Term Project

CS 777 – Big Data Analysis

Technology: APACHE KAFKA

Name: Sarvesh Krishnan Rajendran

BUID: U86908171

Boston University Metropolitan College

Under the guidance of

Dr. Farshid Alizadeh-Shabdiz

APACHE KAFKA INTRODUCTION

Apache Kafka is a distributed event streaming platform designed for high-throughput, real-time data flows. Kafka, which was originally developed at LinkedIn and then open-sourced in 2011, has become a key tool for large-scale, real-time data processing systems. It is meant to publish, subscribe to, store, and process streams of records in real time, allowing organizations to respond to and analyze continuous data flows.

Kafka works well as an alternative to a more traditional message broker. Message brokers are used for a number of purposes (decoupling processing from data providers, buffering unprocessed messages, etc.). In comparison to most messaging systems, Kafka provides higher throughput, built-in splitting, replication, and fault tolerance, making it an excellent choice for large-scale message processing applications.

In modern applications, where huge volumes of data are generated at a rapid pace, having a robust infrastructure to handle and process these data streams in real time is crucial. Kafka’s architecture is built around scalability, fault tolerance, and durability, making it a preferred choice for mission-critical applications across various industries.

Apache The ideas of producers, consumers, topics, and partitions are central to Kafka's operation. While consumers subscribe to various topics to receive the most recent records, producers send records to specified topics, which are essentially categories where messages are stored. Through data distribution over several servers or nodes, parallel processing, and data redundancy, Kafka's partitioning method further improves scalability. Because of its adaptability, businesses may develop real-time pipelines for event-driven architectures, enabling use cases like log aggregation, metrics gathering, and high-performance stream processing.

Apache Kafka serves as a powerful tool for building real-time data pipelines and event-driven applications. Its versatile design allows it to be used in a wide range of use cases such as log aggregation, stream processing, website activity tracking, and metrics collection. By separating data producers from consumers, Kafka enables seamless scalability, enhances system reliability, and maintains low latency, all of which are critical for real-time analytics and data processing.

ARCHITECTURE

The fundamental ideas of producers, consumers, brokers, subjects, and partitions serve as the foundation for its fault-tolerant and scalable design. Let's break down its main components:

Producers: Producers are the data-generating entities responsible for publishing (or sending) messages to Kafka topics. They are designed to handle massive amounts of data and push it to specific topics within a Kafka cluster.

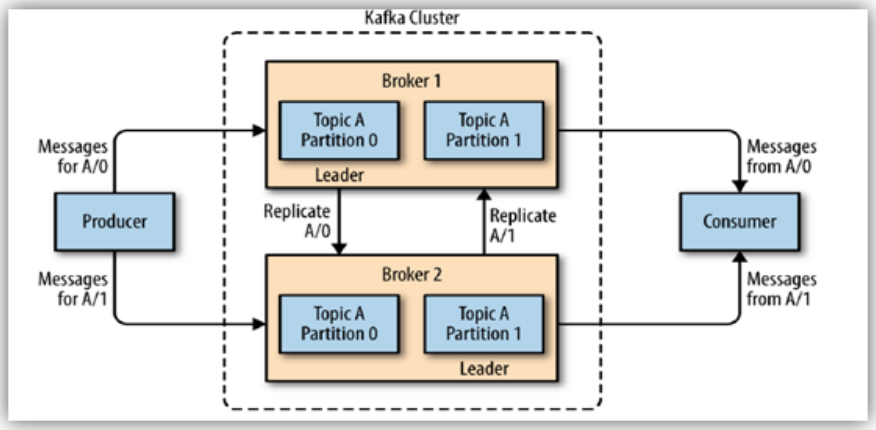
Topics: A topic in Kafka is a category or feed name to which records are published. Think of a topic as a log where events are recorded and stored. Topics are further divided into partitions, enabling parallelism, scalability, and fault tolerance. Each record within a topic is assigned a unique offset, allowing precise identification.

