Real Time Live Streaming Data Processing with Queue Message (KAFKA)

Eric Adams

Chapman University, CADS

CS 770 Big Data Analytic Final Project

Abstract

Apache Kafka is a distributive stream processing system developed by LinkedIn, open source in 2011, and could be written in Scala (Spark) and Java. When Kafka's components are coupled with the other essential parts of a comprehensive data analytics architecture, it serves as the "central nervous system" through which data is transmitted between input and capture applications, data processing engines, and storage lakes. The project relied on the confluent as a data streaming platform that enables the accessibility, storage, and management of continuous real-time streams by tweaking parameters within the Kafka clusters with absolute configurations.

Keywords: Keyword; KAFKA; Broker, Producer, Consumer, Connector, Topic, Replica, Partitions, Confluent, Schema Registry

INTRODUCTION

Big Data refers to the structured, semi-structured, and unstructured complexity set of data that almost satisfies the 5V's (volume, velocity, variety, value & veracity), yet growing exponentially with time that demand cost-effective, innovative forms of information processing to reveal patterns, trends, and associations for a respective system for a specific project. On a high level, data industries can be divisible into three segments, mainly data engineering, data analyst, and data scientist. Technologies required for big data advancement include Hadoop, Hive, NoSQL, ETL processes, etc. The ability to streamline an entire data process is by constructing a complete pipeline (ETL).

AIMS & OBJECTIVE

To demonstrate a live event of continuous data flow from a data source to the analytics and detail how a messaging queuing distributive system (KAFKA) works.

METHODOLOGY

1. Batch Data Processing (BDP): The data processing does not start instantly but waited for some minimum threshold in order to gather sufficient data for the processing. The waiting time can vary in years, months, weeks, days, hours, etc. Frameworks such as Hadoop, Hives, Apache Spark, etc. are sufficient to complete the BDP. Examples include payroll, billing, and orders from customers.

2. Near Real Time (NRT): When compared to a wall clock, activity completion durations, responsiveness, or perceived latency are significant system quality indicators in a near real-time system. Near-Realtime latency is often 5–15 minutes or more. Examples of near real-time processing include processing sensor data, IT systems monitoring, financial transaction processing, etc.

3. Real-Time Data Process: Continuous input, continuous processing, and steady output of data are necessary for real-time processing. Examples include banking transactions, applications for live cricket scores and applications, gaming applications (live tournaments), trending Twitter hashtags, live dashboards (stock exchange apps, live count in a mall), etc. We need a messaging queue distributive system for stream processing.

Streaming Data Pipeline Architecture

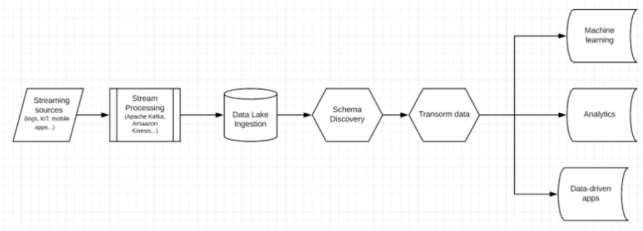

Figure 1 Stream Data Pipeline

Procedure

From Figure 1, we will construct the step-by-step approaches to data streaming.

1. We require an in-memory framework such as a spark.

2. We need a streaming platform such as the Kafka cluster to hold the messaging queue.

3. We need to initialize the in-memory framework, streaming context, etc.

4. To operate on the Kafka cluster, the dataset must be cleansed, transform the data to recognize by a registry schema for its datatype identification.

5. Tune the various parameters in the Kafka distributive system to yield a desirable output.

6. We require a NoSQL database to read each record from the Kafka cluster distributive system.

7. Deliver the output to the process engine for interactive data analysis.

What is KAFKA & The Architecture of KAFKA

Real-time streaming data pipelines and applications that can adapt to the data streams are typically built using Kafka. The three (3) basic component of Kafka includes Producer, Consumer, and Connector. Some use cases of Kafka are metrics, log aggregation, stream processing, messages, website activity, event sourcing, etc. To understand the behavior of Apache Kafka for a distributive streaming system for managing logs purposes, let's gain insight into the Kafka architecture and the interactions between its architectural parameters.

Kafka API & Cluster Architecture

1. Kafka Producer: It's an applications program that publishes or writes data stream to the Kafka topics and determines which partition a specific record is to be published to.

2. Kafka Consumer: It's an application that subscribes to read data streams from Kafka topics. The consumer tracks the subscription records that have been consumed. For the broker to offer a buffer of bytes ready for consumption, the consumer submits an asynchronous pull request.

3. Connector API: It makes it possible to build and use reusable producers and consumers that connect Kafka topics to information systems.

