Event Processing with Kafka and Spark Streaming

1. Project Overview

This project aims to process **clickstream events** in real-time using **Apache Kafka** for ingestion and **Apache Spark Streaming** for processing. The data flow begins with event ingestion into Kafka, followed by the transformation and enrichment of the data with metadata (such as `correlation_id`, `schema_id`, and `ingestion_timestamp`) using Spark Streaming. After processing, the enriched data is sent back to Kafka for further consumption.

---

2. System Architecture

Key Components:

Kafka:

- **Purpose**: Used for ingesting and publishing clickstream events.
- **Kafka Topics**:
  - `acme.clickstream.raw.events`: The topic where raw events are sent via the API.
  - `acme.clickstream.latest.events`: The topic where enriched events are published after processing.

Spark Streaming:

- **Purpose**: Processes the data in real-time.
- **Functionality**: Transforms data and adds metadata, such as `correlation_id`, `schema_id`, and `ingestion_timestamp`.

FastAPI:

- **Purpose**: Receives events via a REST API and sends them to Kafka.
- **Endpoint**: `/collect` (POST method).

Schema Registry:

- **Purpose**: Validates event conformity with the registered schema, ensuring that the data consumed is valid before being processed.

Minio:

- **Purpose**: Local S3-compatible storage used to persist data, if necessary.

---

3. Data Flow

Event Reception:

- **FastAPI** receives clickstream events via the `/collect` endpoint. The data is then sent to the `acme.clickstream.raw.events` topic in Kafka.

Real-Time Processing:

- **Spark Streaming** consumes data from the `acme.clickstream.raw.events` topic.
- **Transformation and Enrichment**: Spark validates, transforms, and enriches the events by adding the following metadata:
  - `correlation_id`: A unique identifier to correlate events.
  - `schema_id`: Identifier for the schema used for validation.
  - `ingestion_timestamp`: The time the data was ingested.

Publishing Enriched Events:

- After processing, the enriched events are published back to Kafka, specifically to the `acme.clickstream.latest.events` topic.

Persistence and Monitoring:

- **Kafka** stores the events in topics, allowing consumers to access them.
- **Minio** stores data for future analysis if required.
- The system is equipped with **monitoring and logging** to track the status of data processing.

---

4. Code Components

collect.py - FastAPI (Event Reception and Sending to Kafka)

This file contains the implementation of the `/collect` endpoint, which receives events via a POST request and sends them to Kafka.

python

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from kafka import KafkaProducer
```

```
import json

app = FastAPI()

# Kafka Producer
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Event Model
class Event(BaseModel):
    id: int
    type: str
    event: dict

@app.post("/collect")
def collect(events: list[Event]):
    for event in events:
        try:
            # Send the event to Kafka
            producer.send('acme.clickstream.raw.events',
event.dict())
        except Exception as e:
            raise HTTPException(status_code=500, detail=str(e))
    return {"message": "Events received successfully"}
```

Functionality: This code sets up the API that receives events in JSON format and publishes them to the Kafka topic `acme.clickstream.raw.events`.

spark_streaming.py - Spark Streaming (Data Processing and Enrichment)

This file contains the implementation of Spark Streaming, which consumes Kafka events, enriches the data, and publishes it back to Kafka.

python

```
import logging as log
import pyspark.sql.functions as f
from pyspark.sql import SparkSession

# Kafka Configuration
```

```
KAFKA_BROKERS = 'localhost:9092'
KAFKA_CHECKPOINT = 'checkpoint'
KAFKA_TOPIC_RAW = 'acme.clickstream.raw.events'
KAFKA_TOPIC_LATEST = 'acme.clickstream.latest.events'

# App Configuration
ACME_PYSPARK_APP_NAME = 'AcmeSparkStreaming'

# Initialize Logging
log.basicConfig(level=log.INFO, format='%(asctime)s [%(levelname)s]
[%(name)8s] %(message)s')
logger = log.getLogger('acme_pyspark')

# Necessary packages for Kafka and Avro
packages = ['org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.0',
            'org.apache.spark:spark-avro_2.12:3.3.0']

# Initialize Spark Session
def initialize_spark_session(app_name):
    try:
        spark = SparkSession.builder \
            .appName(app_name) \
            .master('spark://spark-master:7077') \
            .config('spark.jars.packages', ','.join(packages)) \
            .getOrCreate()

        spark.sparkContext.setLogLevel("WARN")
        logger.info('Spark session initialized successfully')
        return spark
    except Exception as e:
        logger.error(f"Spark session initialization failed. Error:
{e}")
        return None

