TERM PAPER
APACHE KAFKA
MET CS777 - BIG DATA ANALYTICS

INTRODUCTION:

Apache Kafka is an open-source distributed event streaming platform that acts as the backbone for data flow in many modern systems. It enables various applications to send, receive, store, and process data in real-time. I chose to focus on Kafka because it plays a crucial role in supporting applications requiring continuous and instantaneous data movement. For instance, social media platforms provide live feeds, banking systems track and manage transactions instantly, and Internet of Things (IoT) devices continuously generate and transmit data. In all these scenarios, businesses depend on processing large streams of data without delays, and this is where Kafka excels.

Kafka was designed to be scalable, durable, and fault-tolerant, making it essential for handling today's high data volumes and speeds.

Why Kafka for Big Data Analytics?

In the big data landscape, where companies collect and analyze massive amounts of information, speed and reliability are essential. Kafka fits seamlessly because it can handle large data streams while maintaining fault tolerance. Its distributed server architecture allows horizontal scaling across multiple servers, enabling organizations to manage growing data loads without sacrificing performance. This scalability is a key advantage in big data analytics, where both volume and speed can be enormous.

Kafka ensures data consistency and durability by replicating data across multiple servers, reducing the risk of data loss if a server fails. These features make Kafka highly reliable for real-time analytics and large-scale data processing, where maintaining continuous data flow and quick decision-making is crucial.

Why is Kafka Essential for Real-Time Data Streaming and Log Processing?

One of Kafka's main strengths is its ability to support real-time data streaming, capturing and delivering data as soon as it's generated. This capability is crucial for applications like fraud detection, which rely on instantaneous responses, or recommendation engines that deliver personalized content. Kafka's design ensures a continuous flow of data between producers (applications that send data) and consumers (applications that receive and process data). The asynchronous nature of this flow allows producers to keep sending data even if some consumers are temporarily offline, making Kafka highly efficient and resilient.

Moreover, Kafka plays a critical role in log processing, collecting system logs securely and making them available for analysis. This centralized log management helps organizations monitor systems in real-time, identify issues quickly, and analyze patterns.

Main Components of Kafka:

- Producers: Applications that send data.
- Consumers: Applications that read data.
- Brokers: Servers that manage and store data.
- Topics and Partitions: Categories where data is organized and divided for easy handling.

BACKGROUND AND MOTIVATION

As data generation accelerates, businesses need to react and make decisions in real-time to stay competitive. Whether it's social media platforms reacting to trending topics, financial services monitoring transactions for fraud, or retailers managing inventory updates instantly, real-time data processing enables organizations to respond immediately to changing conditions and user actions. Without it, businesses risk missing key opportunities or reacting too slowly to critical situations.

The Problem Kafka Aims to Solve and Its History:

Before Kafka, traditional systems struggled to handle large-scale, real-time data. As LinkedIn grew, it faced this problem and needed a system to manage massive data volumes in real-time across its global platform. In response, LinkedIn's engineers created Kafka in 2010, designing it to be highly scalable, fault-tolerant, and capable of handling high-throughput, low-latency messaging. By making Kafka an open-source project in 2011 under the Apache Software Foundation, LinkedIn enabled other organizations to adopt and expand the platform.

Since then, Kafka has evolved into a widely-used solution for various real-time use cases, such as event streaming, log aggregation, and real-time analytics. Its efficiency and reliability in handling real-time data processing have driven its popularity.

KAFKA ARCHITECTURE

At its core, Kafka's architecture is designed to handle large volumes of data efficiently and reliably. The main components include producers, consumers, brokers, topics, and partitions:

1. Producers: These are the applications or systems that generate and send data to Kafka. For example, a weather sensor sending temperature data, a website generating clickstream data, or a financial application sending transaction records. The role of producers is to push data into Kafka for further processing or storage. They send this data to specific topics in Kafka (discussed later below).

