Assignment
DADS 6005 : Data Streaming and Real-Time Analytics

Chayaphon Sornthananon, 6610422007

Semester 1/2024

Lecturer : Asst.Prof Ekarat Rattagan

Table of Content

Requirements

Data streaming and Realtime analytics System Requirement

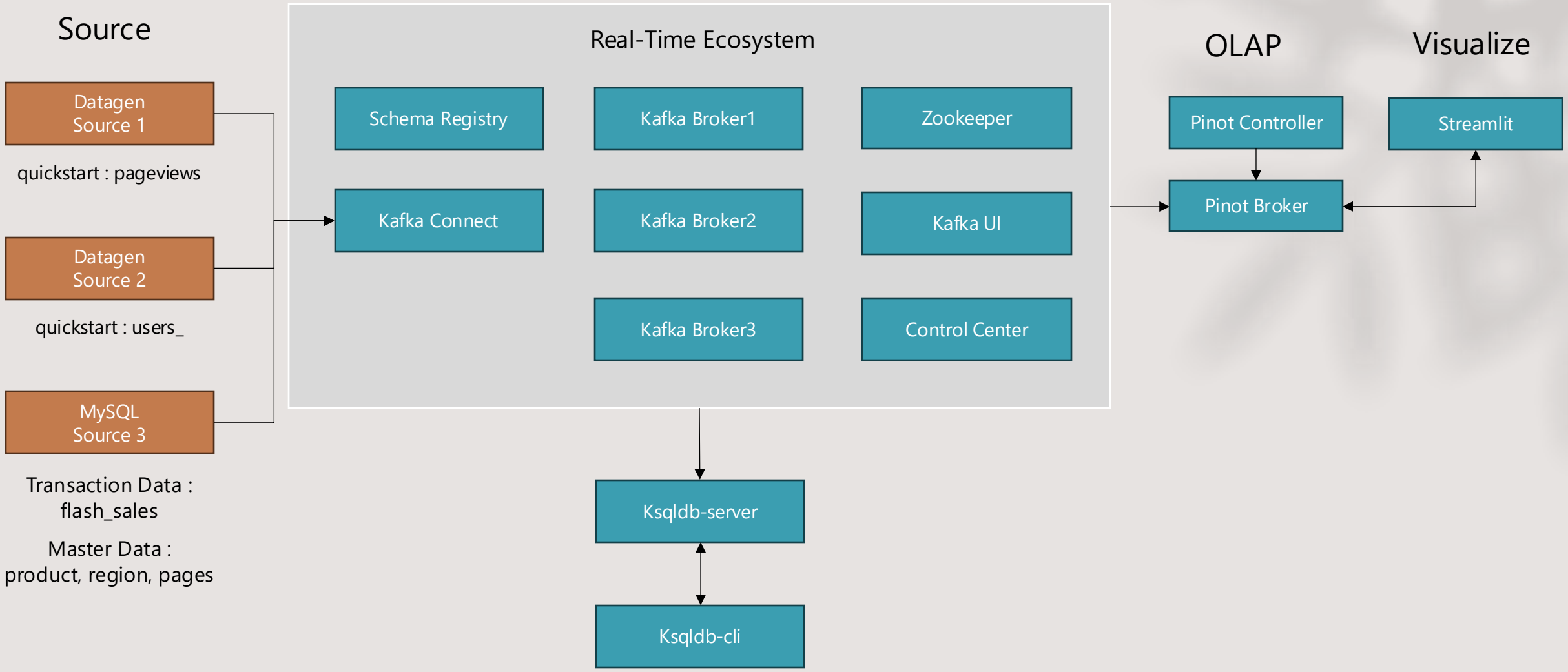
a. Data source 3 sources [1]
    i. Source 1 : PageView (stream datagen) (topic1)
    ii. Source 2 : Users_ (stream datagen) (topic2)
    iii. Source 3 : Your design (relational database) (topic3)

b. Kafka system [2]
    i. 5 partitions
    ii. 3 brokers
    iii. 8 topics
    iv. Schema Register
    v. Kafka connect

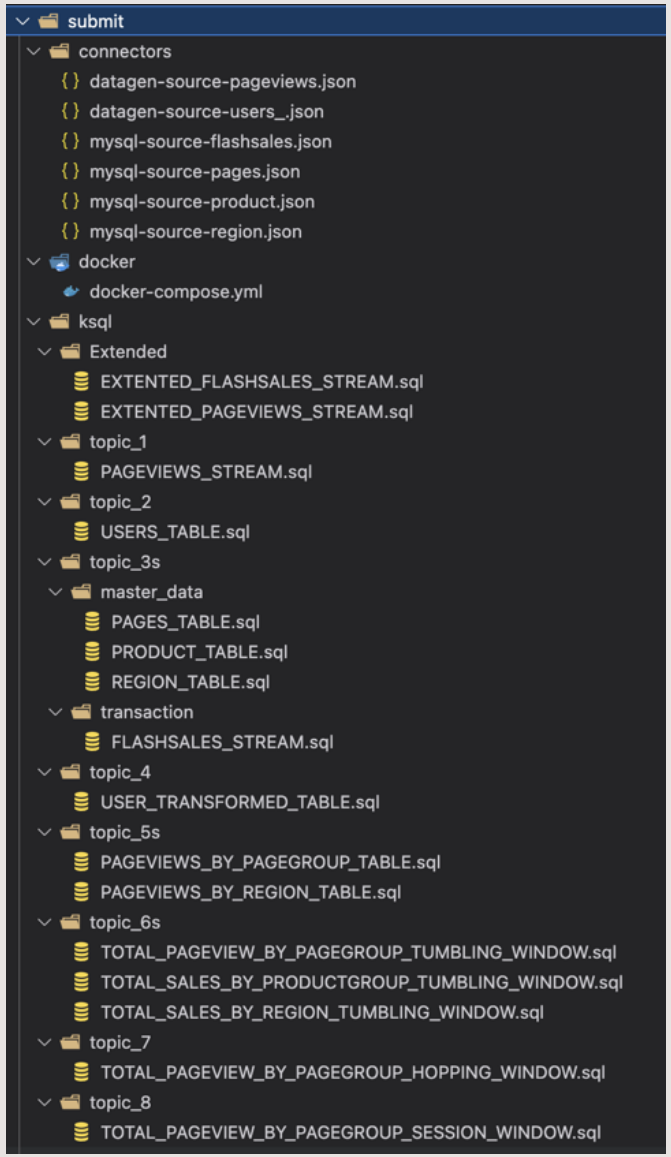
c. ksqlDB operation [3]
    i. Clean or transform data (topic4)
    ii. Aggregation (join + group by) (topic5)
    iii. Windows
        1. Tumbling (topic6)
        2. Hopping (topic7)
        3. Session (topic8)
    iv. Testing the correctness [4]

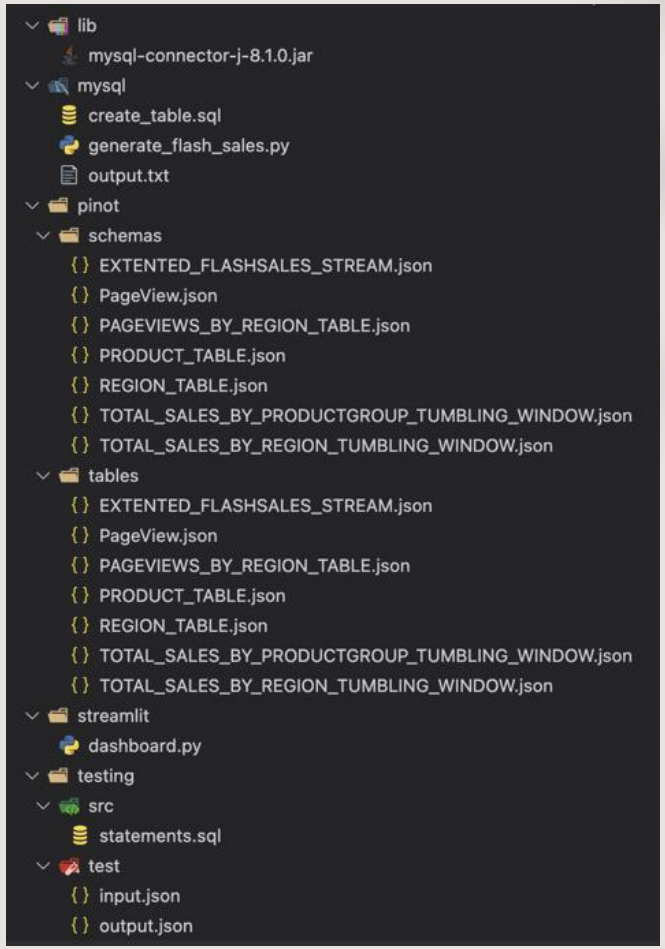
d. Apache Pinot [5,6]
    i. At least 3 queries
e. Dashboard [7]
    i. At least 4 panels
    ii. Bonus: Interactive dashboard

System Diagram



Code Reference

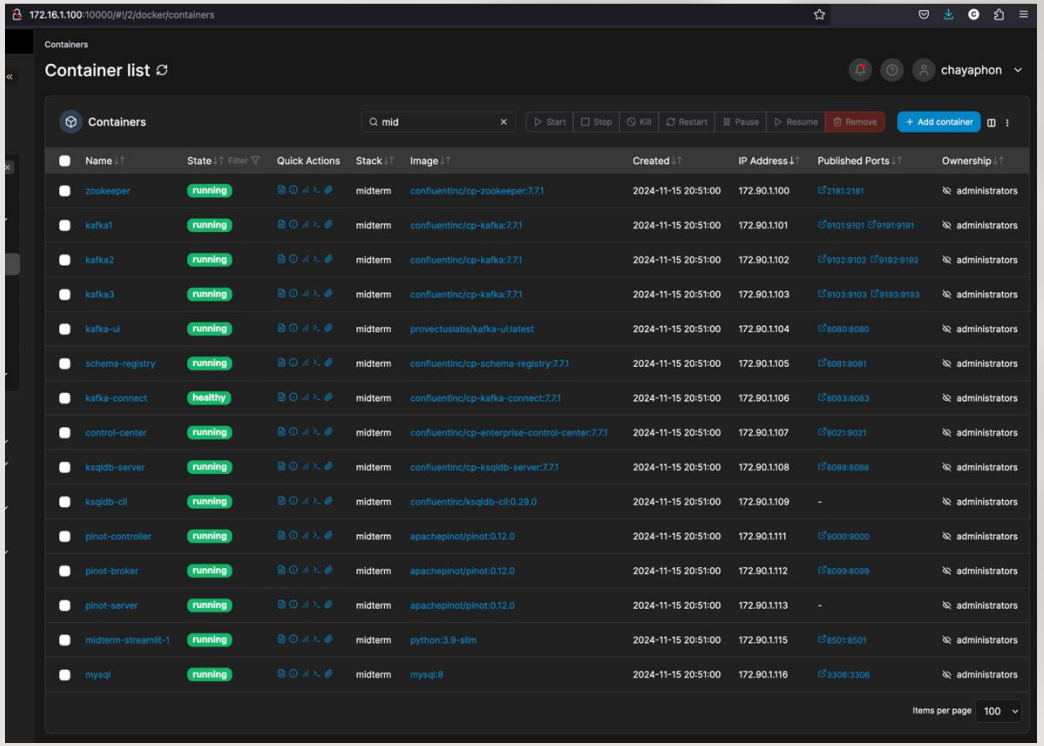




Docker

In this project we use docker technology to help us running each services.

This screenshot show all containers run for this project.



What is Docker

Docker is an **open-source containerization platform** that allows developers to package applications into **containers**.

Containers are lightweight, portable, and isolated environments that include everything needed to run an application, such as code, libraries, and dependencies.

Docker ensures that applications behave consistently across development, testing, and production environments.

Why Use Docker

Docker provides a solution to many challenges in modern application development and deployment:

1.Consistency: Docker ensures that the application runs the same in all environments, eliminating wording "it works on my machine" issues.

2.Portability: Docker containers are platform-independent and can run on any system with Docker installed, including local machines, servers, and cloud platforms.

3.Efficiency: Containers share the host operating system, making them faster and more lightweight compared to traditional virtual machines.

4.Scalability: Applications can scale easily by spinning up or down multiple containers as needed.

5.Ease of Deployment: Docker simplifies the deployment process by packaging applications and dependencies into one container.

