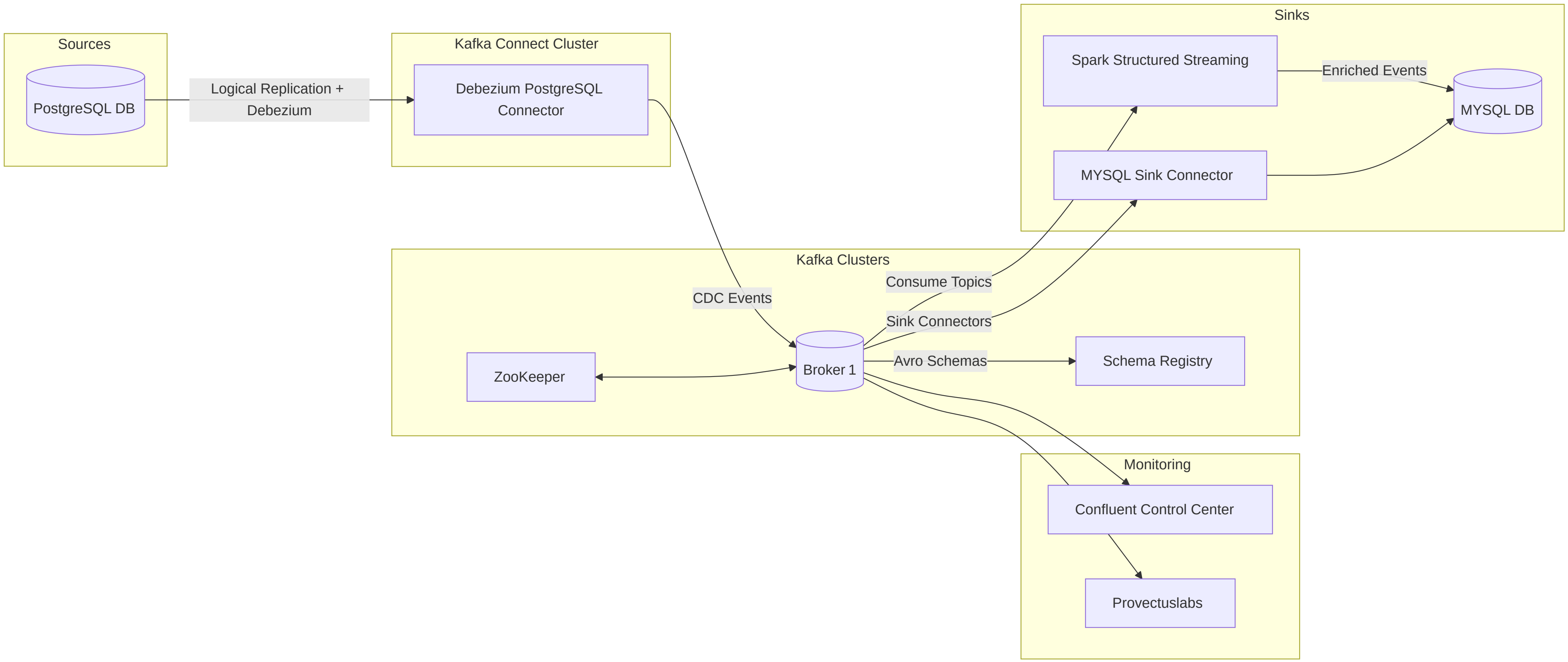
Real Time Streaming

Architecture Diagram of Real Time Streaming



1. Introduction

This document explains how a real-time streaming pipeline ingests change events from source databases, transports them through Kafka, processes them in-flight, and delivers them to downstream systems. I'll cover each component, data flow, and key concepts that are needed to build and operate such a system in production environment.

---

2. High-Level Data Flow

1. **Change Data Capture (CDC)**
   - **Source**: PostgreSQL
   - **Connector**: Debezium PostgreSQL Source Connector
   - **Mechanism**: Logical replication stream (WAL) → Debezium pulls and serializes each INSERT/UPDATE/DELETE into Kafka-friendly events.