2. Consumers: These are applications that read and process the data stored in Kafka. They act as the receivers of information. For instance, an analytics application that processes web traffic data to understand user behaviour, or a monitoring system that checks server logs to detect errors. Consumers they choose which sets of data they want to read and

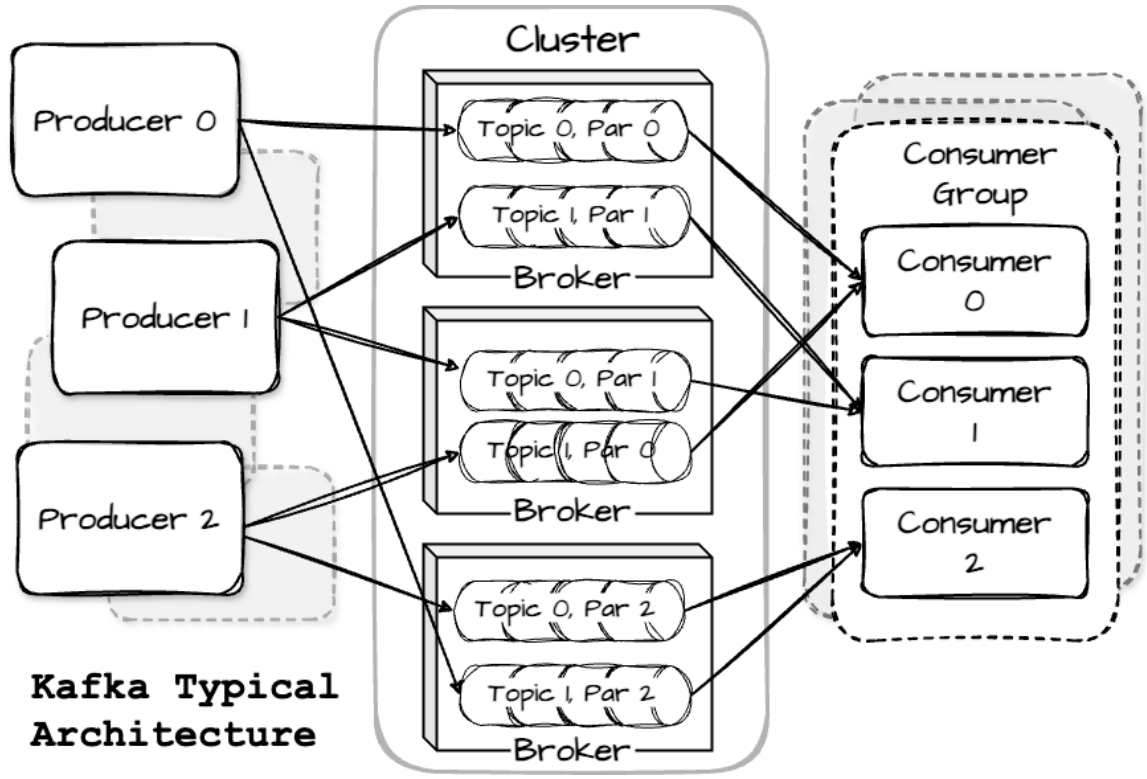
process from Kafka. A single topic can have multiple consumers, each doing different things with the data.

3. Brokers: These are servers that run Kafka and handle the work of storing and managing data. They act like the middlemen between producers and consumers. A broker receives data from producers, stores it securely, and allows consumers to access it whenever they need. Kafka clusters consist of multiple brokers working together. This setup helps to distribute data efficiently and provides fault tolerance (continuous operation even if a broker fails).

4. Topics: These are categories or channels where data is organized. Producers send data to specific topics, like "sales_data" for tracking sales information. Topics allow consumers to access only the relevant data streams.

5. Partitions: Each topic is divided into smaller segments called partitions. Partitions allow data to be spread across multiple servers, enabling parallel processing and improving performance. Additionally, each partition is replicated across brokers, ensuring data availability even if one broker fails.

   By organizing data into topics and partitions, Kafka's architecture enhances scalability, fault tolerance, and efficiency, making it ideal for real-time data streaming and large-scale processing.