How Docker Works

1.Docker file (Input):

1. Developers write a dockerfile or docker-compose, which includes instructions to build a Docker image. The image contains the application code, dependencies, and runtime.
2. Docker-compose help to pack all containers within one stack help for dependencies check on running and easy to maintain.

2.Docker Engine (Process):

1. The Docker Engine reads the dockerfile to create an **image**, which serves as a blueprint for creating containers.
2. When a container is launched from the image, it runs as an isolated, standalone instance.

3.Running Containers (Output):

1. The containerized application is executed in an environment isolated from the host system, ensuring stability and consistency.
2. Docker also enables networking, data sharing, and scaling through its runtime engine.

Role and Responsibility for each services

1. Zookeeper
- **Purpose**: Manages metadata and coordination for Kafka brokers.
- **Responsibilities**:
    - Handles configuration management and leader election for Kafka.
    - Keeps track of Kafka broker states.

•**Interactions**: Essential for Kafka brokers (kafka1, kafka2, kafka3) to function, as it maintains metadata about partitions and replicas.

2. Kafka Brokers (kafka1, kafka2, kafka3)
- **Purpose**: These brokers form the Kafka cluster.
- **Responsibilities**:
    - Store and serve data streams.
    - Distribute partitioned data across brokers for scalability.
    - Ensure fault tolerance with data replication.
- **Interactions**:
    - Communicate with Zookeeper to get metadata.
    - Work together to replicate data and distribute partitions.
    - Provide the backbone for other services like schema-registry, Kafka-connect, and Ksqldb-server.

3. Schema Registry
- **Purpose**: Centralizes schema management for Kafka topics.
- **Responsibilities**:
    - Stores schemas for Avro, JSON, and Protobuf data.
    - Ensures compatibility between producers and consumers.

•**Interactions**:
    - Works with Kafka brokers to store schemas in specific topics.
    - Enables serialization and deserialization for Kafka-connect and Ksqldb-server.

4. Kafka Connect
- **Purpose**: Facilitates data ingestion and export from Kafka.
- **Responsibilities**:
    - Connects external systems (e.g., databases, files) with Kafka.
    - Provides scalable and fault-tolerant connectors.
- **Interactions**:
    - Uses schema-registry for data serialization.
    - Relies on Kafka brokers for publishing and consuming messages.
    - Interacts with databases like MySQL to pull or push data.

5. KSQLDB (Server and CLI)
- **Purpose**: Enables real-time stream processing using SQL-like queries.
- **Responsibilities**:
    - Processes and transforms streaming data.
    - Creates derived topics or materialized views.
- **Interactions**:
    - Relies on Kafka brokers for streaming data.
    - Uses schema-registry for managing schemas.
    - Works with Kafka-connect for external data integration.

6. Kafka UI
- **Purpose**: Web interface to manage and monitor Kafka clusters.
- **Responsibilities**:
    - Visualizes Kafka topics, brokers, and consumer groups.
    - Simplifies topic management and monitoring.
- **Interactions**:
    - Connects to Kafka brokers for real-time data visualization.
    - Uses schema-registry and kafka-connect for enhanced insights.

7. Control Center
- **Purpose**: Monitoring and managing the Confluent platform.
- **Responsibilities**:
    - Provides a UI to monitor Kafka cluster health, latency, and throughput.
    - Supports managing topics, schemas, and connectors.
- **Interactions**:
    - Connects to Kafka brokers, schema-registry, Kafka-connect, and Ksqldb-server to consolidate management.

8. Pinot Components (Controller, Broker, Server)
- **Purpose**: Real-time OLAP (Online Analytical Processing) system for querying Kafka data.
- **Responsibilities**:
    - **Controller**: Manages Pinot cluster configuration.
    - **Broker**: Handles query routing.
    - **Server**: Stores and serves data segments.
- **Interactions**:
    - Pulls data from Kafka topics for real-time analysis.
    - Uses zookeeper for metadata and coordination.

9. MySQL
- **Purpose**: Relational database used for persistent storage.
- **Responsibilities**:
    - Acts as a data source or sink for Kafka via Kafka-connect.
- **Interactions**:
    - Integrates with Kafka for data ingestion/export.

10. Streamlit Dashboard
- **Purpose**: Frontend for visualizing analytics results.
- **Responsibilities**:
    - Displays real-time analytics and insights from Kafka data.
- **Interactions**:
    - Queries data from Pinot (pinot-broker).
    - Presents data to users through an intuitive UI.

**PS. Kafka UI and Control Connect servce as the similar purpose, but control center is paid version (can trial some period) and it also included ui for ksql operation.*

a.Data Source

Datagen Source 1

quickstart : pageviews

Source 1 : Using Kafka's built-in **data generation connectors** to simulate real-time data with quick start pageviews

Datagen Source 2

quickstart : users_

Source 2 : Using Kafka's built-in **data generation connectors** to simulate real-time data with quick start user_

MySQL Source 3

Transaction Data : flash_sales

Master Data : product, region, pages

Source 3 : Using **mysql source connectors** to periodical load data according to poll interval.

There is 1 transactional event data

- Flash sales is the transaction record for each product sales event.

There are 3 master data

- Product : describe the product name and product category (group) from product_id
- Region : describe the region name from region_id
- Pages : describe the pages name and page category (group) from page_id

For flash sales event I use python code to continue iterate simulate sales transaction within 1 secs per iteration.

a.Data Source

What : Source is where data was created and then we push or poll data to kafka topic in real-time.

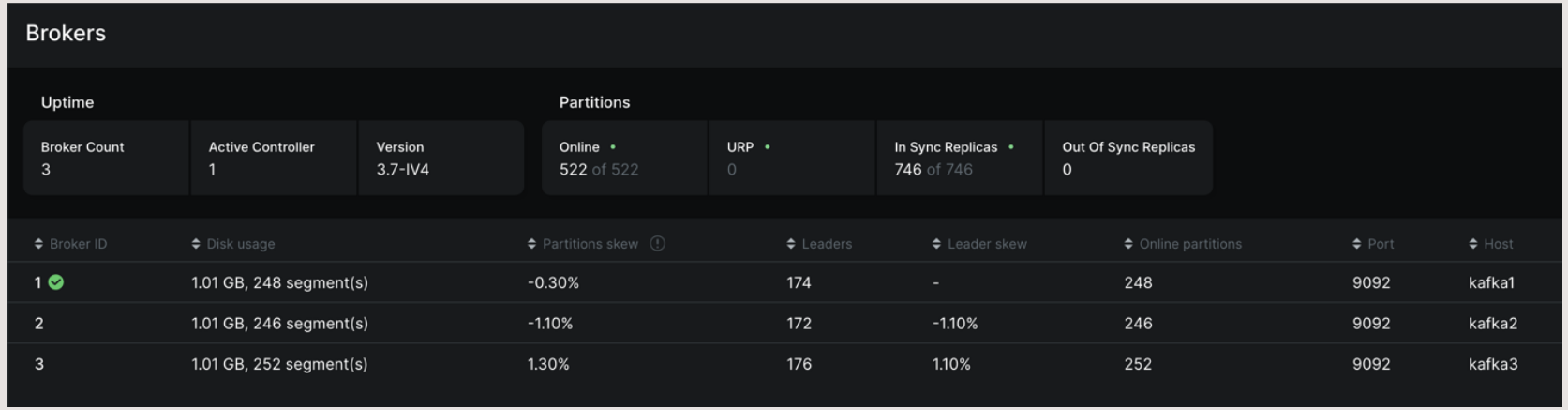
Why : The data can either be push by producer or poll from source driectly by kafka connect.

How :
Input
- Source 1 & 2 **quick start data generation**
- Source 3 Mysql Relational Database

Process
- Source 1 & 2 Kafka's built-in **quick start data generation** connector gererate and serialize data and operate on the **push** mechanism to continuously simulate and stream real-time data to Kafka topic 1 & 2 accordingly.
- Source 3 use **mysql source connectors** operates on the **poll** mechanism with periodical pulling data according to poll interval configuration. (in this example topic3 for flash_sales table) and extra 3 topics for master data

Output
- Serialize data located in kafka partition under its accociate topic.

b.Kafka System : Kafka Broker



Brokers

Uptime

Broker Count	Active Controller	Version
3	1	3.7-IV4

Partitions

Online	URP	In Sync Replicas	Out Of Sync Replicas
522 of 522	0	746 of 746	0

Broker ID	Disk usage	Partitions skew	Leaders	Leader skew	Online partitions	Port	Host
1	1.01 GB, 248 segment(s)	-0.30%	174	-	248	9092	kafka1
2	1.01 GB, 246 segment(s)	-1.10%	172	-1.10%	246	9092	kafka2
3	1.01 GB, 252 segment(s)	1.30%	176	1.10%	252	9092	kafka3

Accordin to the requirements we set-up 3 brokers

What : Broker is the main server where Kafka operate and store each message under its associated topic.

Why : Multiple brokers will help on load balancing (scalable) and fault-tolerant with replica (copy) data to other brokers if config replica > 1 in topic.

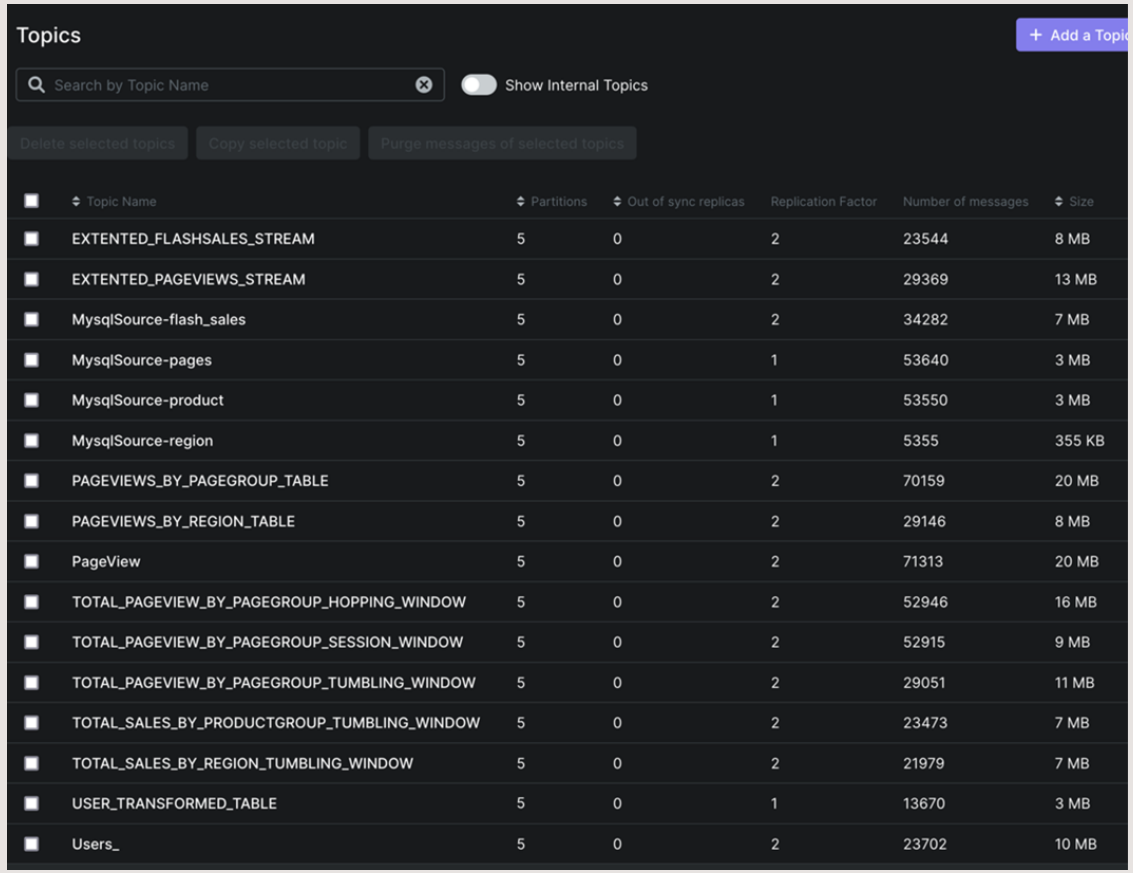
How: Leader and Follower Brokers

For each partition, **one broker is assigned as the leader**, while the others are followers. The leader broker handles all client requests (reads and writes) for its partition, while follower brokers replicate the data to stay synchronized.