2. **Message Broker**
   - **ZooKeeper**: Coordinates the Kafka brokers, maintains metadata, and handles leader election.
   - **Kafka Broker (Broker 1)**: Receives CDC events into the `orders` topic, persists them across partitions and replicas, and serves them to consumers.
3. **Schema Management**
   - **Schema Registry**: Stores Avro (or JSON/Protobuf) schemas for all topics. Producers register schemas on write; consumers fetch schemas on read to ensure compatibility.
4. **Stream Processing**
   - **Spark Structured Streaming**: Consumes one or more Kafka topics, applies transformations/enrichments, and writes results directly to a sink.
5. **Sink Connectors**
   - **MySQL Sink Connector**: Subscribes to Kafka topics and writes events into a downstream MySQL database, maintaining up-to-date tables.
6. **Monitoring & Management**
   - **Confluent Control Center and Provectuslabs Kafka-UI**: Tracks cluster health, topic throughput, consumer lags, and connector status.

---

3. Component Breakdown

3.1 Change Data Capture (CDC)

- **Debezium PostgreSQL connector** : PostgreSQL uses `Write-Ahead Log (WAL)` to record every change before applying it to the actual data. We set `wal = logical` so that PostgreSQL allows row-level changes (insert/update/delete) to be decoded into logical events that CDC tools can consume. Without logical, PostgreSQL would only replicate raw binary file changes (physical replication), which cannot be easily parsed into meaningful per-row CDC events. Although **WAL** always logs all changes, logical decoding will only emit changes for tables which are defined in **publication**, Others are filtered out.
  - **Logical Decoding Plugin:**
    Decodes WAL records into a logical format, **pgoutput** : Native plugin used by PostgreSQL for logical replication
  - **Replication Slot:**
    Acts like a bookmark for a replication client. WAL logs needed by the slot will not be deleted until consumed — ensures no data loss. Different replication slot for different source connector.

  ```sql
  SELECT * FROM pg_create_logical_replication_slot('debezium_slot',                'pgoutput');
  ```

  - **Publication:**
    Specifies **which tables** (and changes) are exposed via logical replication.

    ```
    ALTER ROLE admin WITH REPLICATION;
    CREATE PUBLICATION my_publication FOR ALL TABLES;
    ```

````
   - **PostgreSQL Debezium parameters (postgres_transactions.json)**

```json
"bootstrap.servers": "broker:9092", # Kafka broker hostname and port
"connector.class": "io.debezium.connector.postgresql.PostgresConnector",
"database.hostname": "postgres",
"database.port": "5432",
"database.user": "admin",
"database.password": "admin",
"database.dbname": "transactions",
"max.batch.size": "10", # Maximum number of events to read
"max.queue.size": "20",
"plugin.name": "pgoutput", # Default plugin
"publication.name": "my_publication", # Publication name defined earlier
"slot.name": "transactions", # Replication Slot name
````
````

```
"table.include.list": "public.transaction_details", # List of table to be loaded

"database.server.name": "dev",
"topic.prefix": "dev",
"offset.storage.topic": "dev_connect_offsets",
"config.storage.topic": "dev_connect_configs",
"status.storage.topic": "dev_connect_status",
"offset.storage.replication.factor": "1",
"config.storage.replication.factor": "1",
"status.storage.replication.factor": "1",

"database.history.kafka.topic": "dev_schema_history",
"database.history.kafka.bootstrap.servers": "broker:9092",
"database.history.kafka.replication.factor": "1",

"key.converter": "io.confluent.connect.avro.AvroConverter",
"key.converter.schema.registry.url": "http://schema-registry:8081",
"value.converter": "io.confluent.connect.avro.AvroConverter",
"value.converter.schema.registry.url": "http://schema-registry:8081",
"snapshot.mode": "initial",
"time.precision.mode": "connect",

"transforms": "unwrap",
"transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
"transforms.unwrap.drop.tombstones": "false",
"transforms.unwrap.delete.handling.mode": "rewrite",
"transforms.unwrap.add.fields": "op,source_table,ts_ms",
"include.schema.changes": "false"
```

```
PostgreSQL source connector file is present at :  `/connectors/source/postgres_transactions.json`
```