Kafka Typical Architecture

Zookeeper's Role:

Zookeeper is an external system that manages and coordinates Kafka's brokers, acting like Kafka's "manager." It keeps track of available brokers, assigns roles, and monitors their health. Zookeeper helps elect leader brokers for each partition, stores metadata about brokers, topics, and partitions, and ensures the system runs smoothly by triggering failover actions if a broker goes down.

However, Kafka is gradually transitioning to KRaft mode (Kafka Raft) to replace Zookeeper. KRaft simplifies Kafka's architecture by handling metadata and cluster coordination internally, improving scalability and reliability without relying on an external system. This shift enhances Kafka's overall efficiency in newer versions.

KEY FEATURES AND BENEFITS:

Apache Kafka offers several key features that make it reliable, efficient, and ideal for handling real-time data streams:

1. Durability: Kafka securely stores data by writing incoming messages to a commit log on disk, ensuring they remain intact even during system failures. With data replication across multiple brokers, Kafka guarantees data availability and reliability, making it suitable for critical applications like financial transactions and system monitoring.

2. Fault Tolerance: Kafka's distributed architecture replicates each partition across multiple brokers, allowing it to continue operating smoothly even if some brokers fail. In case of a failure, Kafka automatically elects a new leader broker, ensuring minimal downtime and continuous service.

3. Scalability: Kafka scales efficiently by distributing data across multiple brokers and dividing topics into smaller segments called partitions. This horizontal scalability allows Kafka to handle increasing data volumes and user loads without compromising performance, making it ideal for growing businesses and fluctuating data demands.

4. High Throughput: Kafka's design prioritizes high message processing rates with minimal latency. By utilizing sequential disk I/O and asynchronous processing, Kafka efficiently handles millions of events per second, avoiding bottlenecks and maintaining smooth data flow even under heavy loads.

This combination of durability, fault tolerance, scalability, and high throughput makes Kafka a dependable choice for large-scale, real-time data processing.

COMPARISION WITH OTHER TECHNOLOGIES:

Apache Kafka, RabbitMQ, and ActiveMQ all serve as messaging systems, but they differ in their design and use cases. RabbitMQ and ActiveMQ are traditional message brokers, focusing on reliable message delivery and complex routing. Both use a message queue model

to manage tasks and ensure that messages are delivered correctly. RabbitMQ is known for its strong routing capabilities using exchanges and bindings, making it a popular choice for scenarios where precise message delivery is critical, such as in managing real-time tasks between microservices. ActiveMQ, on the other hand, is often chosen for enterprise-level deployments, especially in Java-based environments, due to its support for the JMS (Java Message Service) protocol. Both systems are effective for handling request-response communications, task queues, and applications requiring guaranteed message delivery.

In contrast, Kafka operates on a log-based architecture designed for large-scale event streaming and real-time analytics. Unlike RabbitMQ and ActiveMQ, which focus on short-lived tasks and reliable delivery, Kafka excels in scenarios where data needs to be processed continuously and stored for future use. Kafka's ability to retain and replay messages, combined with its distributed architecture, allows it to handle high-throughput scenarios and scale horizontally by adding more brokers and partitions. This makes Kafka ideal for applications such as real-time monitoring, log aggregation, and large-scale data integration, where multiple consumers need access to the same data stream over time. While traditional brokers like RabbitMQ and ActiveMQ are well-suited for managing reliable messaging and routing, Kafka's design caters to streaming and data-intensive applications that require scalability, high throughput, and long-term message retention.

DEMOS:

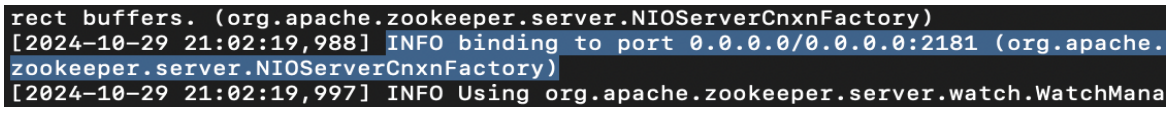
Demo 1

In this demonstration, I set up a simple Kafka system to show how data can be produced, processed, and consumed in real time. This demo highlights basic functionalities such as topic creation, message production, and consumption, showcasing Kafka's strength in real-time data processing. Below are the steps of how I achieved it.