Partitions: Kafka topics are split into smaller units called partitions. A partition is essentially a sequential and ordered record set, and each record in a partition is assigned an offset. This partitioning allows Kafka to scale horizontally, with each partition hosted on different brokers, thereby improving throughput. Additionally, partitions support Kafka's fault tolerance mechanism by replicating data across multiple brokers.

Consumers: Consumers are entities that read data from Kafka topics. Kafka uses a consumer group concept, allowing multiple consumers to coordinate the consumption of messages from topics. Each consumer in a group reads data from different partitions, ensuring that messages are processed efficiently.

Brokers: Brokers are servers that form the backbone of Kafka. They store topic data and handle incoming requests from producers and outgoing requests to consumers. In a Kafka cluster, multiple brokers work together to maintain the entire dataset. Kafka achieves fault tolerance through data replication, where each broker hosts multiple partitions and their replicas.

Zookeeper: Zookeeper was traditionally used for managing the metadata and maintaining the state of the Kafka brokers. It keeps track of the distributed cluster and provides information about each broker's health and the partitions' leaders. However, newer versions of Kafka are moving towards KRaft mode, which eliminates the need for Zookeeper by providing an inbuilt consensus protocol for managing the metadata.



APACHE KAFKA ADVANTAGES

Scalability: One of Kafka's main selling features is scalability. The idea of painless expansion the moment your firm requires it is inherent to its basic architecture. Several data centers and cloud infrastructures may house a single cluster. Partitions can be placed on various brokers by a single subject. This makes horizontal scalability simple; all you need to do is add more servers or data centers to your current setup to accommodate more data.

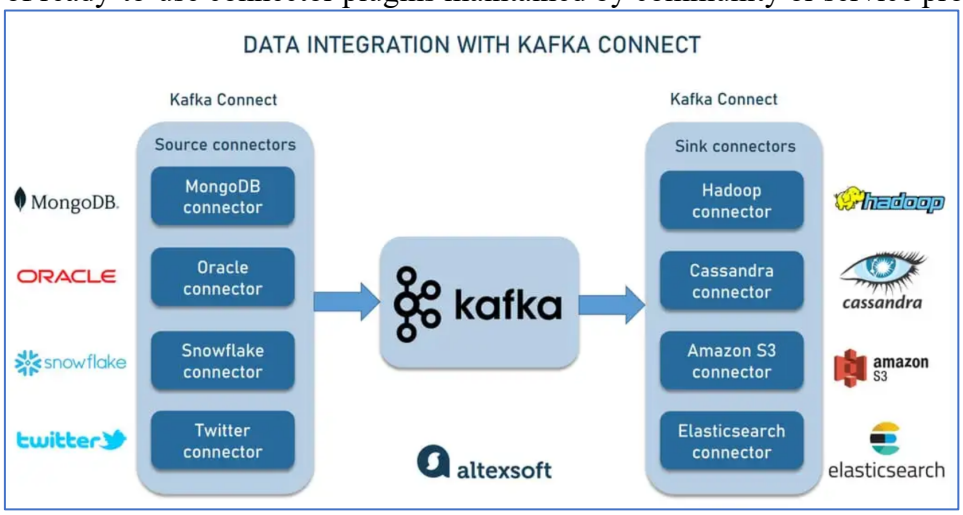
Multiple producers and consumers: Kafka is built to accommodate a large number of customers from both sides. Streams of messages might be committed by several producers to the same topic or to different topics. This makes it possible for systems that use Kafka to compile and standardize data from a variety of sources.

Support for different consumption needs: You can easily scale consumption with Kafka. Additionally, it accommodates various clients' needs and capacities. Apps can choose to consume information in batch or real-time mode and read data at their own pace thanks to the pull-based approach. You can also establish data retention guidelines for a certain topic using Kafka. Depending on the needs of the customer, they will determine how long the messages are kept on the disk.