# Consume data from Kafka
def get_streaming_dataframe(spark, brokers, topic):
    return spark.readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", brokers) \
        .option("subscribe", topic) \
        .load()
```

```
# Transform and enrich data
def transform_streaming_data(df):
    event_schema = """
    struct<id:long,type:string,

event:struct<user_agent:string,ip:string,timestamp:string,page:strin
g,

query:string,product:long,referrer:string,position:long>>
    """

    events_df = df.selectExpr("CAST(value AS STRING)") \
        .withColumn("data", f.from_json(f.col("value"),
event_schema)) \
        .select("data.*")

    enriched_df = events_df.withColumn("correlation_id",
f.col("id")) \
                           .withColumn("schema_id", f.lit(1)) \
                           .withColumn("ingestion_timestamp",
f.current_timestamp())

    return enriched_df

# Send processed data back to Kafka
def initiate_streaming_to_topic(df, brokers, topic, checkpoint):
    query = df.selectExpr("CAST(correlation_id AS STRING) AS key",
"to_json(struct(*)) AS value") \
              .writeStream \
              .format("kafka") \
              .option("kafka.bootstrap.servers", brokers) \
              .option("topic", topic) \
              .option("checkpointLocation", checkpoint) \
              .start()

    query.awaitTermination()

# Main function
def main():
    spark = initialize_spark_session(ACME_PYSPARK_APP_NAME)
```

```
    if spark:
        df = get_streaming_dataframe(spark, KAFKA_BROKERS,
KAFKA_TOPIC_RAW)
        if df:
            transformed_df = transform_streaming_data(df)
            initiate_streaming_to_topic(transformed_df,
KAFKA_BROKERS, KAFKA_TOPIC_LATEST, KAFKA_CHECKPOINT)

# Execute the main function
if __name__ == '__main__':
    main()
```

Functionality: This code consumes data from Kafka, applies the necessary transformations, and publishes the enriched data back to Kafka.

---

5. Docker Configuration

docker-compose.yml

Here is the configuration for orchestrating Kafka, Zookeeper, and Spark containers using Docker Compose:

yaml

```
version: '3'
services:
  kafka:
    image: wurstmeister/kafka
    environment:
      KAFKA_ADVERTISED_LISTENERS: INSIDE://kafka:9093
      KAFKA_LISTENER_SECURITY_PROTOCOL: PLAINTEXT
      KAFKA_LISTENERS: INSIDE://kafka:9093
      KAFKA_LISTENER_NAME_INTERNAL: INSIDE
      KAFKA_LISTENER_PORT: 9092
      KAFKA_LISTENER_INTERNAL_PORT: 9093
    ports:
      - "9092:9092"
```

---

6. Logging Configuration

logging_config.py

The global logging configuration for the project:

python

```
import logging

def configure_logging():
    logging.basicConfig(level=logging.INFO,
                        format='%(asctime)s [%(levelname)s]
[%(name)8s] %(message)s')
    logger = logging.getLogger('acme_pyspark')
    return logger
```

---

7. Running the Project

Step 1: Build Docker Containers

Navigate to the project root and run the following command to start the containers:

bash

```
docker-compose up -d --build
```

Step 2: Start FastAPI

Navigate to the `api/` directory and run:

bash

```
uvicorn collect:app --reload
```

The API will be available at http://localhost:8000/collect.

Step 3: Run Spark Streaming

Enter the Spark container:

bash

```
docker exec -it spark-master bash
```

Execute the Spark streaming job:

bash

```
/opt/bitnami/spark/bin/spark-submit --packages
org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.0
/opt/spark-apps/spark_streaming.py
```

Step 4: Monitor Kafka UI

Access the Kafka UI at http://localhost:8080 to check the topics and messages.

---

8. Tests and Validation

Integration Tests

You can use tools like **Postman** to test the `/collect` endpoint.

Streaming Tests

Verify that events are processed correctly in Kafka and displayed in the Kafka UI.

---

9. Areas for Improvement

1. **Retry Logic**: Implement retry mechanisms to handle Kafka connection failures.
2. **Kafka Partition and Replication Configuration**: Properly configure partitions and replication for scalability.
3. **Dead Letter Queue (DLQ)**: Set up DLQ to handle invalid events and prevent system failure.
4. **Batch Processing**: Add batch processing alongside real-time streaming for handling large datasets.
5. **Performance and Scalability Tests**: Implement performance tests to ensure the system handles large volumes of data efficiently.
6. **Data Persistence in Minio**: Persist processed data in Minio or another storage solution for backup and future analysis.