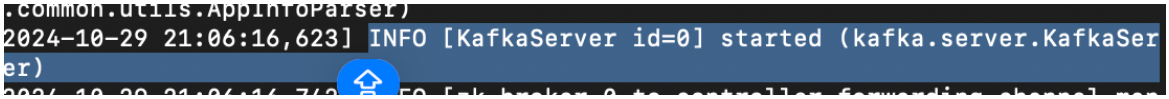
1. Downloaded Kafka and set up a directory.
   cd /Users/snehaagrawal/Downloads/kafka-3.8.0-src

2. Built Kafka using Gradle command to ensure all necessary components were ready.
   ./gradlew jar -PscalaVersion=2.13.14

3. Started Zookeeper that handles the functionality of Kafka
   bin/zookeeper-server-start.sh config/zookeeper.properties



4. Started the Kafka Server (Broker)
   bin/kafka-server-start.sh config/server.properties



5. Created 2 Kafka Topics with 3 partitions each, input-topic and output-topic, to store and organize messages.
   bin/kafka-topics.sh --create --topic input-topic --bootstrap-server localhost:9092 --partitions 3 --replication-factor 1
   bin/kafka-topics.sh --create --topic output-topic --bootstrap-server localhost:9092 --partitions 3 --replication-factor 1

6. To check Kafka Broker Status, verifying if all the topics were accessible and active.
   bin/kafka-topics.sh --list --bootstrap-server 127.0.0.1:9092

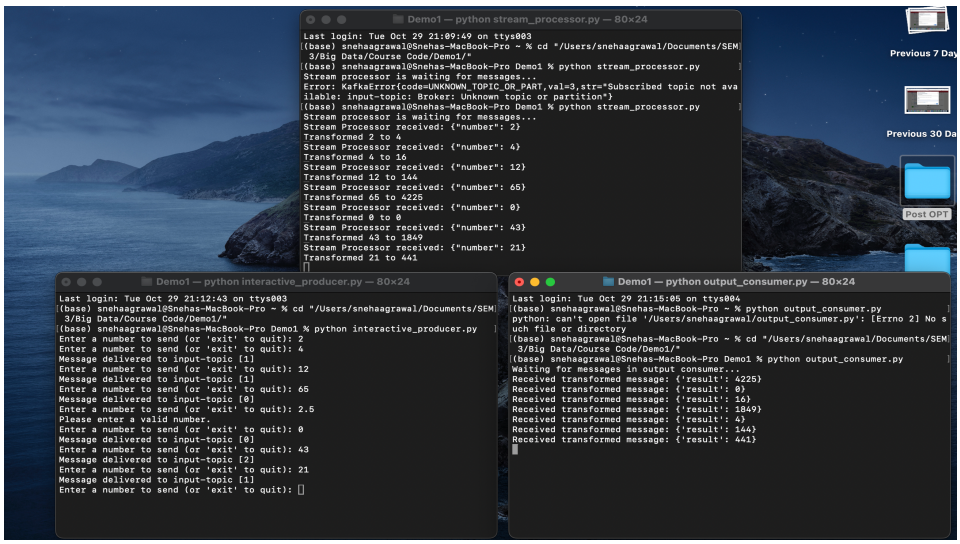
```
input-topic
input_file
input_topic
output-topic
output_file
output_topic
(base) snehaagrawal@Snehas-MacBook-Pro kafka-3.8.0-src %
```

7. Stream Processing using Python: A python script, stream_processor.py, to process incoming messages by reading from the input-topic topic, transforming the data, and sending it to the output-topic topic. (script using confluent_kafka for processing messages).
   cd "/Users/snehaagrawal/Documents/SEM 3/Big Data/Course Code/Demo1/"
   python stream_processor.py

8. Creating a Producer Console: A python script, interactive_producer.py, to allow users to input numbers and send them as messages to the input_file topic.
   python interactive_producer.py