High performance: All of the above-mentioned result in high performance under high load. You can scale up producers, consumers, and brokers to transmit large volumes of messages with the minimum possible delay (in real time or near real time.)
The performance consists of two main metrics, throughput and latency.

High availability and fault tolerance: Kafka has proved itself as an extremely reliable system. It achieves high availability and fault tolerance through replication at the partition level. In short, Kafka creates several copies of the data and stores them on several brokers. Customers will still be able to access the information even if one broker fails or goes offline.

Kafka Connect for simple integration: Kafka Connect is a free component of the Kafka ecosystem that enables simple streaming integration between Kafka and its clients. The tool standardizes work with connectors, programs that enable external systems to import data to Kafka (source connectors) or export it from the platform (sink connectors). There are hundreds of ready-to-use connector plugins maintained by community or service providers.



Security: Kafka provides several security features, such as authorization, authentication, and data encryption while in transit. Therefore, if your topic has sensitive material, an unauthorized user cannot pull it. An audit log to monitor access is an additional security precaution. Additionally, you may track the origins of messages and who makes changes to what, however this feature necessitates additional coding.

APACHE KAFKA DISADVANTAGES

Steep learning curve and technological complexity:

The amount of documentation, guidelines, and tutorials does not imply that you will be able to quickly overcome all technological problems. Kafka is complicated in terms of cluster setup and configuration, maintenance, and data pipeline architecture. Even those with strong technical expertise will need time to learn the platform.

Depending on the kind of deployment (cloud or on-premise), cluster size, and number of integrations, deployment time can range from days to weeks to months. Many firms eventually seek assistance from Kafka vendors (mostly Confluent) or other third-party service providers.

Need for extra tech experts

Kafka cannot just be installed and let to run. A few of in-house specialists are required to constantly monitor what's going on and tweak the system. If your organization lacks the necessary technical skills, you must recruit specialists, which increases the project cost.

No managing and monitoring tools

Apache Kafka does not provide a UI for monitoring, which is crucial for performing tasks correctly. Tech teams must devote substantial effort to this aspect of everyday operations, ensuring that data flow do not halt and nothing is lost. There are third-party administration and monitoring tools for Kafka, but many developers wish they were included into the platform.

ZooKeeper issue

Kafka has relied on ZooKeeper to manage brokers, store information, and pick leaders, but this dependence increases complexity and causes performance bottlenecks. ZooKeeper restricts Kafka clusters to 200,000 partitions and slows down processes when brokers join or depart. Developers find it difficult because of these concerns. To solve this, Kafka is releasing KRaft, a new internal consensus system designed to replace ZooKeeper. KRaft promises increased scalability, simpler deployment, and improved performance, and it is presently in development with the goal of being ready for production shortly.

KAFKA AND ITS ALTERNATIVES

Due to its capabilities, Kafka can’t be compared with other technologies feature by feature. But, it has alternatives to consider when it comes to particular use cases.

Kafka vs Hadoop

Kafka is similar to Hadoop in that it stores and processes data on a big scale. Hadoop works with batches, but Kafka deals with streams, therefore the two are similar.However, these systems fulfill various functions. Hadoop is well-suited to intensive analytics applications that are not time-sensitive and generate insights for long-term planning and strategic decisions. Kafka is more focused on developing services that enable day-to-day commercial activities.

Kafka vs ETL

Kafka is commonly referred to as a speedier ETL. However, there are considerable variances between them. Extract-Transform-Loading technologies emerged as all software products interacted directly with databases. ETL technologies have always focused on moving data from one database to another. Today, more and more apps work with streams of events rather than data stores, which ETL cannot handle.
Kafka allows you to combine all data flows from various use cases. At the same time, ETL engineers may leverage the data streaming platform for workloads that require real-time transformations. As a result, an ETL operation becomes a Kafka producer or consumer.