To run a Kafka Connect source connector, like Debezium for PostgreSQL, we use a POST API request to the Kafka Connect REST endpoint.

```
curl -X POST   -H "Content-Type: application/json"   http://localhost:8083/connectors   -d @postgres_transactions.json
```

To check the status of Kafka connect:

```
curl -s http://localhost:8083/connectors/connector_name/status
```

3.2 Kafka Broker & ZooKeeper

- **Broker**:
  - Hosts one or more **topics** divided into **partitions** for parallelism.
  - Each partition has a single *leader* (handles all reads/writes) and zero or more *followers* (replicate leader data).
  - *Replication factor* ≥ 2 ensures high availability.
- **ZooKeeper**:
  - Manages broker membership, topic metadata, and handles leader elections.

    - In production, run an odd-numbered ensemble (3 or 5 nodes) for quorum.
- **Key parameters**:
    - *ISR (In-Sync Replicas)*: Followers that have fully caught up to the leader.
    - *min.insync.replicas*: Minimum followers that must acknowledge a write to consider it successful.

3.3 Schema Registry

- **Role**:
    - Centralizes schema definitions for Avro/Protobuf/JSON.
    - Validates producer schemas against compatibility rules (BACKWARD, FORWARD, FULL).
    - Allows consumers to evolve without code changes by retrieving schema by subject and version.
- **Key Concepts**:
    - *Schema Compatibility*: Prevents breaking changes in production.
    - *Subject Naming*: `<topic>-key` and `<topic>-value`.

3.4 Stream Processing (Spark Structured Streaming)

- **Overview**
  This Spark application is designed to process financial transaction data ingested from a Kafka topic(dev.public.transaction_details). It performs the following tasks:
- Ingest Avro-encoded Kafka records
- Deserialize messages using Confluent Schema Registry
- Filter and aggregate data in 15-minute windows
- Write aggregated data to a MySQL database using an upsert strategy

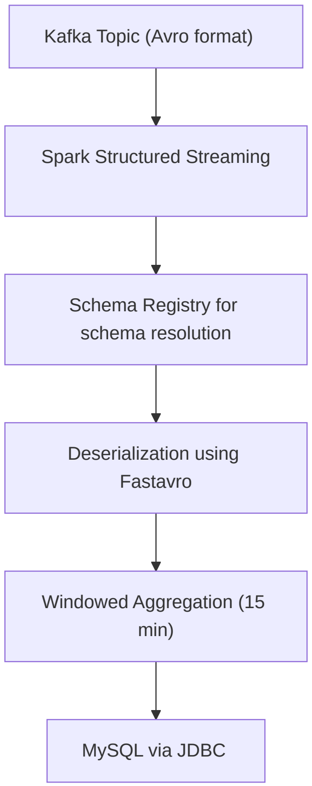


Fig: Flow diagram of spark consumer

- **Window Aggregation:** Uses a 5-minute watermark to handle out-of-order events, Aggregates over **15-minute** sliding windows

```
agg_df = (
    df.withWatermark("last_modified_date", "5 minutes")
      .groupBy(F.window(col("last_modified_date"), "15 minutes"))
      .agg(
         F.sum("amount").alias("total_amount"),
         F.count("amount").alias("txn_count")
      )
)
```

- **Writing to MySQL with Upserts :** `foreachBatch` writes to MySQL using the following strategy:
  - Append mode to a staging table (rp_stage_agg_15min_transactions)
  - Upsert into final table (rp_agg_15min_transactions) using ON DUPLICATE KEY UPDATE
  - Truncate staging table after each batch

```
INSERT INTO rp_agg_15min_transactions (window_start, window_end, txn_count, total_amount)
SELECT window_start, window_end, txn_count, total_amount FROM rp_stage_agg_15min_transactions
ON DUPLICATE KEY UPDATE
window_end = VALUES(window_end),
txn_count = VALUES(txn_count),
```

```
total_amount = VALUES(total_amount);

TRUNCATE TABLE rp_stage_agg_15min_transactions;
```