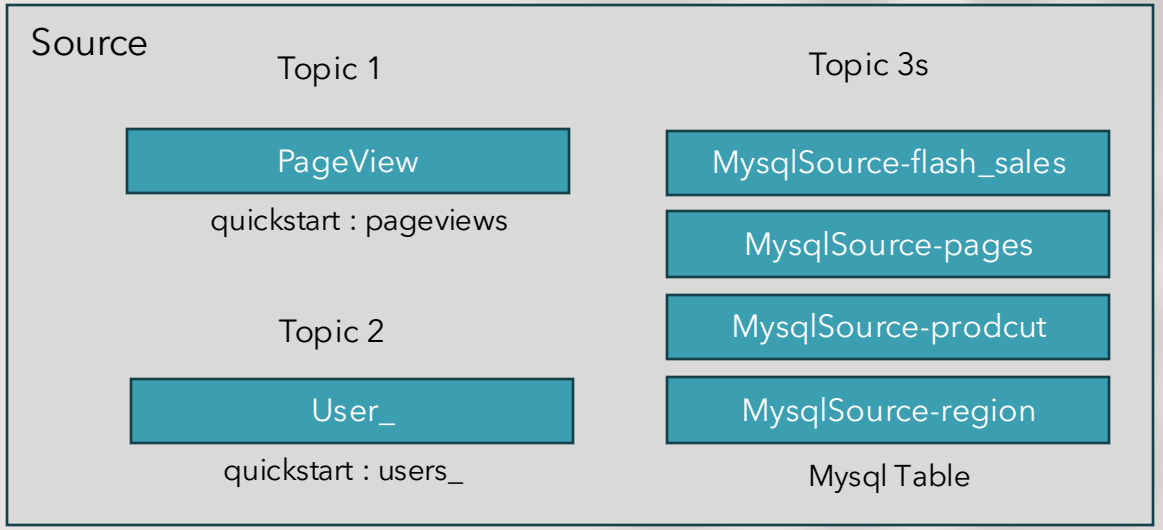
- If a leader broker fails, one of the followers takes over as the new leader, ensuring uninterrupted data access.

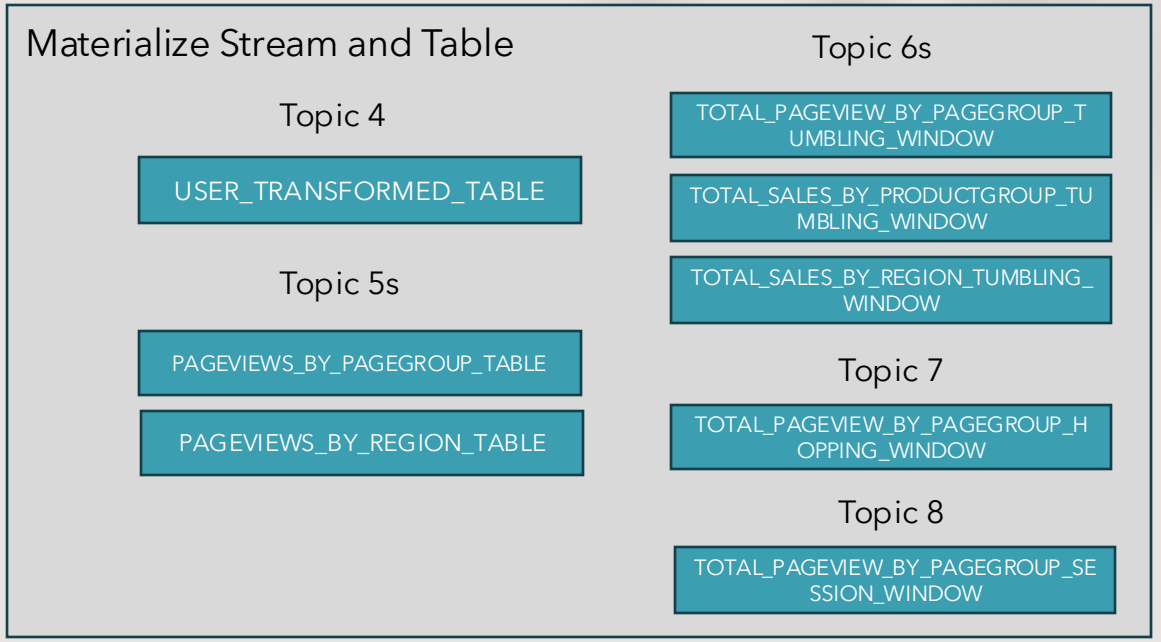
b.Kafka System : Kafka Topics and Partition

I have create 16 topics, each topic has 5 partitions, and important topic (event base) has 2 replica and none importance (master data) has 1 replica.



Topic Name	Partitions	Out of sync replicas	Replication Factor	Number of messages	Size
EXTENTED_FLASHSALES_STREAM	5	0	2	23544	8 MB
EXTENTED_PAGEVIEWS_STREAM	5	0	2	29369	13 MB
MysqlSource-flash_sales	5	0	2	34282	7 MB
MysqlSource-pages	5	0	1	53640	3 MB
MysqlSource-product	5	0	1	53550	3 MB
MysqlSource-region	5	0	1	5355	355 KB
PAGEVIEWS_BY_PAGEGROUP_TABLE	5	0	2	70159	20 MB
PAGEVIEWS_BY_REGION_TABLE	5	0	2	29146	8 MB
PageView	5	0	2	71313	20 MB
TOTAL_PAGEVIEW_BY_PAGEGROUP_HOPPING_WINDOW	5	0	2	52946	16 MB
TOTAL_PAGEVIEW_BY_PAGEGROUP_SESSION_WINDOW	5	0	2	52915	9 MB
TOTAL_PAGEVIEW_BY_PAGEGROUP_TUMBLING_WINDOW	5	0	2	29051	11 MB
TOTAL_SALES_BY_PRODUCTGROUP_TUMBLING_WINDOW	5	0	2	23473	7 MB
TOTAL_SALES_BY_REGION_TUMBLING_WINDOW	5	0	2	21979	7 MB
USER_TRANSFORMED_TABLE	5	0	1	13670	3 MB
Users_	5	0	2	23702	10 MB





b.Kafka System : Kafka Topics and Partition

What :

Topic is a category or feed name where records (messages) are sent by producers and read by consumers.
• It's similar to a queue in other messaging systems but with key differences that allow Kafka to handle distributed and durable message storage.
Partition is a subset of a topic and represents a unit of parallelism.
• Each topic is divided into one or more partitions, which are distributed across Kafka brokers.

Why :

Kafka topics and partitions are essential for Kafka's scalability, performance, and reliability.
Topic:
- **Purpose**: Organizes data logically, ensuring related events are grouped together.
- **Importance**: Makes it easy for producers and consumers to handle specific data streams. For example:
    - Topic PageView: Handles user view page on website events.
    - Topic User_: Handles user registration events.

Partition:
- **Purpose**: Distributes data across brokers for scalability and increase performance of read and write parallelly.
- **Importance**:
    - **Scalability**: Partitions enable Kafka to handle large volumes of data by distributing it across multiple brokers.
    - **Parallelism**: Multiple consumers can process partitions independently, improving performance.
    - **Fault Tolerance**: Kafka replicates partitions across brokers to ensure data durability and availability.

How :

Input (Data) : Producer writes data to a topic -> Partition assignment
Process :
- Each partition stores messages in an append-only log with a unique offset (sequence number) for each message.
- Kafka replicates each partition across multiple brokers for fault tolerance.
- Consumers subscribe to a topic and pull messages from partitions.

Output :
- Messages are stored in the topic partitions and replicated for durability.
- Each consumer reads from one or more partitions, enabling distributed processing of the data stream.
- Kafka retains messages for a configurable period (e.g., 7 days), allowing consumers to replay or reprocess data if needed.

b.Kafka System : Additional Explanation

In this setup, I have configured **3 brokers** to provide **load balancing** and **fault tolerance**.

1. Partitions

- I have created topic with **5 partitions**. This partitioning improves **performance** by spreading the data across multiple brokers, which allows parallel read and write operations.
- By having 5 partitions, the data storage is split into 5 segments, which can be distributed across different brokers. This division increases throughput for both reading and writing data, as different clients can access separate partitions simultaneously.

2. Replicas

- **Replication** is a mechanism to achieve fault tolerance in Kafka. Each partition has a specified number of replicas to ensure data availablity in case of broker failures.

  In my setup:

  - **Event transactions (important topics)** have set **replication factor of 2**. This means each partition in this topic has two copies on different brokers. If one broker fails, another broker with a replica can still serve the data, ensuring reliability.
  - **Master data (less critical topics)** have a **replication factor of 1**. This means there is only one copy of each partition, reducing storage requirements at the cost of lower redundancy.
- **Disadvantages of Replicas:**
  While replicas provide data redundancy, having more replicas consumes additional disk space and can impact performance, as each write operation must be synchronized across all replicas, increasing the I/O load.

3. Leader and Follower Brokers

For each partition, **one broker is assigned as the leader**, while the others are followers. The leader broker handles all client requests (reads and writes) for its partition, while follower brokers replicate the data to stay synchronized.

- If a leader broker fails, one of the followers takes over as the new leader, ensuring uninterrupted data access.

4. Cleanup Policy and Retention

In this setup, each topic is configured with a **delete** cleanup policy to manage data storage effectively. This policy removes data that exceeds the defined retention period.

• **Master Data**: The retention period for master data (less critical topics) is set to **1 day**. This helps reduce storage usage by retaining only the most recent data, as older records are less likely to be needed.

• **Transaction Data**: For transaction data (critical topics), the retention period is set to **7 days**. This ensures that important data remains available for a longer duration to support critical operations before being deleted.

b.Kafka System : Schema-Registry

What : **Schema Registry** is a central component that manages and stores the schemas used for serializing and deserializing data in Kafka. It is commonly used with formats like **Avro**, **Protobuf**, or **JSON Schema**.

- It provides a repository to store schemas, enabling producers and consumers to agree on the data structure (schema) for communication.
- Each schema is versioned, allowing you to track changes and ensure compatibility between producers and consumers.

Why : The Schema Registry plays a critical role in maintaining **data integrity**, **compatibility**, and **evolution** in distributed systems.

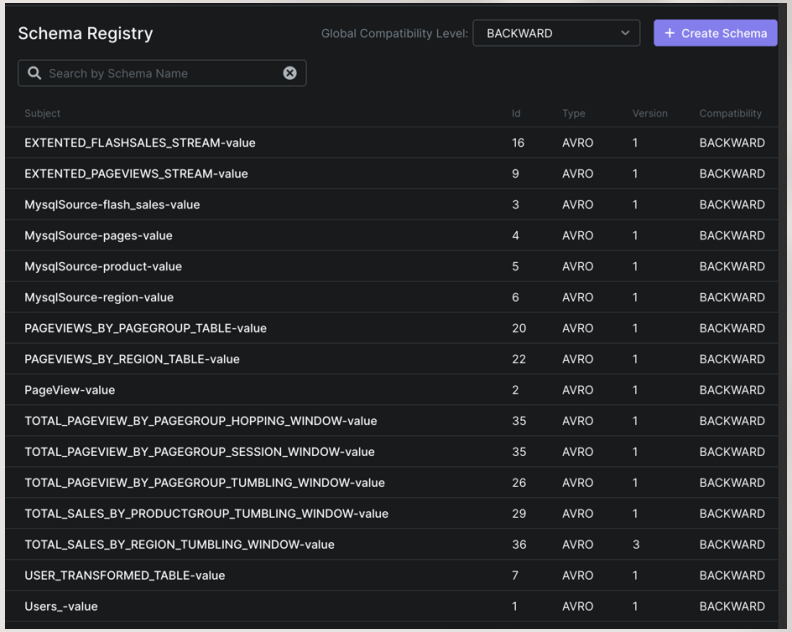
1. **Ensures Data Consistency**:
- Producers must write data that adheres to a defined schema.
- Consumers can safely read and deserialize data, knowing the schema structure.
2. **Manages Schema Evolution**:
- As schemas change over time (e.g., adding fields), the Schema Registry ensures changes are compatible (backward or forward compatibility).
3. **Simplifies Consumer-Producer Communication**:
- Producers and consumers don't need to embed schemas in every message. Instead, they refer to a schema stored in the Schema Registry using a schema ID.
4. **Reduces Data Overhead**:
- Instead of embedding the full schema with each message, only the schema ID is sent, reducing message size and bandwidth usage.
5. **Prevents Errors**:
- Validates messages against the schema before they are sent to Kafka, avoiding corrupt or malformed data.

