Kafka and its Usages for Data Warehouse Projects

Introduction

The business world has been transformed by cloud-based technology, allowing businesses to quickly retrieve and store valuable data about their customers, products, and employees. This information is used to make critical business decisions. To coordinate data streams from corporate branches and operations centers all over the world, many global firms have turned to data warehousing (Figure 1). Some of these use-cases require real-time data processing. Apache Kafka is a distributed streaming platform, which is tailor-made for processing streaming data from real-time applications.

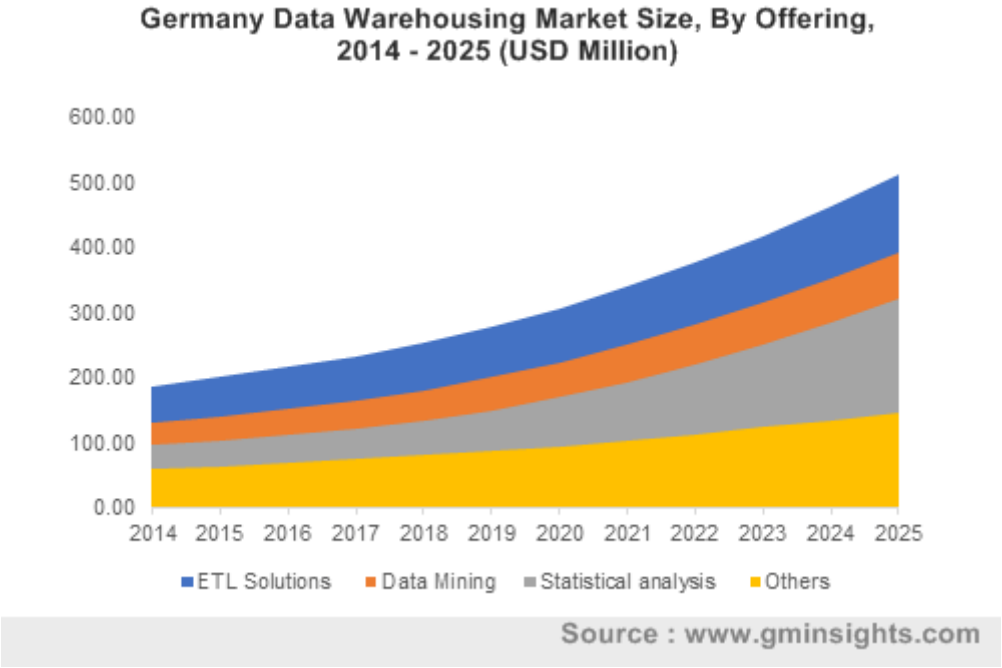

Figure 1 Kafka Log [1]

This short paper explains what Apache Kafka is and why and how it can be helpful in Data Warehouse projects. Furthermore, it is shown how to stream to and read data from Kafka in a small demonstration.

Kafka

Apache Kafka is an open-source streaming platform. It is written in Scala and Java and an open-source software platform developed by the Apache Software Foundation. It was initially created at LinkedIn as a messaging queue [2], but now it is a powerful tool for working with data streams, and Kafka can be used in many use cases. Apache Kafka comes from the German author Franz Kafka because it is optimized for writing, and the developer liked Franz Kafka's work.

Janik Bücher (1718249)

Apache Kafka's architecture is easier to understand than many of the options for application messaging. Kafka is merely a log of commits with a fundamental data structure. It is particularly fault-tolerant and horizontally scalable. The Kafka commit log maintains an ordered data structure over time. Records cannot be deleted or updated directly; they can only be appended to the log. In Kafka logs, the order of the items is secured. For each subject that exists, the Kafka cluster builds and updates a partitioned commit log. Messages sent to the same partition are saved in the order in which they are received. As a result, the records in this commit log structure have an ordered and immutable sequence. Each record is also given a distinct sequential ID, known as an offset, which is used to retrieve data in Kafka (Figure 2).

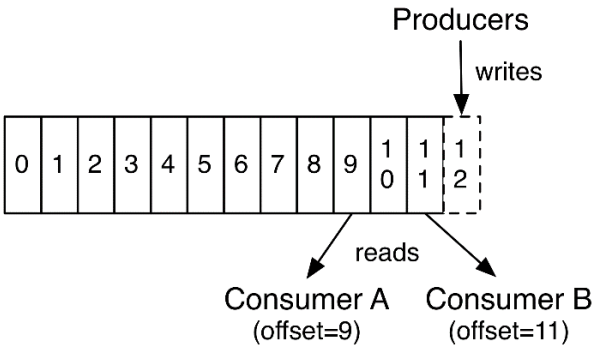

Figure 2 Kafka Log [3]

By providing set ordering and deterministic processing, Kafka addresses common problems with distributed systems. Kafka benefits from sequential disk reads because it stores message data on disk and in an orderly fashion. Given the high resource cost of disk seeks, Kafka processes read and writes at a consistent rate, and reads and writes occur concurrently without interfering with one another, resulting in significant performance gains.

