Real-Time Log Data Streaming Using Kafka

Adv Database Final Project

Nake Aparanji
Computer Science Name
University of Memphis
Memphis TN USA
nspranji@memphis.edu

Rakesh Reddy
Computer Science
University of Memphis
Memphis TN USA
rkatika@memphis.edu

ABSTRACT

Log data plays a crucial role in understanding the health and performance of software systems. As applications generate vast amounts of log entries, the need for efficient log data analysis becomes paramount. This project focuses on the real-time streaming and analysis of log data using Apache Kafka and Java. A simulated log data generator serves as the producer, sending data to a Kafka server. On the consumer side, a Java application retrieves and analyzes the log entries in real time, providing insights into system behavior. Additionally, a Spring Boot project integrates the consumer logic, displaying live log data on a webpage. This project aims to showcase the significance of log data analysis for monitoring and optimizing software systems.

1 Introduction

In the dynamic landscape of software applications, understanding how they perform is crucial. Log data, which consists of records generated by these applications, serves as a valuable source of insights into their behavior. As organizations face the challenge of handling increasingly large volumes of log data, the need for efficient analysis becomes more pronounced.

1.1 Motivation

The motivation behind our Kafka-based application stems from the necessity to adeptly manage and analyze simulated log data in real-time. With the rising tide of digital operations, organizations are inundated with vast amounts of log data. The task at hand is to extract meaningful and timely insights from this sea of information. Our project seeks to address the challenges associated with processing and analyzing massive log data, aspiring to offer a scalable and responsive solution for real-time data analysis.

1.2 Technical Challenges

1.2.1 Getting Log Data. Obtaining relevant log data posed an initial challenge. While some APIs did offer real-time log data, they often came at a cost. Datasets available on platforms like Kaggle existed but were not sufficiently large for our needs. Consequently, we decided to generate simulated log data, providing a controlled and scalable approach for our project.

1.2.2 Setting Up Kafka. The choice of the operating system presented a decision point. Linux, renowned for its performance and stability, is the preferred environment for Kafka in production. However, considering the educational nature of our project, we opted for a Windows environment. Despite occasional challenges like server startup issues and disconnections from Zookeeper in Windows, which were mitigated by restarting, Windows was the better option due to its simplicity.

Another decision revolved around the choice between Apache Kafka and Confluent Kafka. While Confluent Kafka offers additional features, Apache Kafka was deemed sufficient for our educational project, focusing on fundamental learning without unnecessary complexities.

Additionally, the decision between Zookeeper and KRaft came into play. Although newer versions of Kafka use KRaft to eliminate Zookeeper as scaling issues emerge when clusters have over 100,000 partitions (in Zookeper), we felt Zookeeper is still a good skill to master. Since our project also did not require such extensive scaling, we chose Zookeeper for its simplicity and compatibility.

1.2.3 Integrating Consume. The integration of the consumer logic into a Spring Boot project involved the design of a service/controller class. This class serves as the bridge connecting the consumer logic and the HTML view template, enabling the seamless presentation of live log data on a webpage. This integration step was essential to showcase the real-time streaming of log data to end-users.

2 Related Work

We found a simple Kafka application that is integrated with Spring Boot framework on GitHub, and this project's problem closely aligns with our project's problem statement, i.e., streaming real-time data using Kafka on a webpage. In the case from the project we found, the Kafka topic is created with similar configurations as ours, but we only have 1 topic.

Our project differs in the way that we are calculating the said metrics along with streaming data. We are also doing some basic analysis before displaying the log data. Also, the project we found saved the offsets at the end of the consumer code but we did not seem it was necessary due to the presence of just 1 topic. Our Spring Boot application differs vastly from theirs. We have taken help of 'Service' component while linking the consumer logic and HTML page. We used the Apache Kafka Spring dependency to effectively retrieve log data and display it. This the GitHub link to the project: https://github.com/contactsunny/SimpleKafkaExampleSpringBoot/tree/master

3 Solution

3.1 Solution Path – Why Kafka

In our quest to find a robust solution for handling log data, we turned to Apache Kafka for several compelling reasons. Kafka stands out for its ability to address critical aspects of data management, making it an ideal choice for our project.

3.1.1 Scalability. Scalability refers to a system's capability to handle an increasing amount of work, and Kafka excels in this area. It allows us to scale both horizontally, by adding more machines to a Kafka cluster, and vertically, by increasing the capacity of individual machines. This ensures that our solution can effortlessly accommodate growing volumes of log data without compromising performance.