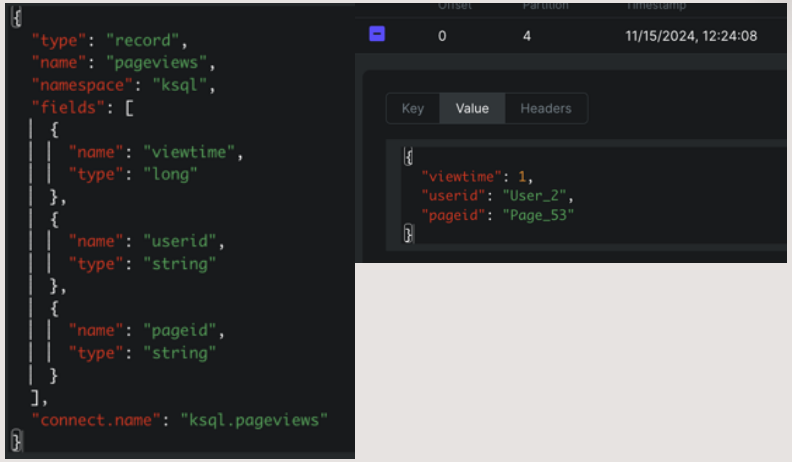
How :

Input:
- A producer sends page view event with the schema { "viewtime": "long", "userid": "string", "pageid":"string"} to Kafka.
- This schema is registered with the Schema Registry as Schema ID 2.

Process:
- The producer serializes the event using Schema ID 2 and sends it to a Kafka topic.
- The consumer reads the event and retrieves Schema ID 2 from the Schema Registry to deserialize it.

Output:
- The consumer processes the deserialized message: { "viewtime": 1, "userid ": "User_2", "pageid":"Page_53"}.





b.Kafka System : Kafka Connect

What : Kafka Connect acts as a framework to integrate Kafka brokers with external systems. It uses connectors that can be configured as **source connectors** to ingest data into Kafka or **sink connectors** to export data from Kafka to external systems. Kafka Connect typically uses a **polling mechanism** for data movement.

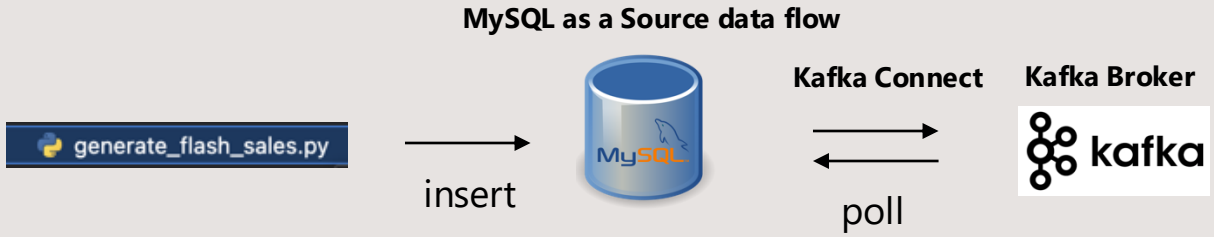
Why : To store and consume data in real-time with low latency we use Kafka services, with external source we require connector who act as a messenger between broker and source/sink.

How : Input and process : (in my configuration)

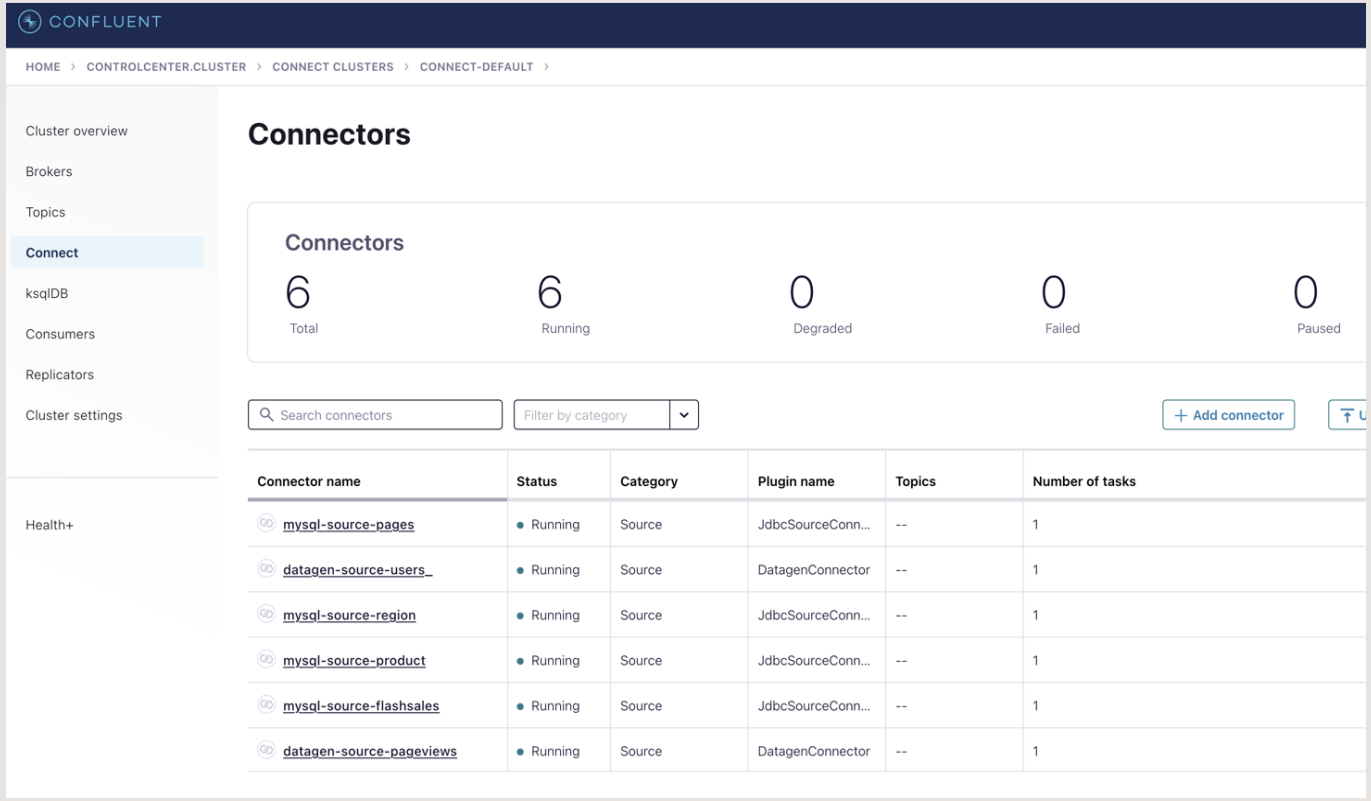
- **Datagen as Source (pageviews and users):**
  - Using Kafka's built-in **data generation connectors** to simulate real-time data for pageviews and users.
  - These connectors generate data at configurable time intervals and send it directly to Kafka topics.

- **MySQL as a Source**
  - **Flash Sales Transactions:**
    - The flash sales transaction data is set up with an **incremental mode** using the sale_id column. This mode checks for new entries based on the sale_id. If a new sale_id is detected, it is loaded into the corresponding Kafka topic (MysqlSource-flash_sales).
    - The **poll interval** for this load is configured at **1000 milliseconds** (1 second), meaning Kafka will check MySQL every second for new sales transactions.
  - **Master Data (product, pages, and region):**
    - For static master data, I've set for **bulk mode**. In bulk mode, Kafka performs a full load of the data at each poll interval, rather than checking for individual updates.
    - The **poll interval** is set to **60000 milliseconds** (1 minute), meaning every minute, Kafka fetches the latest data from MySQL in bulk and updates the topic. In real-life master data will be flush and load on daily interval.

Output : Data arrived serialized and store in Kafka partition under its associated topic, ready for consumer to query.



b.Kafka System : Connectors for data source



mysql-source-flashsales
Incremental load data from mysql table flash_sales every 1 secs

"table.whitelist": "flash_sales",
"mode": "incrementing",
"incrementing.column.name": "sale_id",
"topic.prefix": "MysqlSource-",
"poll.interval.ms": "1000",

mysql-source-pages
Bulk load data from mysql table pages every 60 secs

"table.whitelist": "pages",
"mode": "bulk",
"topic.prefix": "MysqlSource-",
"poll.interval.ms": "60000",

datagen-source-pageviews
- simulate page view event every 1 secs

datagen-source-users_
- simulate users_ registeration event every 3 secs

mysql-source-product
Bulk load data from mysql table pages every 60 secs

"table.whitelist": "product",
"mode": "bulk",
"topic.prefix": "MysqlSource-",
"poll.interval.ms": "60000",

mysql-source-region
Bulk load data from mysql table region every 60 secs

table.whitelist": "region",
"mode": "bulk",
"topic.prefix": "MysqlSource-",
"poll.interval.ms": "60000",

c. KsqlDB

What : Ksqldb is an event streaming database built on top of Apache Kafka. It allows us to work with real-time data streams using SQL-like queries. With ksqlDB, we can process, transform, and analyze streaming data in a declarative and user-friendly way, without needing to write complex code.

Why : Materialize View

- **Streams** are continuous, appending new rows as events occur, suitable for real-time processing of events. (e.g., 2,3.1, e)
- **Tables** capture the current state, providing the latest data per primary key, which is ideal for static or master data that updates over time. (e.g., 1,3,5)

Windowed Aggregations Table:

- The setup also includes **windowed tables** (e.g., 6,7,8), which aggregate data over specific time windows. These tables are useful for time-based analyses, like tracking session activities or events within a particular timeframe (tumbling , hopping and session windows)

c. KsqlDB

- **Stream as a Materialized View**

  A **stream** in ksqlDB represents a continuous, append-only flow of data. While it is not inherently a "materialized view", when query a stream, ksqlDB processes the data in real-time and can materialize the results for a query.

  Characteristics:
  1. **Unbounded Data**: Streams represent an unending flow of events.
  2. **Not State-Based**: Streams are focused on capturing raw, unaggregated data.
  3. **Transient Materialization**: When you run a query on a stream, ksqlDB processes the events in real-time and temporarily materializes the results for the duration of the query.
  4. **No Persistent State**: Streams do not store state or maintain results over time unless explicitly written to a Kafka topic or derived into a table.

- **Table as a Materialized View**

  A **table** in ksqlDB is explicitly a materialized view. It represents the latest state of data, where incoming records are aggregated or updated based on a key.

  Characteristics:
  1. **State-Based**: Tables maintain a persistent, stateful view of the latest data.
  2. **Materialized by Default**: Tables store their state (e.g., aggregations, counts) in an internal **RocksDB store** for fast lookups and updates.
  3. **Supports Updates**: Tables can update their state when new data arrives, similar to how a database table works.
  4. **Queryable State**: Tables allow you to execute point-in-time queries against the current state.

How :
Input: A producer pushes data into a Kafka topic, which serves as the source for ksqlDB queries.
Process: ksqlDB queries process data from the topic and may create materialized views for stateful operations. If the query outputs transformed, cleaned, or aggregated data, ksqlDB writes the results to a **new Kafka topic** for further use.
Output: Queries produce results either as **materialized views** (internally stored for efficient lookups) or as **new Kafka topics**, depending on the query type and configuration.

c. KsqlDB : Clean or transform data (topic4)

```
CREATE TABLE USER_TRANSFORMED_TABLE AS
SELECT
userid,
TIMESTAMPTOSTRING(registertime, 'yyyy-MM-dd HH:mm:ss') AS registertime,
gender,
regionid,
interests[1] AS interests_1,
interests[2] AS interests_2,
contactinfo['phone'] AS phone,
contactinfo['city'] AS city,
contactinfo['state'] AS state,
contactinfo['zipcode'] AS zipcode
FROM USERS_TABLE
```

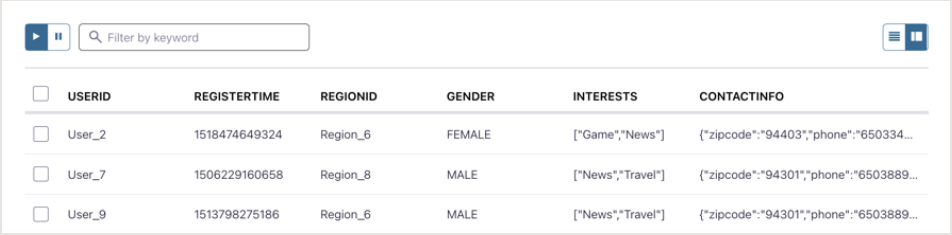
input

Output

What : Sometime our data came with unmeaningful or not ready to use we may need to perform cleam or transformation data like joining key to master table or covert value to be easy for visualize.