- **Streaming Trigger and Checkpointing:**
  - Triggers every 1 minute for low-latency aggregation
  - Checkpointing ensures fault tolerance
  - Output mode is update, meaning only changed rows are emitted

```
.trigger(processingTime="1 minute")
.option("checkpointLocation", "checkpoint/gaurav/agg_15_min_n")
.outputMode("update")
```

All the results processed by spark structured streaming are dumped into rp database.
Intermediate Stage in **MYSQL**

```
SELECT * FROM rp.rp_stage_agg_15min_transactions;
```

Final processed table in **MYSQL**

```
SELECT * FROM rp.rp_agg_15min_transactions order by window_start desc;
```

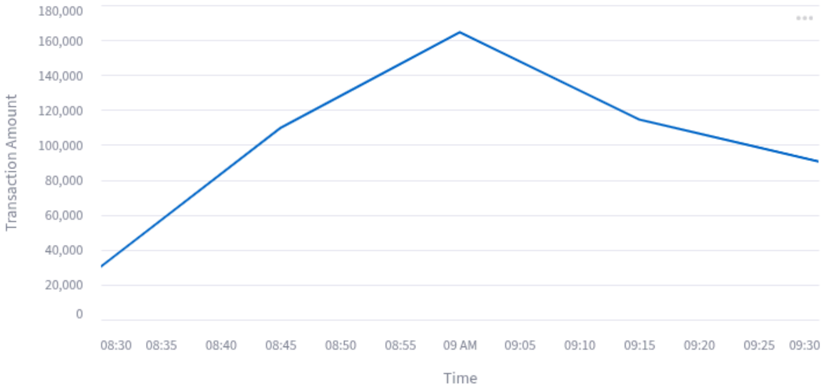
window_start	window_end	total_amoun	txn_coun	created_at
2025-07-22 09:30:00	2025-07-22 09:45:00	90495.77	16	2025-07-22 09:43:10
2025-07-22 09:15:00	2025-07-22 09:30:00	114473.29	22	2025-07-22 09:43:10
2025-07-22 09:00:00	2025-07-22 09:15:00	164494.95	29	2025-07-22 09:43:10
2025-07-22 08:45:00	2025-07-22 09:00:00	109653.91	24	2025-07-22 09:43:10
2025-07-22 08:30:00	2025-07-22 08:45:00	30375.25	9	2025-07-22 09:43:10
2025-07-22 04:45:00	2025-07-22 05:00:00	113291.77	23	2025-07-22 04:58:34
2025-07-22 04:30:00	2025-07-22 04:45:00	179630.80	33	2025-07-22 04:55:28
2025-07-22 04:15:00	2025-07-22 04:30:00	122291.23	26	2025-07-22 04:55:28
2025-07-22 04:00:00	2025-07-22 04:15:00	100418.04	18	2025-07-22 04:55:28

Streamlit UI is available after running file available at
`/home/jovyan/work/gaurav/streamlit/real_time_transactions.py`

```
Streamlit run real_time_transactions.py
```

Real Time Streaming

15 minutes Aggregation of Transaction Amount



	window_start	window_end	total_amount	txn_count	created_at
0	2025-07-22 09:30:00	2025-07-22 09:45:00	90495.77	16	2025-07-22 09:43:10
1	2025-07-22 09:15:00	2025-07-22 09:30:00	114473.29	22	2025-07-22 09:43:10
2	2025-07-22 09:00:00	2025-07-22 09:15:00	164494.95	29	2025-07-22 09:43:10
3	2025-07-22 08:45:00	2025-07-22 09:00:00	109653.91	24	2025-07-22 09:43:10
4	2025-07-22 08:30:00	2025-07-22 08:45:00	30375.25	9	2025-07-22 09:43:10
5	2025-07-22 04:45:00	2025-07-22 05:00:00	113291.77	23	2025-07-22 04:58:34
6	2025-07-22 04:30:00	2025-07-22 04:45:00	179630.8	33	2025-07-22 04:55:28