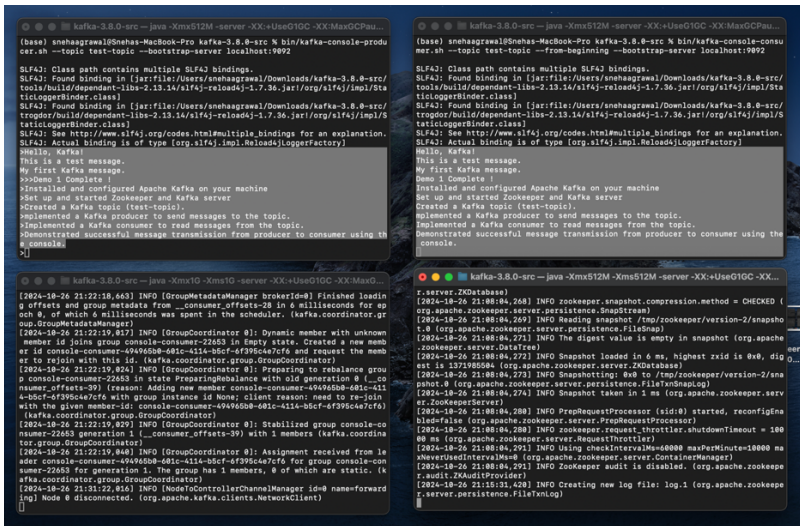
9. Creating a Consumer Console: A python script, output_consumer.py, to consume messages from the output_file topic and display the transformed results.
   python output_consumer.py

10. Finally, running all terminal parallelly I was able to demonstrate how messages entered into the producer were processed in real-time and displayed in the consumer console. This verified that the messages produced were successfully processed and consumed.



This demonstration provided a basic yet practical implementation of Kafka, illustrating its core architecture and real-time capabilities. By setting up producers, consumers, and processing streams, the demo showcased Kafka's strengths in data streaming and scalability, laying the groundwork for more advanced use cases.

The Producer and Consumer Console to test live:



Producer: bin/kafka-console-producer.sh --topic test-topic --bootstrap-server localhost:9092
Consumer: bin/kafka-console-consumer.sh --topic test-topic --from-beginning --bootstrap-server localhost:9092

Demo 2

In this demonstration, I created a real-time user management system using Apache Kafka, PostgreSQL, and Streamlit. The system supports user registration and login, while Kafka handles event tracking, and PostgreSQL stores user data and login events. Below are the steps and details of the implementation:

Necessary libraries: pip install kafka-python psycopg2-binary bcrypt streamlit

1. Kafka Topic to send and receive data:
   bin/kafka-topics.sh --create --topic login-events --bootstrap-server localhost:9092 --partitions 3 --replication-factor 1

2. Configuring PostgreSQL: Connected to a PostgreSQL database using Python's psycopg2 library. The database is used to store user information and events.

```
# PostgreSQL setup
try:
    conn = psycopg2.connect(
        dbname="postgres",  # Database where the tables are created
        user="postgres",
        password="password123",
        host="localhost",
        port="5433"
    )
    cursor = conn.cursor()
    print("Connected to the database successfully!")  # Debugging information
```

3. Create a Kafka Producer: Created a Kafka producer in Python to send events like incorrect login attempts and new user creation to the "login-events" topic.

```
# Create Kafka producer
producer = KafkaProducer(
    bootstrap_servers=KAFKA_BROKER,
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)
```

4. Implemented functions to validate usernames and passwords, handle user registration, and track events. This includes:
- Checks if the username is between 4-15 characters long and contains only letters and numbers.
- Ensures the password contains at least 8 characters, including an uppercase letter, lowercase letter, number, and special character.