Horizontal scaling is easy with Kafka. This means that Kafka would deliver the same high level of achievement regardless of the task at hand, from small to large.

Kafka's Architecture

Kafka architecture consists of topics, producers, consumers, consumer groups, clusters, brokers, partitions, replicas, leaders, and followers. The following diagram offers an overview of its architecture (Figure 3):

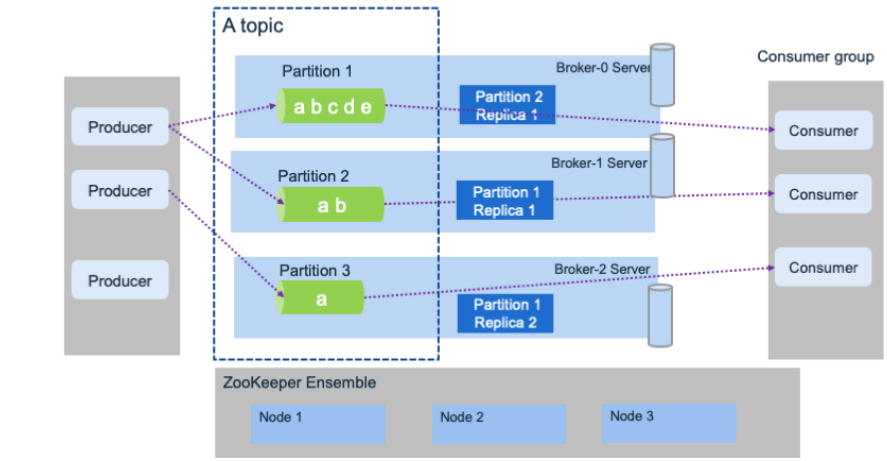

Figure 3 Kafka Architecture [4]

A Kafka cluster contains several brokers. Multiple brokers often form a Kafka cluster, which provides load balancing, reliable redundancy, and failover. The cluster is managed and coordinated by brokers using Apache ZooKeeper. Without affecting performance, each broker instance can handle read and write volumes of hundreds of thousands per second (and terabytes of messages). Each broker has its ID and can be in charge of one or more subject log partitions. ZooKeeper is also used by Kafka brokers for leader elections, in which a broker is chosen to lead the handling of client requests for a specific partition of a subject. Any broker connection bootstraps a customer to the whole Kafka cluster. A minimum of three brokers should be used to achieve reliable failover; the higher the number of brokers, the more accurate the failover.

ZooKeeper is used by Kafka brokers to maintain and organize the Kafka cluster. When changes occur in the Kafka Cluster, such as when brokers and themes are added or deleted, ZooKeeper notifies all nodes. ZooKeeper also facilitates broker and subject partition pair leadership elections, allowing users to choose which broker will be the leader for a given partition and which brokers will hold replicas of the same data. To be protected against a broker crash event, ZooKeeper notifies the cluster of broker changes and a new partition leader can be assigned.

A Kafka topic is a data-streaming channel defined by Kafka. Consumers read messages from the themes they subscribe to, and producers write messages to topics. Topics are used to organize and structure messages, with different messages being issued to different topics. Within a Kafka cluster, topics are named by specific names, and there is no limit to the number of topics that can be created. Partitions serve to replicate data across brokers. Each Kafka topic is divided into partitions, and each partition can be placed on a separate node.