3.5 Kafka Connect Sink

- **MySQL Sink Connector**:
  - Pulls from Kafka topics, converts message format, and writes to MySQL tables.
  - Can perform upserts, deletes, and configurable batching.
- **Key Concepts**:
  - *Offset Storage*: Connect workers commit their progress back to a Kafka internal topic, allowing exactly-once delivery.
- **MYSQL Sink parameters**
  MYSQL sink connector file is present at : `/connectors/sink/mysql_transactions.json`

```
"connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
"tasks.max": "1",
"topics": "dev.public.transaction_details",

"connection.url": "jdbc:mysql://mysql:3306/transactions?
useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true&sessionVariables=sql_mode='ALLOW_INVALID_DATES,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION'",
"connection.user": "root",
"connection.password": "root",
```

```
"insert.mode": "upsert",
"auto.create": "true",
"auto.evolve": "true",
"pk.mode": "record_key",
"pk.fields": "txn_id",
"delete.enabled": "true",
"batch.size": "3000",
"delete.handling.mode": "rewrite",

"key.converter": "io.confluent.connect.avro.AvroConverter",
"key.converter.schema.registry.url": "http://schema-registry:8081",
"value.converter": "io.confluent.connect.avro.AvroConverter",
"value.converter.schema.registry.url": "http://schema-registry:8081",

"transforms": "unwrap,route",
"transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
"transforms.unwrap.drop.tombstones": "true",

"transforms.route.type": "org.apache.kafka.connect.transforms.RegexRouter",
"transforms.route.regex": "dev\\.public\\.(.*)",
"transforms.route.replacement": "$1"
```

To run a Kafka Connect sink connector, like JDBC for MYSQL, we use a POST API request to the Kafka Connect REST endpoint.

```
curl -X POST   -H "Content-Type: application/json"   http://localhost:8083/connectors   -d @mysql_transactions.json
```

real time data streaming into **MYSQL**

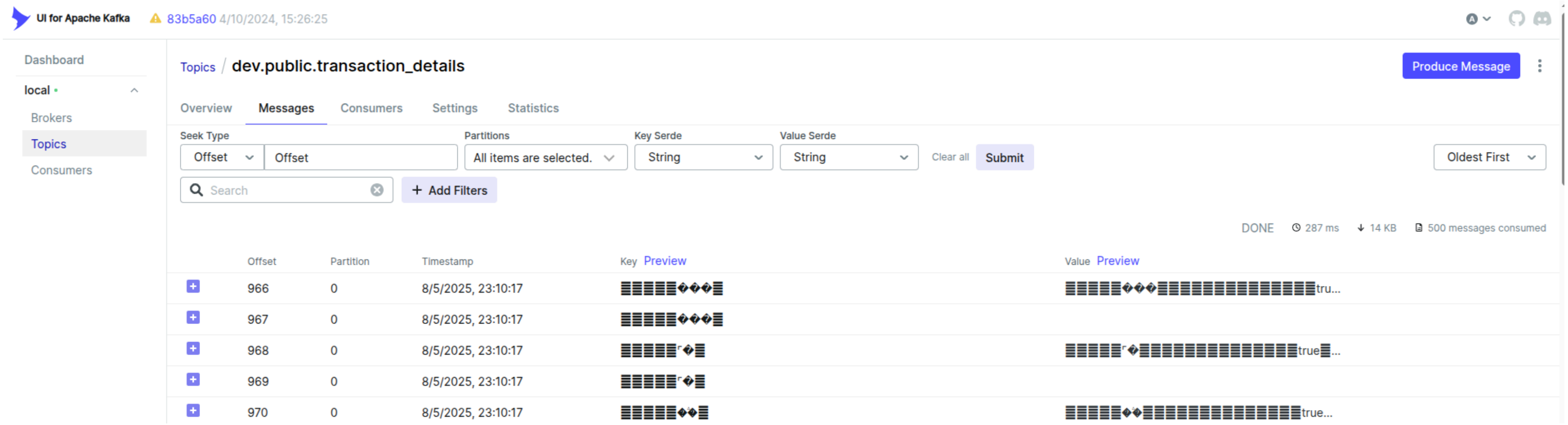
```
SELECT * FROM transactions.transaction_details;
```