Kafka vs MQ software

Messaging is one of Kafka's most popular use cases. As a result, it is frequently referred to as another message queue (MQ) platform, alongside IBM MQ, ActiveMQ, and RabbitMQ. However, Kafka is faster, more scalable, and can handle all of the company's data streams. Aside from that, Kafka can keep data for a long time and process streams in real time.

CODE AND OUTPUT

DEMO 1:

This code implements a real-time data streaming pipeline using Apache Kafka and Spark to predict heart disease risk. The pipeline includes pre-trained model and pre-processing steps like StringIndexer, OneHotEncoder and VectorAssembler, which were saved and reused for streaming real-time patient data and predicting outcomes efficiently.

INPUT:

```
(base) sarveshkrishnan@Sarveshs-MacBook-Pro ~ % kafka-console-producer --broker-
list localhost:9092 --topic heartinput
>{"age": 60, "sex": 1, "chest pain type": 4, "resting bp s": 140, "cholesterol":
 280, "fasting blood sugar": 0, "resting ecg": 0, "max heart rate": 172, "exerci
se angina": 0, "oldpeak": 2.5, "ST slope": 1}
```

CODE:

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


from pyspark.sql import SparkSession
from pyspark.sql.functions import col, from_json
from pyspark.sql.types import StructType, StructField, DoubleType,IntegerType

#Creating a Spark session with Kafka configuration
spark = SparkSession.builder \
    .appName("Kafka-Heart-Test") \
    .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.3") \
    .getOrCreate()

#Defining schema for incoming data
schema = StructType([
    StructField("age", DoubleType(), True),
    StructField("sex", DoubleType(), True),
    StructField("chest pain type", DoubleType(), True),
    StructField("resting bp s", DoubleType(), True),
    StructField("cholesterol", DoubleType(), True),
    StructField("fasting blood sugar", DoubleType(), True),
    StructField("resting ecg", DoubleType(), True),
    StructField("max heart rate", DoubleType(), True),
    StructField("exercise angina", DoubleType(), True),
    StructField("oldpeak", DoubleType(), True),
    StructField("ST slope", DoubleType(), True)
])
#Reading data from Kafka
kafka_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "heartinput") \
    .option("failOnDataLoss", "false") \
    .load()

#Printing the schema of kafka_df
kafka_df.printSchema()


# In[2]:

#Parsing the JSON data from Kafka
from pyspark.sql.functions import col, from_json
value_df = kafka_df.selectExpr("CAST(value AS STRING)as json_value")
parsed_df = value_df.select(from_json(col("json_value"), schema).alias("data")).select("data.*")
parsed_df.printSchema()
```

```
# In[3]:


# value_df.writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .option("truncate", "false") \
#     .start() \
#     .awaitTermination()


# In[4]:


# parsed_df.writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .option("truncate", "false") \
#     .start()


# In[5]:

#Apply pre-trained ML model for predictions
from pyspark.ml import PipelineModel
pipeline_model = PipelineModel.load("/Users/sarveshkrishnan/Desktop/termpaper/heartdisease/pipeline")

predicted = pipeline_model.transform(parsed_df)


# In[6]:


selected_predictions = predicted.select("age", "sex", "prediction")


# In[8]:


# selected_predictions.writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .option("truncate", "false") \
#     .start()


# In[9]:
```

```
# selected_predictions.writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .option("truncate", "false") \
#     .start()


# In[9]:
from pyspark.sql.functions import when, col

# Creating a new column with a descriptive result based on the prediction
output = selected_predictions.withColumn(
    "result",
    when(col("prediction") == 1, "Risk of Heart Disease")
    .otherwise("No Risk of Heart Disease")
)

# Streaming the results to Kafka
query = output.selectExpr("CAST(result AS STRING) AS value") \
    .writeStream \
    .outputMode("append") \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "heartoutput") \
    .option("checkpointLocation", "/Users/sarveshkrishnan/spark-checkpoints/sentimentoutput") \
    .start()

query.awaitTermination()
```

OUTPUT:

```
(base) sarveshkrishnan@Sarveshs-MacBook-Pro ~ % kafka-console-consumer --bootstrap-server localhost:9092 --topic heartoutput
Risk of Heart Disease
```

DEMO 2:

This code sets up a real-time sentiment analysis pipeline by reading input text data from Kafka, pre-processing it by loading a pipeline and loaded the pre-trained sentiment model in PySpark, and streaming the sentiment predictions to an output Kafka topic.

INPUT:

```
(base) sarveshkrishnan@Sarveshs-MacBook-Pro ~ % kafka-console-producer --broker-
list localhost:9092 --topic sentimentinput
>I used to watch this show when I was growing up. When I think about it, I remem
ber it pretty well. If you ask me, it was a pretty good show. Anytime I think ab
out it, I don't remember the opening sequence and theme song very well. In addit
ion to that, everyone was ideally cast. Also, the writing was very strong. The p
erformances were top-grade, too. I hope some network brings it back so I can see
 every episode. Before I wrap this up, I'd like to say that I'll always remember
 this show in my memory forever, even though I'm not sure if I've seen every epi
sode. Now, in conclusion, if some network ever brings it back, I hope that you c
atch it one day before it goes off the air for good.
>
```

CODE:

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.ml import PipelineModel
from pyspark.sql.functions import col, udf
from pyspark.sql.types import StringType

#Creating a Spark Session with Kafka configurations
spark = SparkSession.builder \
    .appName("KafkaSentimentAnalysis") \
    .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.0") \
    .getOrCreate()


# In[2]:

#Reading data from Kafka
kafka_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "sentimentinput") \
    .option("failOnDataLoss", "false") \
    .load()

# Convert the Kafka message key and value from bytes to strings
kafka_string_df = kafka_df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING) as review")


# In[3]:


# kafka_df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)").writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .start() \
#     .awaitTermination()


# In[4]:


```

```python
from pyspark.sql.functions import col

#Converting the Kafka message from bytes to strings (assuming the message is plain text)
parsed_df = kafka_df.selectExpr("CAST(value AS STRING) as review")

#Displaying the schema of the parsed_df
parsed_df.printSchema()

# Write the parsed DataFrame to the console
# parsed_df.writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .start() \
#     .awaitTermination()


# In[5]:


#Loading the preprocessing pipeline model
pipeline_model = PipelineModel.load("/Users/sarveshkrishnan/Desktop/termpaper/sentiment/preprocess")


# In[6]:


df=pipeline_model.transform(parsed_df)


# In[7]:


# df.writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .start()


# In[8]:

#loading the pretrained model
from pyspark.ml.classification import LogisticRegressionModel
loaded_model = LogisticRegressionModel.load("/Users/sarveshkrishnan/Desktop/termpaper/sentiment/saved_model")
```

```python

# In[9]:
final_predictions = loaded_model.transform(df)
# In[10]:


# final_predictions \
#     .writeStream \
#     .outputMode("append") \
#     .format("console") \
#     .start()


# In[11]:
from pyspark.sql.functions import when, col

# Creating a new column with a descriptive result based on the prediction
output = final_predictions.withColumn(
    "result",
    when(col("prediction") == 1, "Positive Sentiment")
    .otherwise("Negative Sentiment")
)

# Streaming the results to Kafka
query = output.selectExpr("CAST(result AS STRING) AS value") \
    .writeStream \
    .outputMode("append") \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("topic", "sentimentoutput") \
    .option("checkpointLocation", "/Users/sarveshkrishnan/spark-checkpoints/sentimentoutput") \
    .start()

query.awaitTermination()
```

OUTPUT:

```
(base) sarveshkrishnan@Sarveshs-MacBook-Pro ~ % kafka-console-consumer --bootstrap-server localhost:9092 --topic sentimentoutput
Negative Sentiment
```