Process :
This materialize process transformation from USER_TABLE and put to USER_TRANSFORMED_TABLE

- Tranform registertime from unixtimestamp to string format with human readble.
- Tranform Interests with array of 2 key value pair to each individual column interest1 and interest2.
- Tranform contactinfo by mapping it separate columns



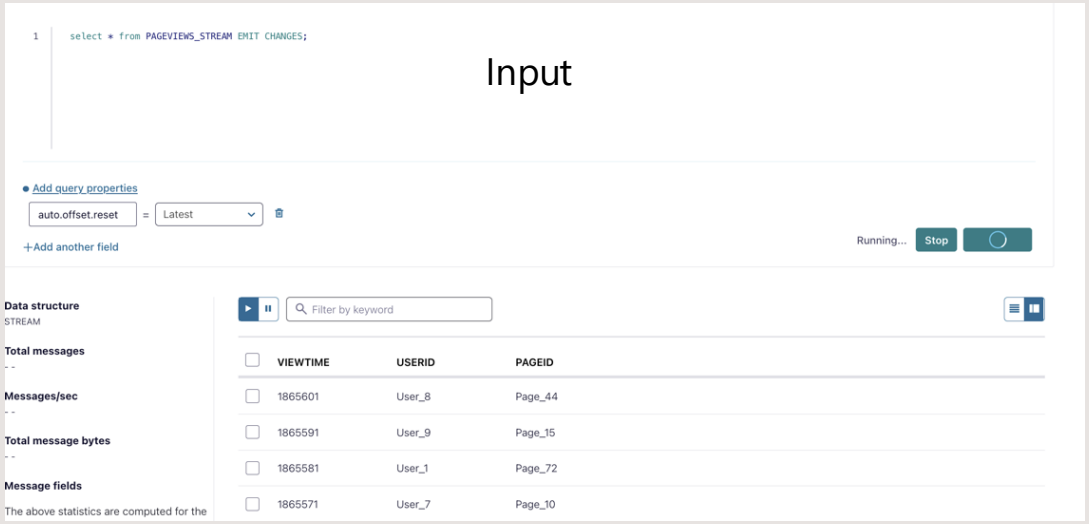
c. KsqlDB : Aggregation (join + group by) (topic5)

```
CREATE TABLE PAGEVIEWS_BY_PAGEGROUP_TABLE AS
SELECT
pt.page_group as key,
CAST(pt.page_group AS STRING) AS page_group,
COUNT(*) AS count_view
FROM pageviews_stream pvs
LEFT JOIN pages_table pt ON pt.page_id = pvs.pageid
GROUP BY pt.page_group
EMIT CHANGES;
```

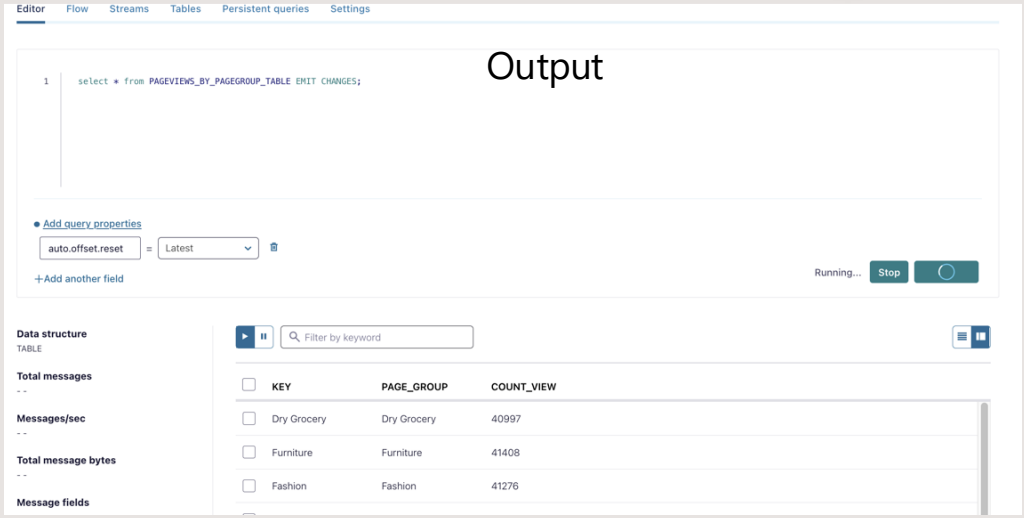
What : Sometime our data came with lower granularity (transaction level) and we may need to perform aggregation to be easy for understand big picture of data and describable as per each dimension.

Process :
This materialize process join pageviews_stream with pages_table to get page_group attribute and aggregate by count(*) count view per each page_group. And put to PAGEVIEWS_BY_PAGEGROUP_TABLE

Input



Output



c. KsqlDB : Tumbling Window (Topic 6s)

A **Tumbling Window** is a type of time window that divides time into fixed intervals with no overlap between the windows. Each event belongs to exactly one window, making it easy to calculate metrics such as total sales for a specific time period.

```
CREATE TABLE TOTAL_SALES_BY_PRODUCTGROUP_TUMBLING_WINDOW AS
SELECT
  product_group AS key,
  CAST(product_group AS STRING) AS product_group,
  SUM(sale_price * quantity) AS total_sales,
  TIMESTAMPTOSTRING(WINDOWSTART, 'yyyy-MM-dd HH:mm:ss', 'UTC') AS window_start,
  TIMESTAMPTOSTRING(WINDOWEND, 'yyyy-MM-dd HH:mm:ss', 'UTC') AS window_end
FROM EXTENTED_FLASHSALES_STREAM
WINDOW TUMBLING (SIZE 1 HOUR)
GROUP BY product_group
EMIT CHANGES;
```

In this example, ksqlDB calculates the total sales for each product_group every hour. Each window is distinct and does not overlap with others.

Scenario:

- **Window 1:** From 9:00 to 10:00 — calculates total sales in this period only.
- **Window 2:** From 10:00 to 11:00 — calculates total sales for this interval only.
- **Window 3:** From 11:00 to 12:00 — calculates total sales for this interval only.

If a sale occurs at 10:15, it will be included in **Window 2 (10:00 to 11:00)**. Each window displays the total sales for that specific period.

c. KsqlDB : Hopping Window (Topic 7)

A **Hopping Window** is a type of time window that allows for overlapping intervals. It is defined by a **size** (duration of the window) and an **advance interval** (how frequently the window starts). Events can belong to multiple windows due to the overlap.

```
CREATE TABLE TOTAL_SALES_BY_PRODUCTGROUP_ HOPPING _WINDOW AS
SELECT
  product_group AS key,
  CAST(product_group AS STRING) AS product_group,
  SUM(sale_price * quantity) AS total_sales,
  TIMESTAMPTOSTRING(WINDOWSTART, 'yyyy-MM-dd HH:mm:ss', 'UTC') AS window_start,
  TIMESTAMPTOSTRING(WINDOWEND, 'yyyy-MM-dd HH:mm:ss', 'UTC') AS window_end
FROM EXTENTED_FLASHSALES_STREAM
WINDOW HOPPING (SIZE 1 HOUR, ADVANCE BY 30 MINUTES)
GROUP BY product_group
EMIT CHANGES;
```

In this example, ksqlDB calculates the total sales for each product_group using a 1-hour window that advances every 30 minutes. Each window overlaps with the previous and next window.

Scenario:

- **Window 1**: From 9:00 to 10:00 — calculates total sales in this period.
- **Window 2**: From 9:30 to 10:30 — calculates total sales for this overlapping interval.
- **Window 3**: From 10:00 to 11:00 — calculates total sales for this interval.
- **Window 4**: From 10:30 to 11:30 — calculates total sales for this interval.

If a sale occurs at 9:45:
- It will be included in **Window 1 (9:00 to 10:00)** and **Window 2 (9:30 to 10:30)** because of the overlap.

Each window displays the total sales for its respective overlapping period.

c. KsqlDB : Session Window (Topic 8)

A **Session Window** groups events based on activity rather than fixed intervals. The window remains open as long as events continue to arrive within a specified **inactivity gap**. Once no events are received for the duration of the gap, the session closes.

```
CREATE TABLE TOTAL_SALES_BY_PRODUCTGROUP_ SESSION _WINDOW AS
SELECT
  product_group AS key,
  CAST(product_group AS STRING) AS product_group,
  SUM(sale_price * quantity) AS total_sales,
  TIMESTAMPTOSTRING(WINDOWSTART, 'yyyy-MM-dd HH:mm:ss', 'UTC') AS window_start,
  TIMESTAMPTOSTRING(WINDOWEND, 'yyyy-MM-dd HH:mm:ss', 'UTC') AS window_end
FROM EXTENTED_FLASHSALES_STREAM
WINDOW SESSION (15 MINUTES)
GROUP BY product_group
EMIT CHANGES;
```

In this example, ksqlDB calculates the total sales for each product_group within session windows. A session starts when an event occurs and remains active as long as new events arrive within a 15-minute gap. If no events are received within the gap, the session closes.

Scenario:

- **Session 1**: Starts at 9:00 with sales at 9:05, 9:10, and 9:12.
    - If no further sales occur within the next 15 minutes, the session ends at 9:12.
- **Session 2**: Starts at 9:30 with sales at 9:32 and 9:40.
    - Ends if no sales occur by 9:55.

If a sale occurs at 9:10 and another at 9:12:
- Both events belong to **Session 1** because the gap between them is less than 15 minutes.

If another sale occurs at 9:30:
- A **new session (Session 2)** starts at 9:30, as the gap from the last sale in Session 1 is greater than 15 minutes.

c. KsqlDB : Testing the correctness

statements.sql

```
CREATE STREAM USER_DATA (
    registertime BIGINT,
    userid VARCHAR,
    regionid VARCHAR,
    gender VARCHAR,
    interests ARRAY<VARCHAR>,
    contactinfo STRUCT<
        zipcode VARCHAR,
        state VARCHAR,
        phone VARCHAR,
        city VARCHAR>
) WITH (
    KAFKA_TOPIC = 'User_',
    VALUE_FORMAT = 'AVRO',
    TIMESTAMP = 'registertime'
);

CREATE TABLE USERS_PER_REGION AS
    SELECT regionid,
           COUNT(*) AS USER_COUNT
    FROM USER_DATA
    GROUP BY regionid
    EMIT CHANGES;
```

output.json


```
{
  "outputs": [
    {
      "topic": "USERS_PER_REGION",
      "key": "Region_1",
      "value": {"USER_COUNT": 1},
      "timestamp": 1500023204411
    },
    {
      "topic": "USERS_PER_REGION",
      "key": "Region_2",
      "value": {"USER_COUNT": 1},
      "timestamp": 1500023204412
    },
    {
      "topic": "USERS_PER_REGION",
      "key": "Region_1",
      "value": {"USER_COUNT": 2},
      "timestamp": 1500023204413
    }
  ]
}
```


Input.json


```
{
  "inputs": [
    {
      "topic": "User_",
      "key": null,
      "value": {
        "registertime": 1500023204411,
        "userid": "User_1",
        "regionid": "Region_1",
        "gender": "MALE",
        "interests": ["Sports", "Music"],
        "contactinfo": {
          "zipcode": "12345",
          "state": "NY",
          "phone": "1234567890",
          "city": "New York"
        }
      }
    },
    {
      "topic": "User_",
      "key": null,
      "value": {
        "registertime": 1500023204412,
        "userid": "User_2",
        "regionid": "Region_2",
        "gender": "FEMALE",
        "interests": ["Movies", "Travel"],
        "contactinfo": {
          "zipcode": "54321",
          "state": "CA",
          "phone": "0987654321",
          "city": "Los Angeles"
        }
      }
    },
    {
      "topic": "User_",
      "key": null,
      "value": {
        "registertime": 1500023204413,
        "userid": "User_3",
        "regionid": "Region_1",
        "gender": "OTHER",
        "interests": ["Gaming", "Tech"],
        "contactinfo": {
          "zipcode": "67890",
          "state": "IL",
          "phone": "1112223333",
          "city": "Chicago"
        }
      }
    }
  ]
}
```