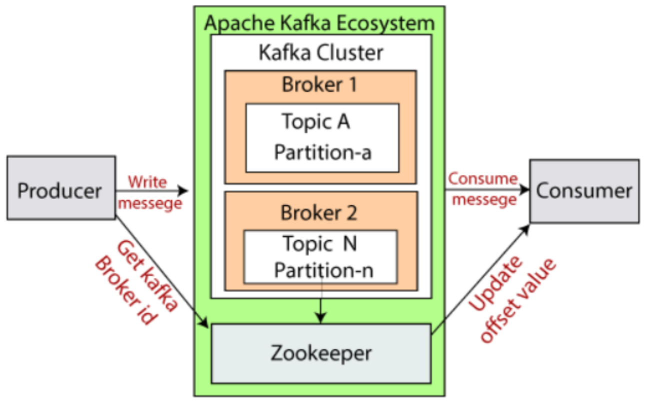
3.1.2 Durability Up Kafka. Durability is crucial for preserving data integrity and ensuring that no information is lost, especially in scenarios where log data is pivotal. Kafka achieves durability through its distributed architecture and replication mechanism. Data is replicated across multiple Kafka brokers (servers), providing fault tolerance. Even if a broker goes down, the data remains accessible, ensuring the reliability of our log data.

3.1.3 Real-time Processing. Real-time processing involves handling and analyzing data as it arrives, enabling timely insights. Kafka's design revolves around supporting real-time data streaming. It allows producers to send data in real-time, and consumers can retrieve and process this data as soon as it's available. The concept of topics and partitions further enhances parallel processing, making Kafka a powerful tool for real-time data scenarios.

In summary, Kafka's scalability, durability, and real-time processing capabilities align perfectly with the challenges posed by handling and analyzing log data in our project. The decision to leverage Kafka as the backbone of our solution provides a robust foundation for effective and efficient log data management.

3.2 Kafka Architecture

At its core, Apache Kafka serves as an event streaming platform, allowing us to publish (write) and subscribe to (read) streams of events. It's designed for storing and processing these streams efficiently.



Apache Kafka Architecture

3.2.1 Kafka as a Distributed System. Kafka operates as a distributed system, with servers/brokers communicating via TCP. These servers, known as brokers, form a cluster that can span across data centers. Some brokers handle storage, while others run the Kafka Connect API. This distributed architecture ensures high scalability and fault tolerance.
Kafka clients enable the creation of distributed applications and microservices. They empower applications to read, write, and process streams of events in parallel.

3.2.2 Kafka Events. When reading or writing data to Kafka, it's done in the form of events. Each event consists of a key, value, timestamp, and optional meta headers.
Producers and Consumers: Producers write or publish events to Kafka, while consumers subscribe to or read these events. Events are stored in topics, forming a structured way to organize and manage the flow of data.

3.2.3 Kafka Topics. A Kafka topic represents a particular stream of data, similar to a table in a database. Topics are identified by names and can accommodate various message formats.
Partitions: Topics are split into partitions (e.g., 100 partitions), where each partition contains an ordered sequence of messages. Messages within a partition receive incremental IDs called offsets.
Immutability: Kafka topics are immutable, meaning that once data is written to a partition, it cannot be changed. Data is retained for a limited time, typically configurable.

3.2.4 Kafka Consumers. Data Reading: Consumers pull data from topics in a specific order, reading from low to high offset within each partition. Deserialization transforms bytes into objects or data.
Consumer Groups: Consumers within an application read data as a group, each reading from exclusive partitions. Multiple consumer groups can exist on the same topic.

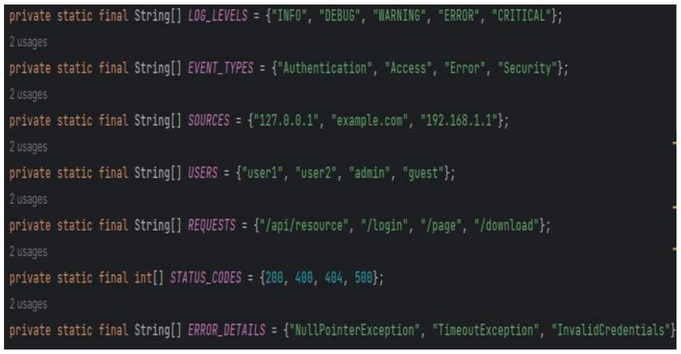
3.2.5 Kafka Producers. Responsibility: Producers write data to topics, considering partitioning to distribute the load across brokers. They can send a key with the message for targeted delivery.
Message Structure: A Kafka message comprises a key (binary format), value (binary format), compression type, headers, partition, offset, and timestamp. Serialization transforms objects/data into bytes.