5. Implementing the User Registration and Login System:
- Created a registration function to store new users in PostgreSQL and send a "new_user_created" event to the Kafka topic:
  def register_user(username, password):
- Developed a login function that verifies the user's credentials and sends an "incorrect_password" event to Kafka if the login fails:
  def login(username, password):

6. Creating the Streamlit UI: Designed a user-friendly interface using Streamlit to allow users to register and log in.

```
# Streamlit UI
st.title("User Registration and Login System with Kafka and PostgreSQL Integration")

# Sidebar Navigation
page = st.sidebar.selectbox("Choose Action", ["Login", "Register"])
```

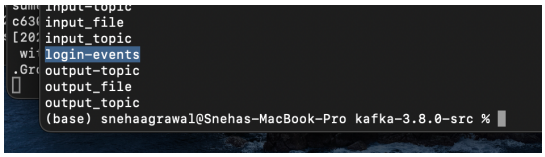
How to run?

cd /Users/snehaagrawal/Downloads/kafka-3.8.0-src

1. Starting Zookeeper: bin/zookeeper-server-start.sh config/zookeeper.properties

2. Startting Kafka Server: bin/kafka-server-start.sh config/server.properties

3. Creating a topic named login-events:
bin/kafka-topics.sh --create --topic login-events --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
bin/kafka-topics.sh --list --bootstrap-server localhost:9092



4. Preparing the Python Environment
   python3 -m venv kafka-env
   source kafka-env/bin/activate

5. pip install streamlit kafka-python bcrypt

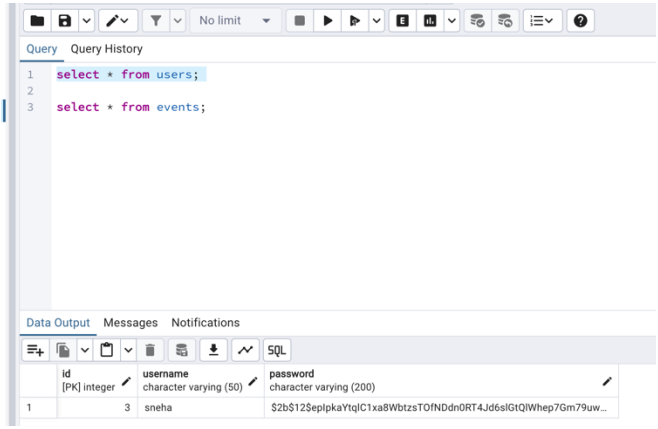
6. Consumer console:
   bin/kafka-console-consumer.sh --topic login-events --from-beginning --bootstrap-server localhost:9092

7. Python code
   cd "/Users/snehaagrawal/Documents/SEM 3/Big Data/Course Code/Demo 2/"
   streamlit run app.py

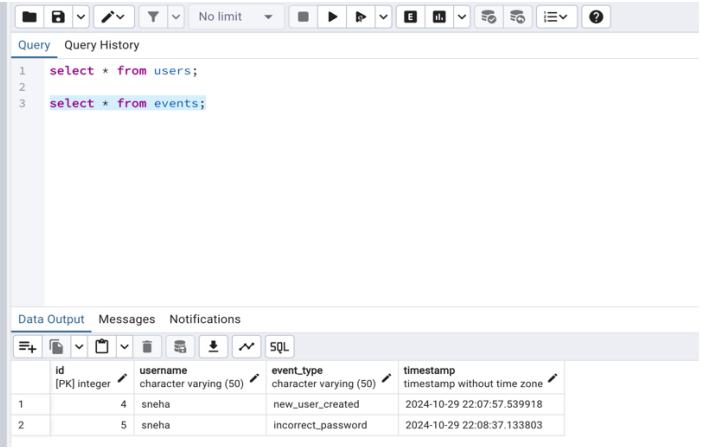
8. Testing and Demonstrating the System

9. In PostgreSQL to see results:

- To see table:
  select * from users;



- To see events:
  select * from events;



Observations:

- The system successfully tracked and sent user-related events to the Kafka topic "login-events", highlighting Kafka's real-time streaming capabilities.
- PostgreSQL was used effectively to store and manage user data and event logs.
- The integration of Streamlit, Kafka, and PostgreSQL provided a complete solution for managing and monitoring user activities.