4. Kafka Brokers: A broker is a server or virtual machine participating in a Kafka cluster. Brokers use Apache Zookeeper for its administration and coordination. Several broker instances created within the Kafka clusters can handle numerous metrics of read and write of stream data without affecting the performance of the architect. Each broken is defined by a unique identification for managing divisions of topic logs.

5. Kafka Zookeeper: It's highly priorities for the managing and coordinating of a continual functionality of the entire cluster network. It registers brokers, stores metadata such as topics, partitions, leader information, Quota information, Access Control List (provides vital authorization controls of who to write onto the cluster) dynamic configurations, etc. Zookeeper makes it easier for brokers and topic partition pairings to elect leaders by determining which broker will serve as lead for each partition and which might have identical copies as supporters.

6. Kafka Topic: Topics are small segments created within Kafka that receives published data stream from the producer and make it available for the consumer to subscribe to. Variety records are published to certain topics, and topics organize the messages to support Kafka in achieving its overall network functionality.

7. Kafka Partitions: Within the Kafka cluster, topics are divided into partitions and duplicated between brokers. Multiple consumers from each partition can read the same topic simultaneously. Additionally, producers can include a key to a message, which will send that division for all messages with that key. Key-containing messages are round-robin transmitted to partitions, but keyless messages are gradually added and kept inside partitions. If two messages in Kafka have the same key, you can use keys to enforce the processing order.

8. Consumer group: A Kafka consumer group is made up of consumers who are related or who do the same task. Members of the group who are consumers receive messages from topic partitions through Kafka. A single consumer from the group reads each segment only once at the time it is read. A consumer group has a unique group-id and can run many processes or instances simultaneously. For each consumer group with several partitions, one user can read from a single partition.

kafka Achitecture

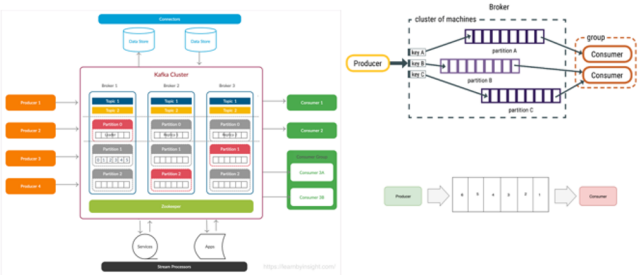

Figure 2 Kafka Achitecture

PROCEDURES

From Figure 2, we will construct the necessary procedures for achieving the streaming goal. Below is a simplified mechanism to follow through.

1. Get Kafka.

2. Start the Kafka environment.

3. Create topics to store the events.

4. Publish (write) events into the topics.

5. Consume (read) the events.

6. Import/Export the data stream with Kafka and Process it.

7. Terminate the Kafka environment.

HOW KAFKA WORKS

A Kafka cluster acts like a central hub for the entirety of events in a system. It basically consists of a server, producers, and consumers. A single Kafka server within the cluster is called the broker. A Kafka cluster usually consists of three (3) brokers to provide enough level of redundancy. A single broker can handle tons of partitions with tons of messaging logs. Kafka brokers are designed as cluster controllers for administrative operations including assigning partitions to brokers and monitoring broker failures. Producers are application programs publishing metrics of data streams to the Kafka topics whiles consumers read data streams from Kafka topics. The consumer keeps track of its position in the data stream by recognizing which offset has already been consumed by storing offsets in the zookeeper or Kafka cluster. Consumers can only read committed messages. Consumer partitioning (consumer groups) usually works together. The consumer ensures that each partition is only consumed by one member of a consumer group to enable consumers to scale horizontally to consume topics with greater messaging logs. Moreover, if a consumer fails, the remaining consuming members can rebalance the partitions to make it up for the missing consumer member. Offsets are written at a time messages are written into Kafka and correspond to a specific record in a specific partition. These brokers receive messages from the producers, assigning offsets and committing messages to desks. It is also responsible for consumers' pull requests and serving messages. In Kafka, messages are sent to the topic. Topics enhance a means of categorizing sent data into various partitions which can process new data streams and metrics of stored data. Within the same topic, multiple partitions can have different offsets. Each partition has a separate log commit which guarantees the messaging orders across the typical partition. A partition is owned by a single broker in the cluster known as the leader partition. These partitions may also be assigned to multiple topics within brokers to ensure a replication effect. This allows other brokers to take leadership in case there is a broker failure. Nevertheless, all producers and consumers must operate with the partition connected to the leader. Partitioning data makes simplifies horizontal scalability, parallelisms, and fault tolerance to improve throughput. Each partition is read by a distinct consumer which allows high throughput and low latency since both partition and consumers can be split into multiple servers. In Kafka, a message is a single unit of data in the form of a byte or array with a different datatype that can be received or sent. A message has a key and value recognized by a schema registry that can be controlled to multiple partitions within the same topic. Since new records might be written to a partition in a round robin-fashion, we can avoid this issue by redefining a consistent way for choosing the same partition based on the record's key designed from a schema registry to assign the same partition to the same key. Sending records over the architecture network could create a lot of overhead, thereby, writing messages into Kafka in batches is desirable. A batch is a collection of records, easily compressed with efficient transfer capability; produced for the same topic and partition which provides a tradeoff between latency and throughput by furtherly controlled a tweak in Kafka settings. In as much the dataset is in bytes, there are multiple schema options such as JSON, XML, Avro, or protobuf that could be used to transform the datatype to a key-value pair. A reliable key feature of Kafka is retention which provides durable record storage. Kafka brokers are configured with a default retention setting within the confluent streaming plate form. Confluent is a full-scale data streaming that enables a user to easily access, store and manage data as continuous, real-time events. Now, the parameter settings of topics within the confluent could stay for a certain period (7 days by default) or the topic reaches its maximum capacity. Older messages are expired and deleted to allow the retention configurations to continue to have space for new records. Kafka has vital reliability guarantees such as message ordering. For instance, if message A was written before message B using the same producers in the same partition, then Kafka guarantees that the offset of message B will be higher than message A, which allows consumers to read message A before message B. Messages are considered committed when written to the leader and all in synchronous replica's and these can be configured. Committed messages will stay intact for as much as one replica remains alive and the retention policy holds. Kafka provides at least one delivery semantics message. Kafka systems are built with reliable trade-offs reliability and consistency versus availability, high throughput, and low latency.