3.2.6 Kafka Brokers and Replication. Cluster Composition: A Kafka cluster consists of multiple brokers, each identified by an ID. Brokers connect to form the entire cluster.
Replication Factor: Topics should have a replication factor > 1 (usually between 2 and 3) for data durability. If a broker goes down, another can serve the data. involves handling

3.2.7 Zookeeper. Role: Zookeeper manages brokers, aids in leader election for partitions, and notifies Kafka of changes. In Kafka 2.x, it's essential, but Kafka 3.x and beyond can operate without Zookeeper.
Design: Zookeeper operates with an odd number of servers (1, 3, 5, and 7), with one leader and the rest as followers. It doesn't store consumer offsets.

In our Kafka project, the architecture revolves around the core components of Apache Kafka. We have a Zookeeper instance and a Kafka server running, forming the backbone of our distributed system. Leveraging Zookeeper for management tasks and leader election, our Kafka server operates within the constraints of our laptop's memory. For the purpose of this project, we created a Kafka topic named "log-data-topic" with specific configurations. Due to memory limitations, we opted for a single partition and a replication factor of one during the topic creation process. This decision ensures that the Kafka cluster is well-suited for our development environment, allowing us to effectively simulate log data streaming scenarios while accommodating the constraints of our local setup.

3.3 Log Data – Dataset



```java
private static final String[] LOG_LEVELS = {"INFO", "DEBUG", "WARNING", "ERROR", "CRITICAL"};
2 usages
private static final String[] EVENT_TYPES = {"Authentication", "Access", "Error", "Security"};
2 usages
private static final String[] SOURCES = {"127.0.0.1", "example.com", "192.168.1.1"};
2 usages
private static final String[] USERS = {"user1", "user2", "admin", "guest"};
2 usages
private static final String[] REQUESTS = {"/api/resource", "/login", "/page", "/download"};
2 usages
private static final int[] STATUS_CODES = {200, 400, 404, 500};
2 usages
private static final String[] ERROR_DETAILS = {"NullPointerException", "TimeoutException", "InvalidCredentials"}
```

Values for each parameter are randomly chosen during the log entry generation process. This randomness introduces variability and diversity into the simulated log data, mimicking the unpredictability of real-world log events.

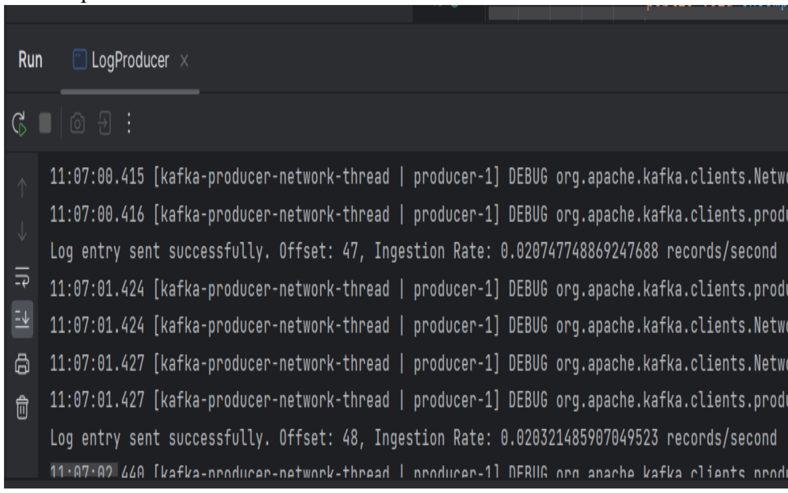
3.4 Spring Boot Integration

Our project extends its capabilities by leveraging Spring Boot and Thymeleaf to present log data results on a user-friendly web page. To achieve this integration, we incorporated the Apache Kafka Spring dependency, providing seamless connectivity between our Kafka-based application and the Spring ecosystem. The heart of the integration lies in a dedicated controller class that encapsulates the consumer logic. This controller class orchestrates the retrieval of log data from Kafka and serves as a bridge to the HTML template, facilitating the dynamic display of real-time log information.

Thymeleaf, a modern server-side Java template engine for web and standalone environments, enhances the HTML template, enabling the effortless rendering of log data results on the web page. Through this Spring Boot and Thymeleaf integration, our project achieves a cohesive and accessible user interface, allowing users to visualize and comprehend log data insights with ease.