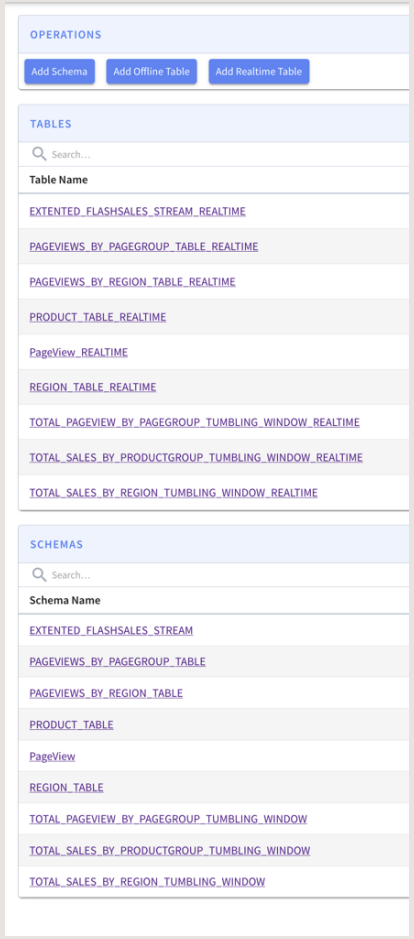

```
(base) ek@ek-docker:~/dads6005/midterm/volumes$ docker exec ksqldb-cli ksql-test-runner -i /opt/app/test/input.json -s
/opt/app/src/statements.sql -o /opt/app/test/output.json
        >>> Test passed!
(base) ek@ek-docker:~/dads6005/midterm/volumes$
```

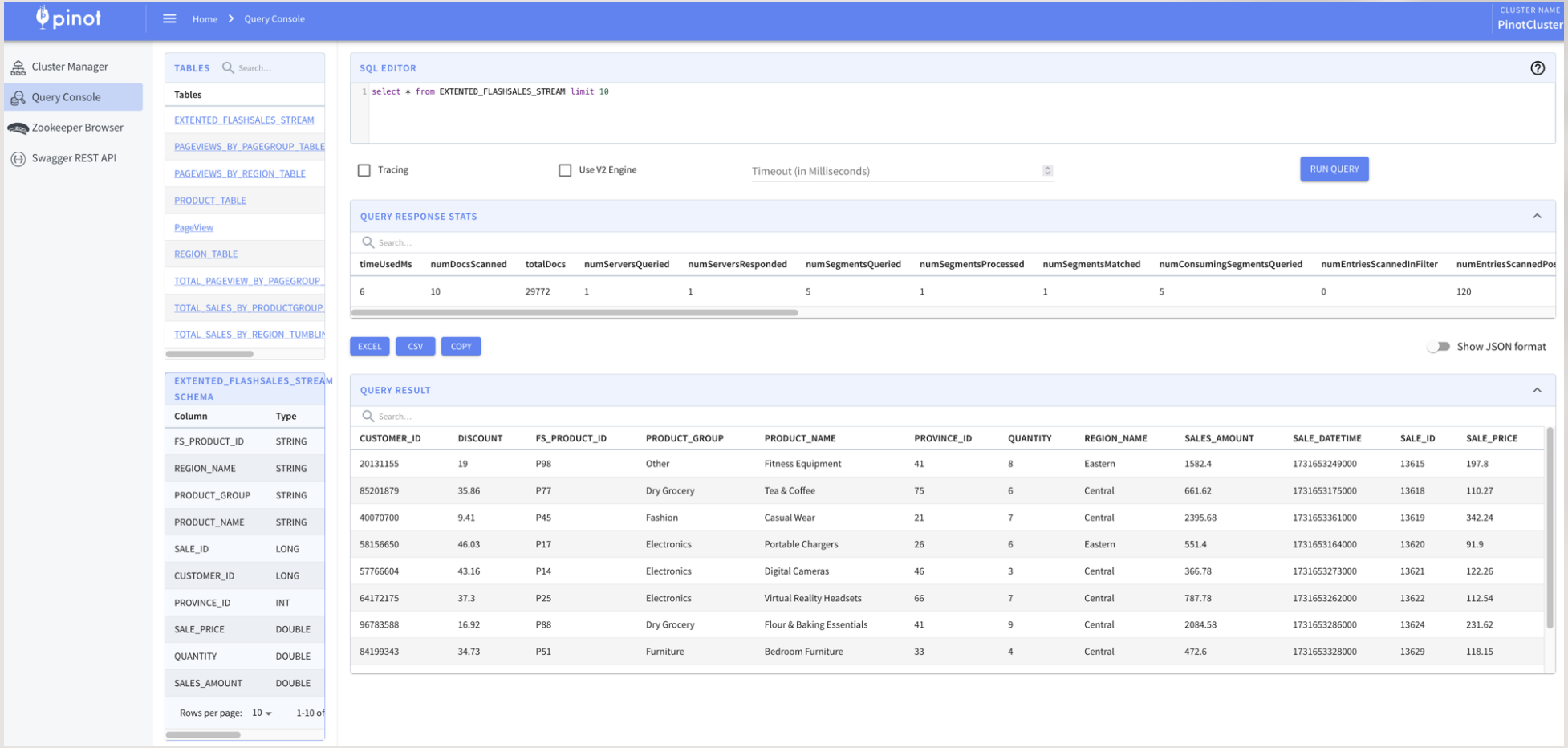
What : **ksql-test-runner**
A tool designed for testing ksqlDB queries without requiring a live Kafka cluster. It takes static input, runs the SQL, and validates the output.
Tests ksqlDB queries (statements.sql) against mock input (input.json) and expected output (output.json).

Why : Ensures queries work as intended, preventing issues in production and enabling CI/CD integration.

How : 1. Takes input data. 2. Runs SQL queries. 3. Compares results to expected output.
“Test passed!” if results match; error output otherwise.

d. Apache Pinot





d. Apache Pinot

What : Serving as online alnalytic processing (OLAP) layer which able to serve huge traffic and load for data reading with good performace.

Why : because real-time data processing and dashboard need a low latency compute and frequency, so Pinot is built for serving this.

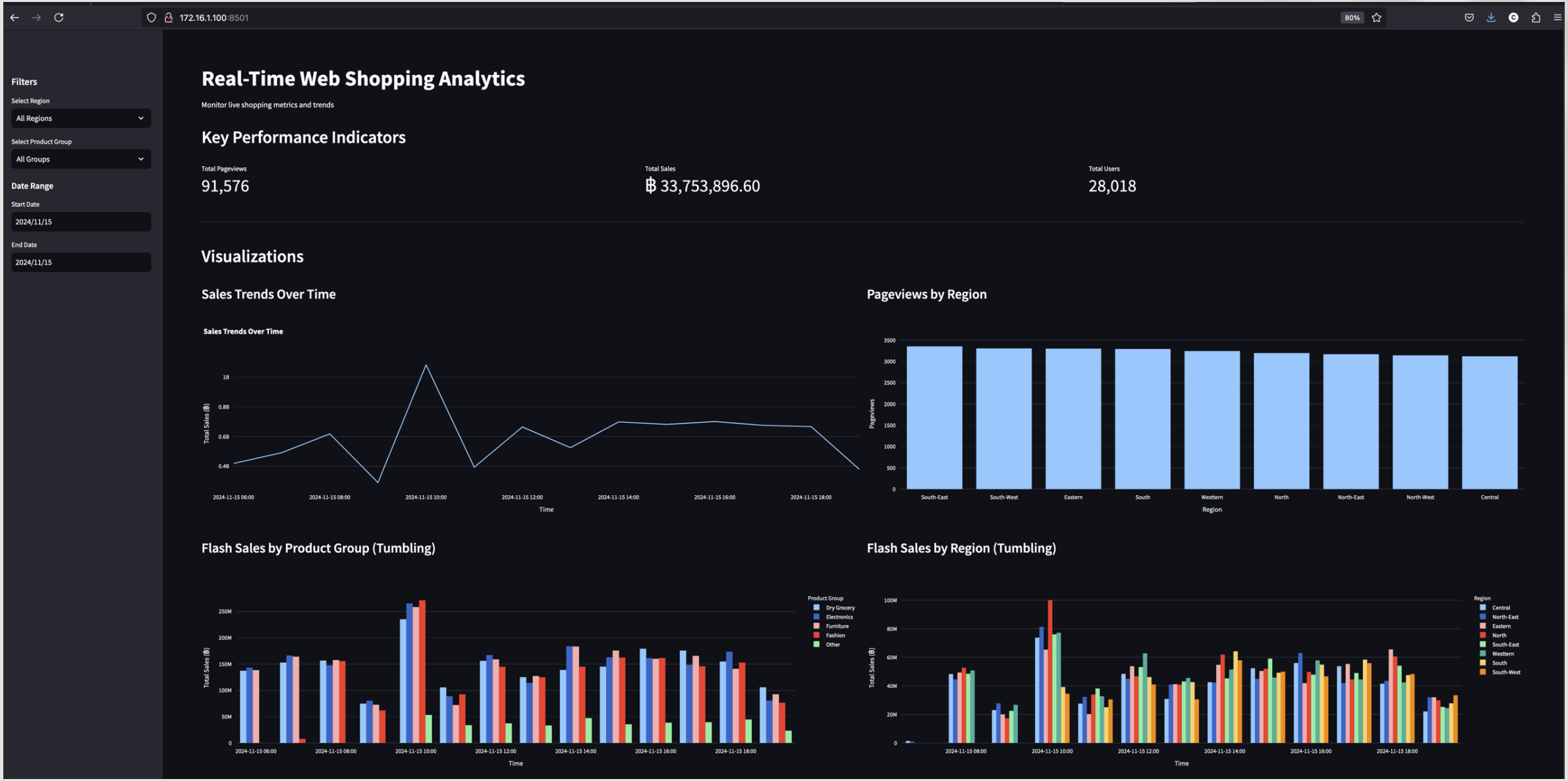
How : When create real-time table in Pinot server it will consume data from Kafka topics (from each brokers) as real-time and keep data within it for a better performance on query.

By creating real-time table in Pinot, we must have topic create in Kafka first and then create Schema align with Kafka schema. The schema name should be the same name as table name in case of adding table by Pinot Controller (UI).