txn_id	sender_account_i	receiver_account_i	amount	last_modified_date	product_ic	product_type_i	transactor_module_i	module_ic	status	__ts_ms	__op	__deleted	test
100629703	886794	805711	2085.18	2025-07-15 08:11:20	116	13	7	1	3	1752570320191	c	false	0
100816087	510910	109517	7572.26	2025-07-15 08:49:41	524	2	3	4	2	1752569806500	c	false	0
102170866	806184	537095	5895.74	2025-07-22 03:42:29	777	19	1	3	3	1753160088485	c	false	0
103002651	343459	479371	5042.18	2025-07-15 08:38:37	616	19	7	2	2	1752569713522	c	false	0
103464565	345299	410346	3831.45	2025-07-15 08:31:13	518	12	5	1	2	1752569713522	c	false	0
103647205	677223	973721	1564.00	2025-07-15 08:11:45	136	7	7	2	1	1752568935961	c	false	0
104754670	685833	737735	7205.78	2025-07-16 03:42:05	796	17	1	4	2	1752638871260	c	false	0
105000122	886227	390698	4717.22	2025-07-15 08:19:10	879	14	6	1	2	1752569018065	c	false	0
105102199	652601	400035	581.71	2025-07-22 04:32:30	693	1	7	4	3	1753160315113	c	false	0
105985329	785742	142630	3312.00	2025-07-22 09:19:16	302	4	9	4	4	1753177289377	c	false	0
106612954	806722	115970	4297.75	2025-07-22 03:45:42	532	13	1	3	3	1753160088597	c	false	0

3.6 Monitoring & Management

- **Confluent Control Center and Provectuslabs kafka-ui**:
  - Visualizes broker topic throughput, and consumer lags.

- Monitors connector health and data drift.



3.7 Run and Validation

1. Run `/connectors/source/postgres_transactions.json` , PostgreSQL debezium source connector file to start real time change data capture (CDC)

```
curl -X POST   -H "Content-Type: application/json"   http://localhost:8083/connectors   -d @postgres_transactions.json
```

2. Run `/connectors/sink/mysql_transactions.json` , MYSQL connector file to ingest data into destination database(mysql, transactions db)

```
curl -X POST   -H "Content-Type: application/json"   http://localhost:8083/connectors   -d @mysql_transactions.json
```

3. Run `/home/jovyan/work/gaurav/streamlit/data_producer_transactions.py` file to ingest data into source database table

```
spark-submit   --master local[2]   --jars /home/jovyan/work/jars/mysql-connector-java-8.0.33.jar,/home/jovyan/work/jars/postgresql-42.7.7.jar   data_producer_transaction.py
```

4. Run `/home/jovyan/work/gaurav/streamlit/real_time_transactions.py` file to monitor real time update in streamlit dashboard

```
streamlit run real_time_transactions.py
```

5. Run `/home/jovyan/work/gaurav/streamlit/spark_consumer.py` to perform real time streaming transformation using spark

```
spark-submit   --master local[*]   --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,org.apache.spark:spark-avro_2.12:3.5.0   --jars /home/jovyan/work/jars/mysql-connector-java-
8.0.33.jar,/home/jovyan/work/jars/postgresql-42.7.7.jar   spark_consumer.py
```

---

4. Summary

This real-time streaming pipeline captures database changes in near-real time, reliably transports them through a resilient Kafka cluster, processes or enriches the data on-the-fly, and delivers it to downstream systems(mysql) with end-to-end monitoring. By adhering to the concepts and best practices above, we can build a scalable, fault-tolerant, and maintainable streaming architecture.