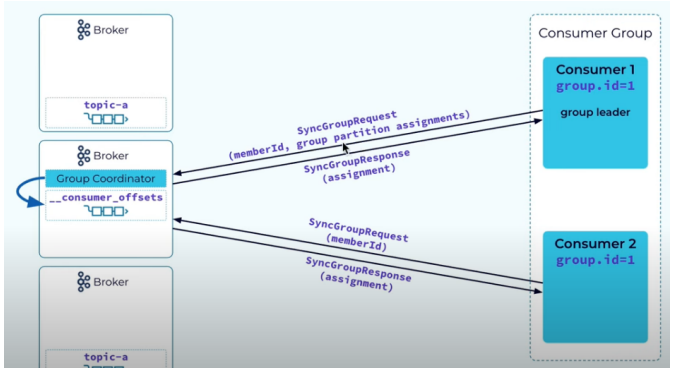

Figure 3 How to parallelize consumers

In Kafka, data are stored based on how it's processed. The process is usually done in consumer applications. However, the consumer group is an important means of scaling up consumer instances. Once a consumer group is defined if subscribed to, can be evenly divided among the listed consumers, then the records could be processed in parallel. Partition is the unit of load distribution. Nevertheless, if a new member joins or existing members leaves, the records will be rebalanced. Considering Figure 3, the workforce behind the consumer group concept is called the Group Coordinator which combines all members in the consumer group and coordinates workload assignment to all members. Each consumer is required to identify a respective group coordinator located in the broker. The Group Coordinator is located within the broker, which stores all the metadata of a particular group. The group coordinator is determined by the leader of the partition a consumer group will be hashed into. A consumer instance sends a joint group request to the group coordinator by including a member's ID, and subscription, For non-consumer instances, the group leader only sends the member's ID. Considering some assignment strategies like range partitions using Figure 4. In this scenario, partitions are distributed at each individual topic level. For instance, topic-1 will divide up its partitions among consumer members and then proceeds to the next topic, and so on... As such, partition-0, from both topics is assigned to consumer-1, and similarly, partition-1 from both topics will be assigned to consumer-2. However, the rest of the consumers will be idle. Here, it's essential to do colocating of joints. You might have several topics with shared keys.

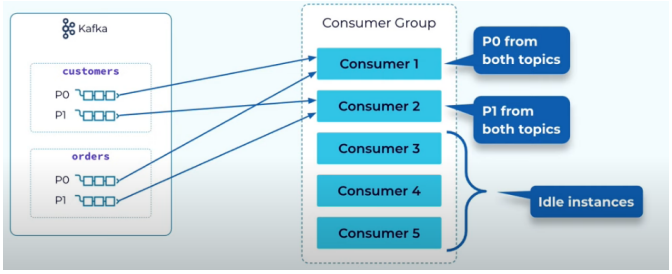

Figure 4 How to parallelize consumers

So, all the records with the same key will be assigned to the same partition by having the same partition assigned to the consumer, thereby, can easily be joined across the two topics locally.