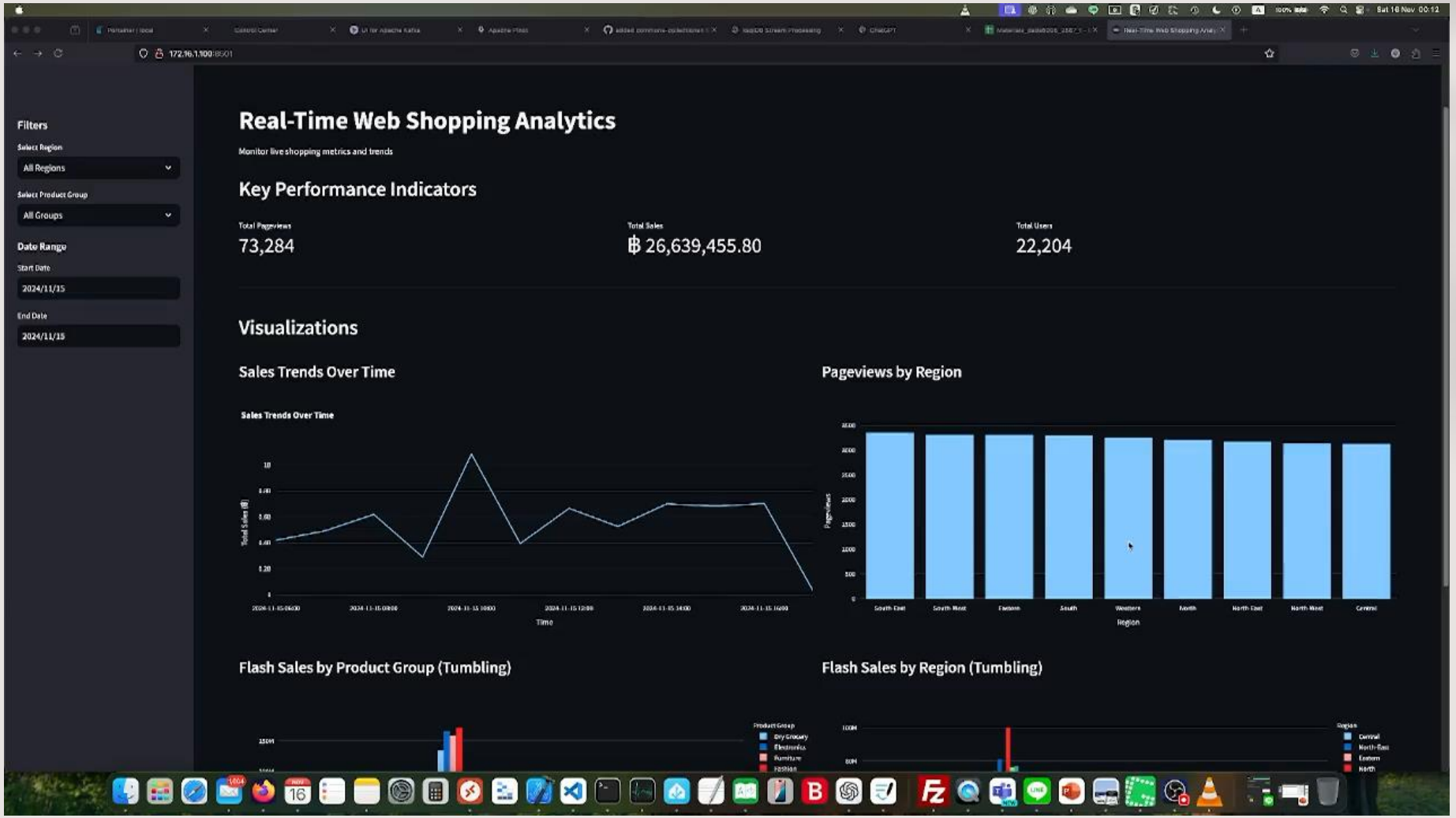
I have create real time table with flushing old data every 24 hours, and each segnent size as of 100 MB.
All of my table use AVRO format which require Schema-Registry server for identify the message schema for a correct deserialize message.

```
"tableIndexConfig": {
  "streamConfigs": {
    "streamType": "kafka",
    "stream.kafka.topic.name": "EXTENTED_FLASHSALES_STREAM",
    "stream.kafka.broker.list": "kafka1:9092,kafka2:9092,kafka3:9092",
    "stream.kafka.consumer.type": "lowlevel",
    "stream.kafka.consumer.prop.auto.offset.reset": "smallest",
    "stream.kafka.consumer.factory.class.name": "org.apache.pinot.plugin.stream.kafka20.KafkaConsumerFactory",
    "stream.kafka.decoder.class.name": "org.apache.pinot.plugin.inputformat.avro.confluent.KafkaConfluentSchemaRegistryAvroMessageDecoder",
    "realtime.segment.flush.threshold.rows": "0",
    "realtime.segment.flush.threshold.time": "24h",
    "realtime.segment.flush.threshold.segment.size": "100M",
    "stream.kafka.decoder.prop.schema.registry.rest.url": "http://schema-registry:8081",
    "stream.kafka.decoder.prop.format": "AVRO"
  },
```

e. Streamlit Dashboard



Quick Demo with Interactive Dashboard



e. Streamlit Dashboard

What : Streamlit use for data visulization (dashboard) in real-time.

Why : Because our kafka plattorm is for real-time data processing so we need dashboard wher we can visualize our data in real-time as well.

How : The dashbaord written by python with library **streamlit, plotly, pandas, pinotdb ;** it comsume data from Apache Pinot brokers (OLAP).

We createa real-time table in Pinot which read data from Kafka and ingestion to Pinot (data will be keep in Pinot server) which serving as online alnalytic processing (OLAP), by using library **pinotdb we can connect and query data from pinot with sql statement and load into pandas data frame then plot visualization with plotly. And all of this process is executing and operation by python with streamlit library.**

There 2 parts in dashboard

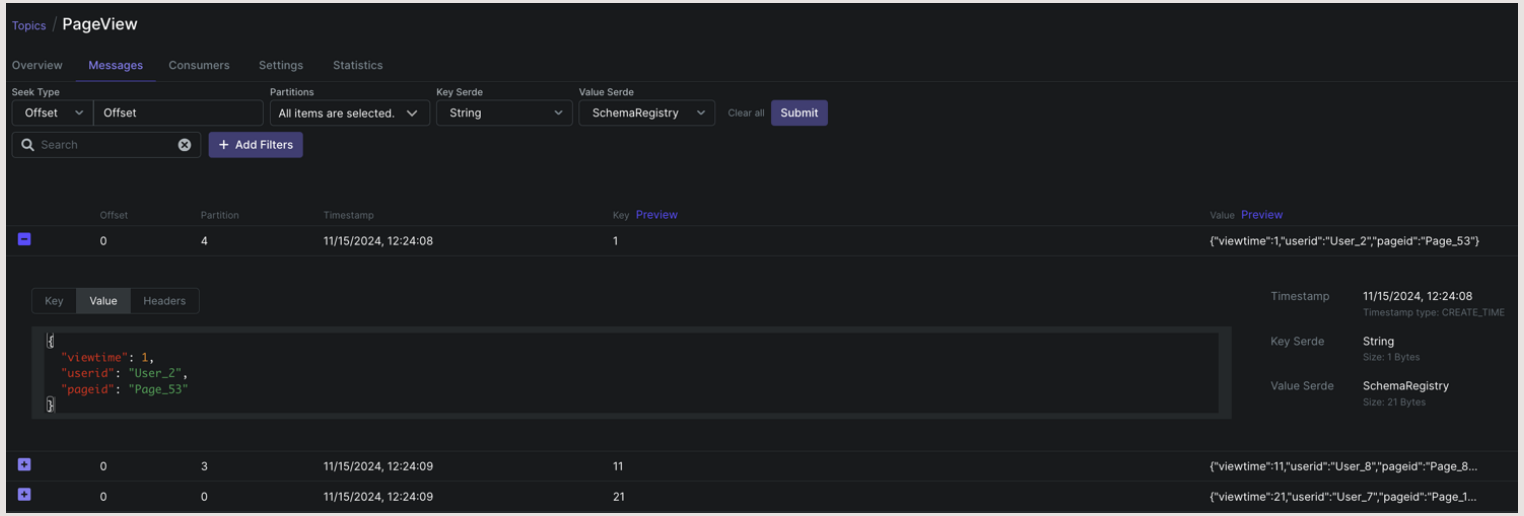
- Key Performace Indicator is show the metrics of Total Page Views, Total Sales and Total Users.
- Visualization chart there are 4 Charts,
  - Overall Sales Trend
  - PageView By Region (Topic 5 Agrregate Count PageView, Group by Region)
  - Flash Sales By Product Group (Topic 6 Tumbling Windows)
  - Flash Sales By Region (Topic 6 Tumbling Windows)

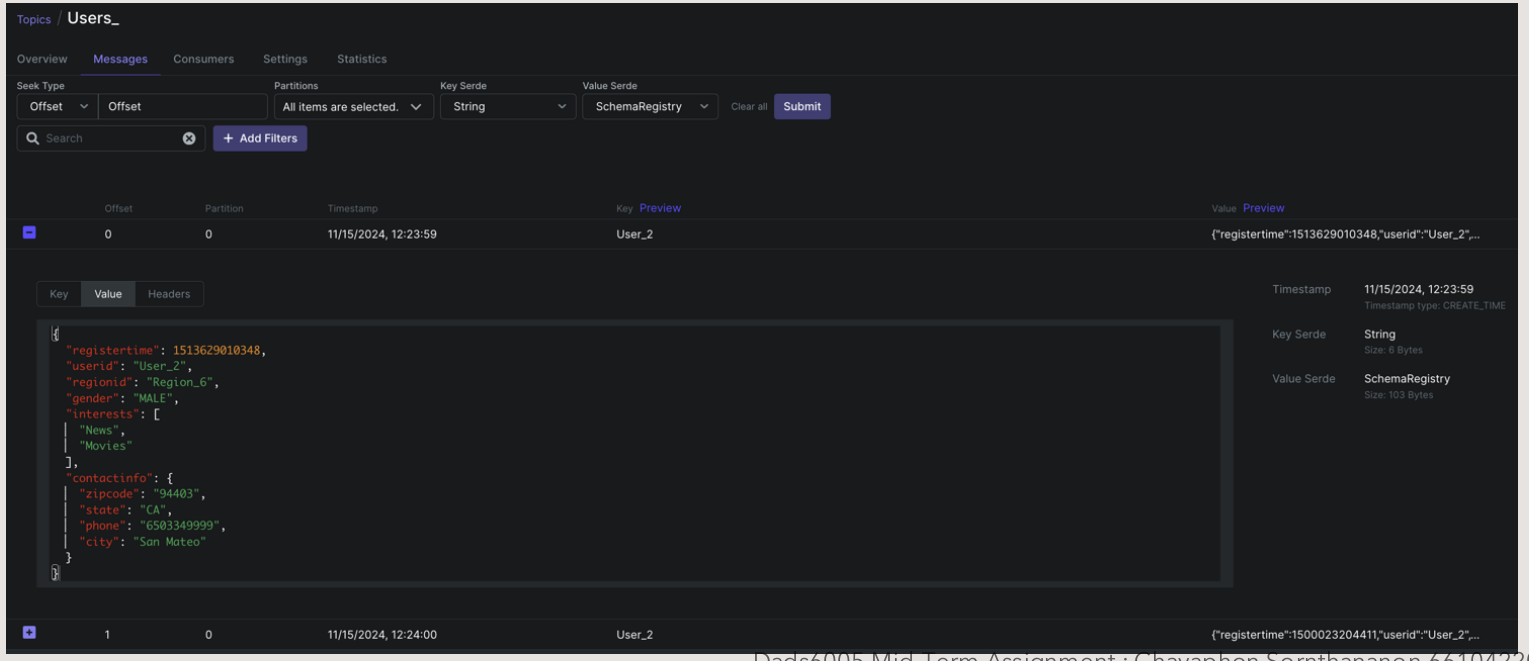
- The are filter on the left side to see data in specific condition (Date, Product Group, Region Name)

The dashboard is set refresh with every 1 second.