This demonstration showed how Kafka can be leveraged for real-time event tracking in a user management system. By using Kafka, PostgreSQL, and Streamlit, the demo illustrated Kafka's ability to seamlessly handle event streams and integrate with external systems like databases. This setup can be extended to more complex applications that require real-time monitoring and data processing capabilities.

LIMITATIONS AND CHALLENGES:

While Apache Kafka is powerful, it presents some key challenges that require attention:

1. Cluster Management Complexity: Managing Kafka is challenging because it involves several components like brokers (the servers), partitions (divisions of data within topics), and topics (data categories). Successfully scaling Kafka and balancing the load between these components requires a good understanding of distributed systems and continuous monitoring to keep everything running smoothly.

2. Exactly-Once Semantics: Kafka offers a feature to ensure that each message is delivered and processed only once. However, achieving this consistently across an entire data flow can be difficult. Integrating Kafka with other systems, like databases, requires careful setup to avoid duplicate messages or data inconsistencies.

3. Latency vs. Throughput: Kafka is optimized to process large amounts of data quickly (throughput), but it does this by grouping messages and writing them to disk in sequence. While this increases efficiency, it can cause minor delays (latency), which might be an issue for applications that require immediate data delivery.

4. Data Storage Management: Kafka stores data on disk, which means that managing storage space is crucial. Setting up proper data retention policies and monitoring disk usage is important, especially when dealing with large volumes of data. Without careful planning, storage costs can rise, or critical data might be deleted unintentionally.

These challenges emphasize the need for thoughtful planning and expertise when deploying Kafka to ensure its effective use in real-time streaming environments.

APPLICATIONS AND USE CASES:

Apache Kafka is utilized by leading companies across various industries due to its real-time data streaming capabilities. For instance, LinkedIn, which initially developed Kafka, uses it to track billions of user activities daily, enabling real-time monitoring, personalized content delivery, and valuable insights into platform performance. Netflix is another key user of Kafka, employing it within its monitoring infrastructure to gather and analyze billions of data points from microservices, allowing for seamless video streaming and rapid issue detection and resolution.

In the music streaming industry, Spotify leverages Kafka to process vast amounts of real-time event data, such as song streams, user interactions, and playlist updates. This enables Spotify to deliver personalized music recommendations and maintain real-time analytics on listener behaviour. Amazon, on the other hand, relies on Kafka for real-time order tracking, inventory updates, and customer interactions. By utilizing Kafka, Amazon can provide accurate order statuses, manage inventory efficiently, and optimize its logistics operations at a massive scale.

In the finance sector, large banks use Kafka for high-volume transaction monitoring and fraud detection. Kafka helps process millions of transactions per second, allowing banks to identify suspicious patterns and prevent fraud in real-time. Similarly, in social media, Twitter relies on Kafka to handle vast streams of user-generated content, enabling it to identify trending topics quickly and tailor feeds to users' preferences. These examples showcase how Kafka's real-time processing, scalability, and durability make it essential for enhancing efficiency and reliability in diverse applications.

CONCLUSION:

Apache Kafka has established itself as a vital technology for handling real-time data streaming and large-scale event processing. It serves as a backbone for modern applications, enabling seamless and efficient data flow across multiple systems. With its scalable, durable, and fault-tolerant architecture, Kafka excels in managing massive data streams in real-time, making it indispensable for businesses across various sectors.
From its initial development at LinkedIn to its widespread adoption by companies like Netflix, Spotify, Amazon, and major financial institutions, Kafka's flexibility and efficiency in data integration and streaming have proven crucial for enhancing operational efficiency and delivering personalized user experiences. Despite its challenges, such as the complexity of managing clusters and achieving exactly-once semantics, Kafka's benefits in high-throughput processing and scalable architecture outweigh these limitations.

Overall, Apache Kafka is not just a messaging system; it is a comprehensive data streaming platform that meets the demands of modern big data applications. Its ability to process, store, and distribute data reliably and in real-time makes it a powerful choice for businesses aiming to stay competitive in a data-driven world. With ongoing improvements like the introduction of KRaft mode to replace Zookeeper, Kafka continues to evolve, becoming even more efficient and scalable for future data needs.