Schema Registry for Data Serialization and Deserialization

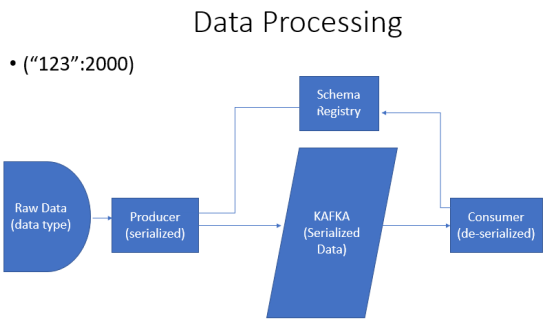

Figure 5 Schema Registry

Figure 5, In Confluent Kafka, there is a concept called Schema Registry which acts like a central repository for the schemas. It holds all the schema datatypes for a particular topic. The schema registry also helps in determining the kind of data that will write for each topic and which schema format it has to be used. There are several file format schema repositories namely; JSON, AURO, Protobuf, etc. For instance, A particular producer will be reading data with the same schema format. We first serialized the dataset to ensure data integrity and consistency. Also, the consumer will first contact the schema registry to check for the latest schema version needed to use for data deserialization and later subscribes to the Kafka topic to read the latest or earliest records from the topic based on the user's configurations. If a new data stream is to be published on any schema version say V1, after serialization, the consumer will also read from that version (V1). However, if new data follows a different schema version say (V2), the previous version (V1), will be canceled, and the data published to the schema version (V2) will then be serialized, then the consumer will read from the updated schema version (V2). The latest schema version is always required to be read, however, if the user wants to read the earliest version, it's the user's responsibility to tweak or make changes in the architecture to be able to read the desired version that sees fit.

DATA ANALYSIS

In this chapter, we shall look at the information that can be derived from the data collected and make inferences based on our outcome of results of the live streaming and then give a conclusion and necessary recommendations. Online Kaggle sources were used to gather the data for the car streaming test study, which determines the continual flow of cars entering a parking lot over time. The project relied on the confluent streaming plate form for tweaking the various parameters in the Kafka cluster. According to Figure 6, there are 15411 cars parked

Figure 6 Data Analysis of Event

in a parking lot and the goal is to run a real-time streaming processing of cars entering the parking instantaneously.

Confluent Kafka Streaming Setup Procedure and Processing

1. Setup Cluster
2. Topic Creation
3. API keys generations
4. Schema Registry

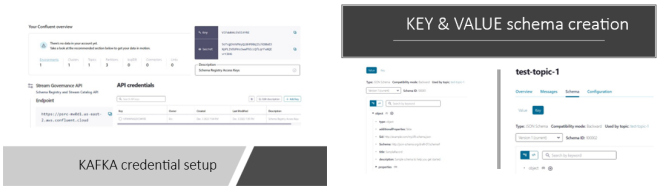

Figure 7 Confluent Setup

We are required to install Confluent Kafka in order to use its APIs to connect to the Kafka cluster. We will be writing a python code to configure with the Confluent in order to certainly ensure the streaming processing works. Find attached python codes from my GitHub account for the Producer and Consumer activation.

Result, Conclusion & Recommendation

Based on the results obtained from this research work, a live stream event from the continual parking of different car models was illustrated for both the producer and the consumer. Moreover, from Figure 8, the stream data outputs can also be verified on the schema registry. We can search for any records from the confluent schema registry based on their topic and partitioning type for any future processing. Whenever the stream messages are successfully delivered to the Kafka topic from the producer, we initialize a delivery report function which helps in printing all offsets and partitions which are being published. Also, each time a good amount of records are published, there is a need to flush the buffer memory in order to avail space to accept new records.

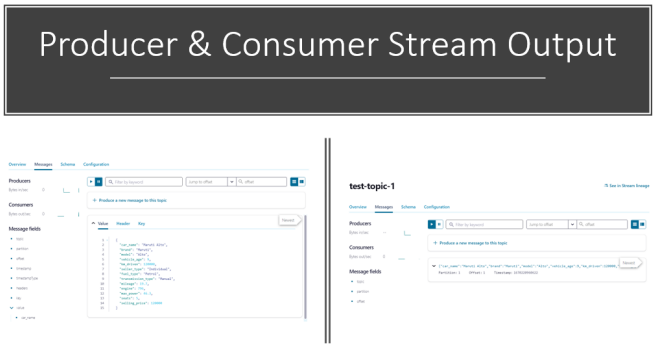

Figure 8 Streaming Live Check Result on the Kafka Confluent Interface