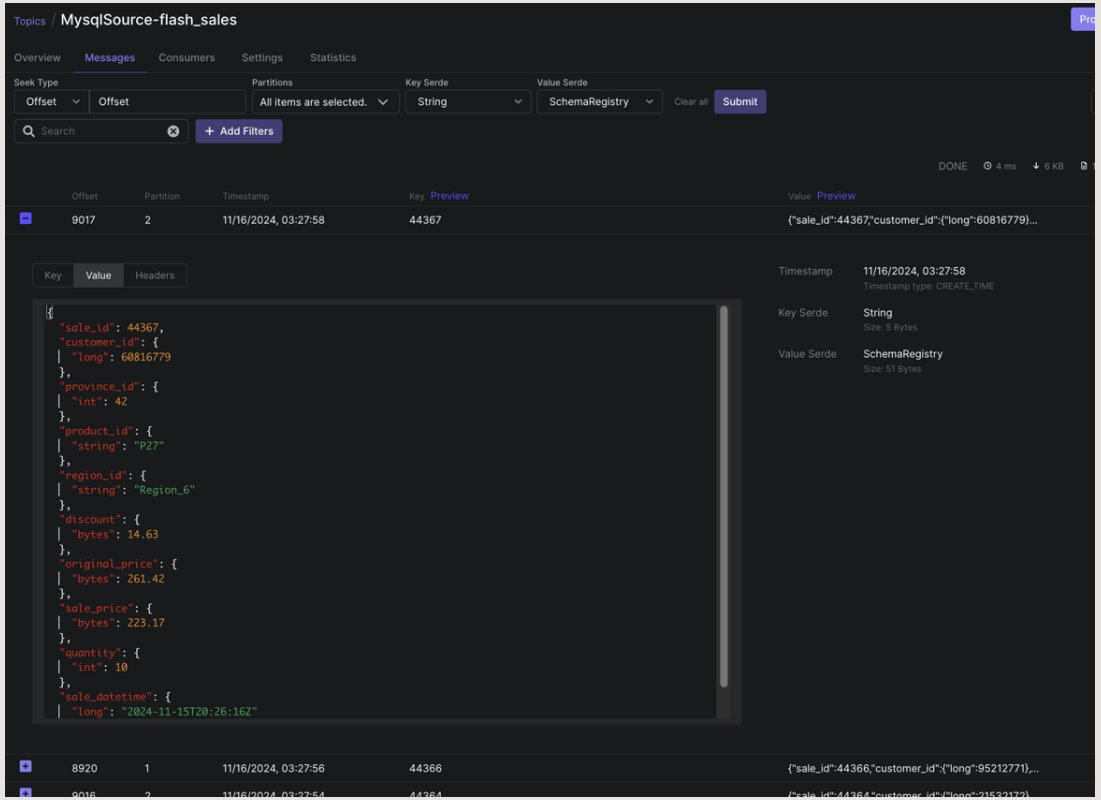
Appendix

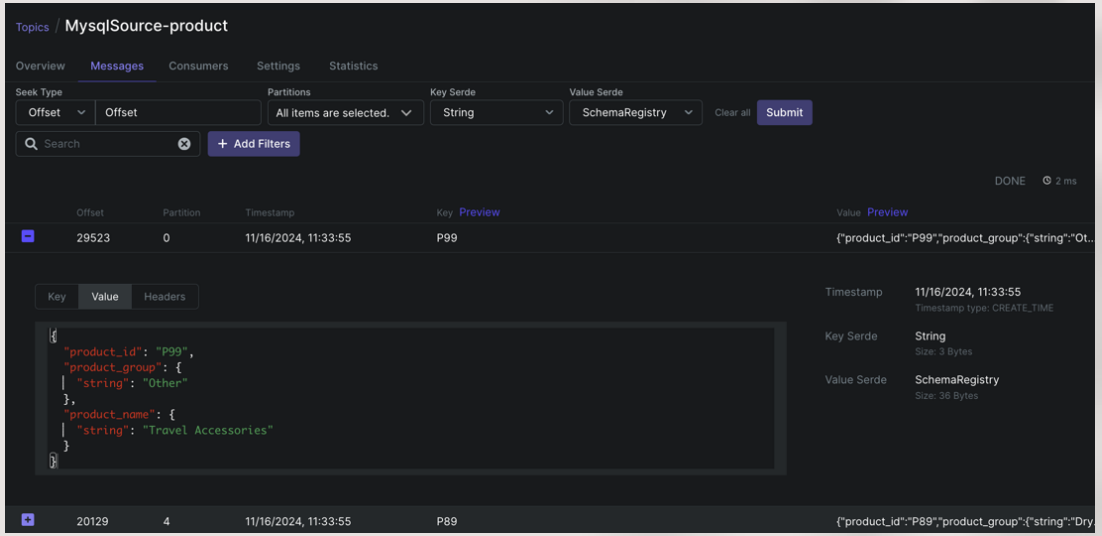
Sample Screentshot Topics for Source 1 & 2

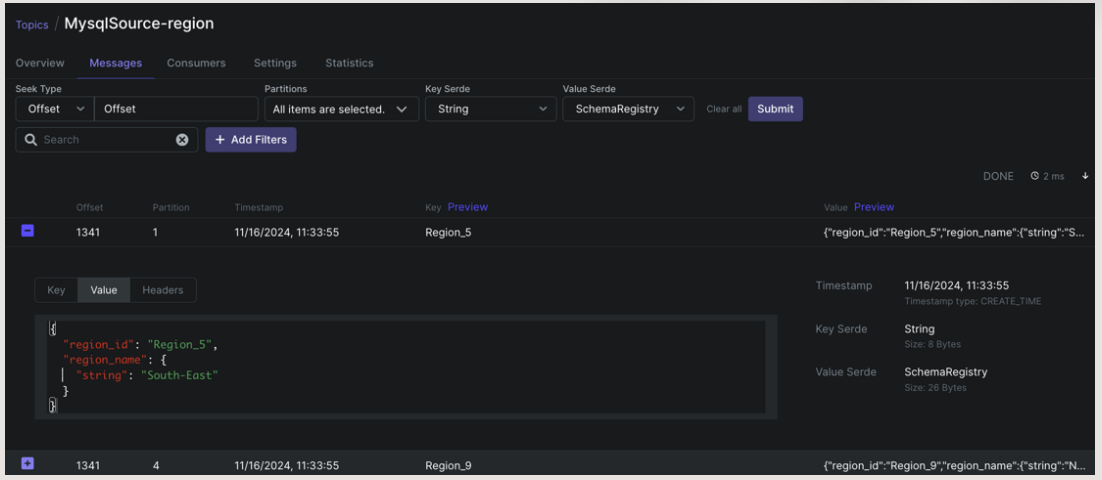


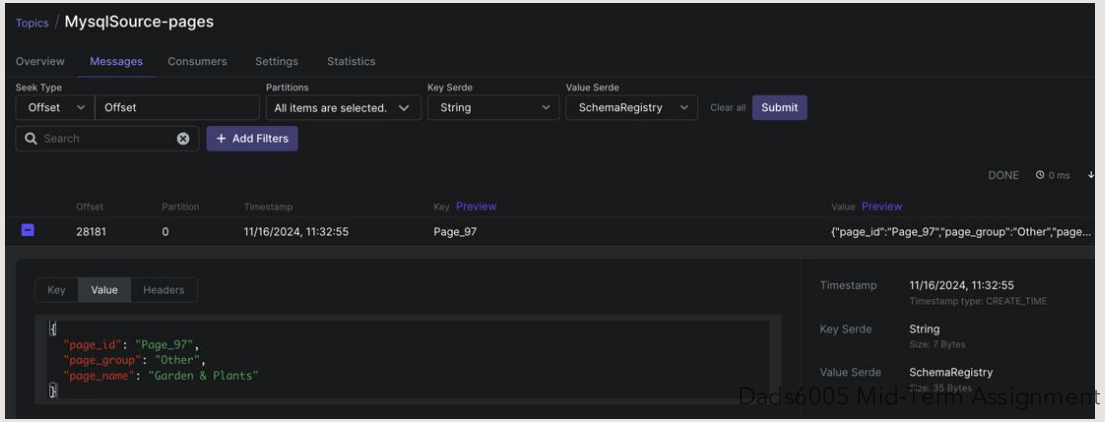


Sample Screentshot Topics for Source 3









Appendix 1 : Output of Mysql Table

```
mysql> show tables;
+--------------------+
| Tables_in_dads6005 |
+--------------------+
| flash_sales        |
| pages              |
| product            |
| region             |
+--------------------+
4 rows in set (0.02 sec)
```

```
mysql> select * from region;
+-----------+-------------+
| region_id | region_name |
+-----------+-------------+
| Region_1  | Central     |
| Region_2  | North       |
| Region_3  | North-East  |
| Region_4  | Eastern     |
| Region_5  | South-East  |
| Region_6  | South       |
| Region_7  | South-West  |
| Region_8  | Westtern    |
| Region_9  | North-West  |
+-----------+-------------+
9 rows in set (0.00 sec)
```

```
mysql> select * from product limit 30;
+------------+---------------+--------------------------+
| product_id | product_group | product_name             |
+------------+---------------+--------------------------+
| P10        | Electronics   | Smartphone Accessories   |
| P11        | Electronics   | Laptops & Notebooks      |
| P12        | Electronics   | Headphones & Earbuds     |
| P13        | Electronics   | Smart Home Devices       |
| P14        | Electronics   | Digital Cameras          |
| P15        | Electronics   | Gaming Consoles          |
| P16        | Electronics   | Television & Accessories |
| P17        | Electronics   | Portable Chargers        |
| P18        | Electronics   | Bluetooth Speakers       |
| P19        | Electronics   | Wearable Technology      |
| P20        | Electronics   | Computer Components      |
| P21        | Electronics   | Networking Devices       |
| P22        | Electronics   | Drones & Quadcopters     |
| P23        | Electronics   | Smart Watches            |
| P24        | Electronics   | Tablet Accessories       |
| P25        | Electronics   | Virtual Reality Headsets |
| P26        | Electronics   | Home Audio Systems       |
| P27        | Electronics   | Printers & Scanners      |
| P28        | Electronics   | Projectors               |
| P29        | Electronics   | e-Readers                |
| P30        | Fashion       | Men's Clothing           |
| P31        | Fashion       | Women's Clothing         |
| P32        | Fashion       | Kids' Wear               |
| P33        | Fashion       | Footwear                 |
| P34        | Fashion       | Bags & Luggage           |
| P35        | Fashion       | Jewelry                  |
| P36        | Fashion       | Watches                  |
| P37        | Fashion       | Sunglasses               |
| P38        | Fashion       | Hats & Caps              |
| P39        | Fashion       | Belts                    |
+------------+---------------+--------------------------+
```

```
mysql> select * from pages limit 30;
+---------+------------+--------------------------+
| page_id | page_group | page_name                |
+---------+------------+--------------------------+
| Page_10 | Electronics | Smartphone Accessories   |
| Page_11 | Electronics | Laptops & Notebooks      |
| Page_12 | Electronics | Headphones & Earbuds     |
| Page_13 | Electronics | Smart Home Devices       |
| Page_14 | Electronics | Digital Cameras          |
| Page_15 | Electronics | Gaming Consoles          |
| Page_16 | Electronics | Television & Accessories |
| Page_17 | Electronics | Portable Chargers        |
| Page_18 | Electronics | Bluetooth Speakers       |
| Page_19 | Electronics | Wearable Technology      |
| Page_20 | Electronics | Computer Components      |
| Page_21 | Electronics | Networking Devices       |
| Page_22 | Electronics | Drones & Quadcopters     |
| Page_23 | Electronics | Smart Watches            |
| Page_24 | Electronics | Tablet Accessories       |
| Page_25 | Electronics | Virtual Reality Headsets |
| Page_26 | Electronics | Home Audio Systems       |
| Page_27 | Electronics | Printers & Scanners      |
| Page_28 | Electronics | Projectors               |
| Page_29 | Electronics | e-Readers                |
| Page_30 | Fashion     | Men's Clothing           |
| Page_31 | Fashion     | Women's Clothing         |
| Page_32 | Fashion     | Kids' Wear               |
| Page_33 | Fashion     | Footwear                 |
| Page_34 | Fashion     | Bags & Luggage           |
| Page_35 | Fashion     | Jewelry                  |
| Page_36 | Fashion     | Watches                  |
| Page_37 | Fashion     | Sunglasses               |
| Page_38 | Fashion     | Hats & Caps              |
| Page_39 | Fashion     | Belts                    |
+---------+------------+--------------------------+
```

```
mysql>
mysql> select * from flash_sales order by sale_id desc limit 10;
+---------+-------------+-------------+------------+-----------+----------+----------------+------------+----------+---------------------+
| sale_id | customer_id | province_id | product_id | region_id | discount | original_price | sale_price | quantity | sale_datetime       |
+---------+-------------+-------------+------------+-----------+----------+----------------+------------+----------+---------------------+
|   44367 |    60816779 |          42 | P27        | Region_6  |    14.63 |         261.42 |     223.17 |       10 | 2024-11-15 20:26:16 |
|   44366 |    95212771 |          64 | P11        | Region_4  |    48.18 |         486.89 |     252.31 |        2 | 2024-11-15 20:24:19 |
|   44365 |    74770033 |          58 | P16        | Region_8  |    21.11 |         216.88 |     171.10 |        8 | 2024-11-15 20:23:38 |
|   44364 |    21532172 |          76 | P56        | Region_5  |    20.85 |         386.04 |     305.55 |        4 | 2024-11-15 20:26:53 |
|   44363 |    88396172 |          57 | P65        | Region_1  |    45.65 |         320.77 |     174.34 |        1 | 2024-11-15 20:24:04 |
|   44362 |    68333674 |          23 | P47        | Region_8  |    37.33 |         353.94 |     221.81 |        6 | 2024-11-15 20:25:16 |
|   44361 |    57175108 |          14 | P66        | Region_6  |     7.61 |         351.79 |     325.02 |        5 | 2024-11-15 20:24:03 |
|   44360 |    46896886 |          33 | P28        | Region_8  |    38.11 |         320.62 |     198.43 |        1 | 2024-11-15 20:27:29 |
|   44359 |    81445923 |          32 | P25        | Region_8  |    16.23 |         160.04 |     134.07 |        6 | 2024-11-15 20:23:48 |
|   44358 |    69216433 |          20 | P28        | Region_2  |    22.76 |         458.38 |     354.05 |        3 | 2024-11-15 20:23:10 |
+---------+-------------+-------------+------------+-----------+----------+----------------+------------+----------+---------------------+
10 rows in set (0.01 sec)
```