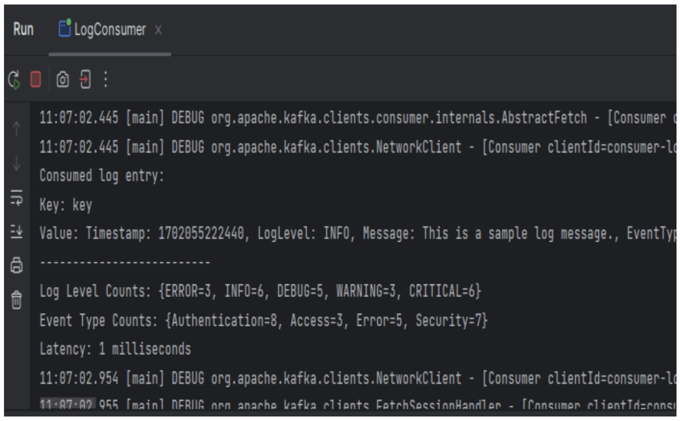
4 Evaluation

The evaluation of our Kafka-based log data streaming solution involves running both the producer and consumer simultaneously to ensure a comprehensive assessment of the system's performance. The producer's console output provides crucial information such as the successful sending of log data, displaying key-value pairs, and presenting the ingestion rate—measured in records per second.



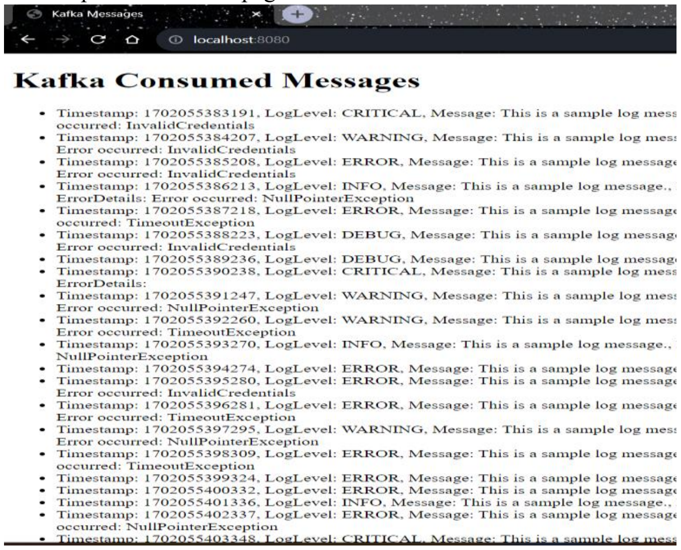
```
11:07:00.415 [kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.Netw
11:07:00.416 [kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.prod
Log entry sent successfully. Offset: 47, Ingestion Rate: 0.020747748869247688 records/second
11:07:01.424 [kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.prod
11:07:01.424 [kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.Netw
11:07:01.427 [kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.Netw
11:07:01.427 [kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.prod
Log entry sent successfully. Offset: 48, Ingestion Rate: 0.020321485907049523 records/second
11:07:02.440 [kafka-producer-network-thread | producer-1] DEBUG org.apache.kafka.clients.prod
```

On the other hand, the consumer output can be observed either on the console or a dedicated web page. The log data is presented, including basic analyses such as log level counts and event type counts. Additionally, the latency, representing the time taken to retrieve data from the Kafka server, is displayed.



```
11:07:02.445 [main] DEBUG org.apache.kafka.clients.consumer.internals.AbstractFetch - [Consumer c
11:07:02.445 [main] DEBUG org.apache.kafka.clients.NetworkClient - [Consumer clientId=consumer-lo
Consumed log entry:
Key: key
Value: Timestamp: 1702055222440, LogLevel: INFO, Message: This is a sample log message., EventTyp
------------------------
Log Level Counts: {ERROR=3, INFO=6, DEBUG=5, WARNING=3, CRITICAL=6}
Event Type Counts: {Authentication=8, Access=3, Error=5, Security=7}
Latency: 1 milliseconds
11:07:02.954 [main] DEBUG org.apache.kafka.clients.NetworkClient - [Consumer clientId=consumer-lo
11:07:02.955 [main] DEBUG org.apache.kafka.clients.FetchSessionHandler - [Consumer clientId=consu
```

The output on the web page looks like in the screenshot below:



Kafka Consumed Messages

```
• Timestamp: 1702055383191, LogLevel: CRITICAL, Message: This is a sample log mess
  occurred: InvalidCredentials
• Timestamp: 1702055384207, LogLevel: WARNING, Message: This is a sample log mess
  Error occurred: InvalidCredentials
• Timestamp: 1702055385208, LogLevel: ERROR, Message: This is a sample log message
  Error occurred: InvalidCredentials
• Timestamp: 1702055386213, LogLevel: INFO, Message: This is a sample log message.,
  ErrorDetails: Error occurred: NullPointerException
• Timestamp: 1702055387218, LogLevel: ERROR, Message: This is a sample log message
  occurred: TimeoutException
• Timestamp: 1702055388223, LogLevel: DEBUG, Message: This is a sample log messag
  Error occurred: InvalidCredentials
• Timestamp: 1702055389236, LogLevel: DEBUG, Message: This is a sample log messag
• Timestamp: 1702055390238, LogLevel: CRITICAL, Message: This is a sample log mess
  ErrorDetails:
• Timestamp: 1702055391247, LogLevel: WARNING, Message: This is a sample log mess
  Error occurred: NullPointerException
• Timestamp: 1702055392260, LogLevel: WARNING, Message: This is a sample log mess
  Error occurred: TimeoutException
• Timestamp: 1702055393270, LogLevel: INFO, Message: This is a sample log message.,
  NullPointerException
• Timestamp: 1702055394274, LogLevel: ERROR, Message: This is a sample log message
• Timestamp: 1702055395280, LogLevel: ERROR, Message: This is a sample log message
  Error occurred: InvalidCredentials
• Timestamp: 1702055396281, LogLevel: ERROR, Message: This is a sample log message
  Error occurred: TimeoutException
• Timestamp: 1702055397295, LogLevel: WARNING, Message: This is a sample log mess
  Error occurred: NullPointerException
• Timestamp: 1702055398309, LogLevel: ERROR, Message: This is a sample log message
  occurred: TimeoutException
• Timestamp: 1702055399324, LogLevel: ERROR, Message: This is a sample log message
• Timestamp: 1702055400332, LogLevel: ERROR, Message: This is a sample log message
• Timestamp: 1702055401336, LogLevel: INFO, Message: This is a sample log message.,
• Timestamp: 1702055402337, LogLevel: ERROR, Message: This is a sample log message
  occurred: NullPointerException
• Timestamp: 1702055403348, LogLevel: CRITICAL, Message: This is a sample log mess
```

The hardware/software configuration for our experiments includes running the solution on a laptop with defined memory constraints. We benchmark the solution by analyzing the ingestion rates, latency, and the accuracy of basic log data analysis.

Our solution exhibits effective performance, with ingestion rates consistently ranging between 0.1 and 0.9 records per second. This range aligns with our expectations, reflecting a steady and reliable flow of log data through the system. Notably, the observed latency is predominantly 0, indicating minimal time to retrieve data from the Kafka server (value 0 could be attributed to the small size of our log data). In some instances, latency registers as 1 though,

5 Conclusion

Our project demonstrates the successful integration of Apache Kafka into a log data processing pipeline, offering a resilient, scalable, and real-time solution. Results we got highlights the efficiency in retrieving data from the Kafka server.

5.1 Future Works

Our project extends its capabilities by leveraging Spring Boot and Thymeleaf

- Explore and implement more sophisticated log data analysis modules, incorporating machine learning algorithms for anomaly detection or pattern recognition. This could provide deeper insights into system behavior and potential issues.
- Conduct extensive scalability testing by deploying the solution in larger environments or cloud platforms. Assess the system's performance and resource utilization under increased workloads, ensuring scalability for enterprise-level applications.
- Explore integrations with complementary technologies, such as Elasticsearch and Kibana, to enhance log data storage, search, and visualization capabilities. This could lead to a more comprehensive log management solution.

REFERENCES

[1] https://github.com/contactsunny/SimpleKafkaExampleSpringBoot/tree/master
[2] https://spring.io/projects/spring-boot
[3] https://kafka.apache.org/
[4] https://romanglushach.medium.com/the-evolution-of-kafka-architecture-from-zookeeper-to-kraft-f42d511ba242#:~:text=KRaft%20mode%20eliminates%20the%20need,lower%20latency%20and%20higher%20throughput.
[5] https://www.confluent.io/resources/online-talk/total-economic-impact-webinar/?utm_medium=sem&utm_source=google&utm_campaign=ch.sem_br.nonbrand_tp.prs_tgt.dsa_mt.dsa_rgn.namer_lng.eng_dv.all_con.online-talks&utm_term=&creative=&device=c&placement=&gad_source=1&gclid=CjwKCAiAmsurBhBvEiwA6e-WPFLXKEF3sv7AQtr_-ah9BRx8VHSHzYcXDGqmzhbrvz5dohi1C2AN7xoCwQkQAvD_BwE