Kappa Architecture

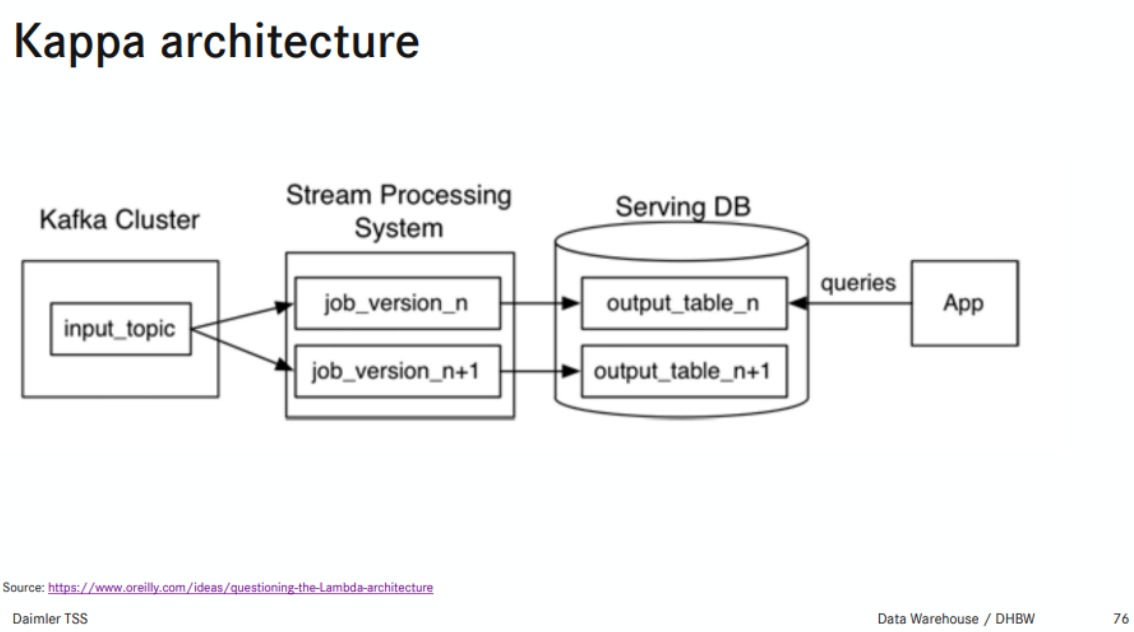

Figure 4 Kappa Architecture (Slide from Lecture) [5]

Since it uses the same technology stack to manage both real-time stream processing and historical batch processing, the Kappa Architecture (Figure 4) is considered a simplified alternative to the Lambda Architecture. Usually, the Kappa Architecture is based on Apache Kafka and a high-speed stream processing engine.

One of the Kappa Architecture's most significant advantages over the Lambda Architecture is combining streaming and batch processing into a single device. This means it is possible to create a real-time stream processing program. The Lambda Architecture suggests that batch processing should be handled by a separate technology.

Data Warehouse Capabilities

Apache Kafka is part of a streaming architecture. A streaming architecture is a group of technologies that work together to manage stream processing, which acts on a collection of data while it is being generated. In several current deployments, Kafka serves as the store for the streaming data, and then various stream processors can operate on the data stored in Kafka to produce varied outputs.[6]

In another demo application [7], a consumer was used for the evaluation, which queries data from the topics in intervals, approximately every minute, and stores them temporarily in a local database. This database contains only the data of the last x time intervals to remain small and quickly evaluable. A simple reporting was integrated to have an almost real-time evaluation. Each time the report is refreshed, the latest data is integrated from Kafka.

The demo

This short demo shows how to stream data to Kafka and how to read this data as well. For this, a producer and consumer were implemented in the python programming language (see below). Both producer and consumer utilize the "kafka-python" package to connect to the Kafka broker.

The Kafka broker and ZooKeeper client both run in docker containers, using public available docker images. To simplify the demo's installation and usage, the producer and the consumer run in separate docker containers. All containers run in a docker network to achieve easy port access and communication. This is all automatically set up by running a "docker-compose" file. A topic called 'Test' is created automatically, with one partition and one replica to keep the demo simple.

```
7  def produce():
8      # write to the topic
9      print('\n<Producing>')
10     producer = KafkaProducer(bootstrap_servers=[bootstrap_server])
11     for i in range(20):
12         producer.send('Test', ('Message: ' + str(i)).encode() )
13     producer.flush()
14     producer.close(timeout=2)
15
```

Figure 5 Producer code snippet

Figure 5 shows a code snippet from the producer. At first, the producer establishes a connection with the Kafka broker (line 10). Then it created data by sending 20 times a message to the 'Test' topic (line 11-12). At last, the connection is safely closed.

```
8  def consume():
9      print('\n<Consuming>')
10     consumer = KafkaConsumer(
11         'Test',
12         auto_offset_reset='earliest',
13         enable_auto_commit=True,
14         group_id=None,
15         #value_deserializer=lambda m: loads(m.decode('utf-8')),
16         bootstrap_servers=[bootstrap_server])
17
18
19     for m in consumer:
20         print(m.value)
```

Figure 6 Consumer code snippet

Figure 6 shows a code snippet from the consumer. The consumer establishes a connection with the broker and subscribes to the 'Test' topic (line 10-16). Have a look at the "auto_offset_reset='earliest'" parameter, which tells the consumer to read all entries within

the topic and not only the latest ones (line 12). Finally, all entries of the topic are printed to the console (line 19-20).

For further information have a look to the Github repository.

Conclusion and Outlook

Apache Kafka is a perfect tool to create real-time stream processing for large data sets. It is used to store a large amount of data quickly to be processed by other stream processors. Furthermore, it can be used in DWH projects to gain real-time information about data stored in the DWH.

Kafka uses a pull-based architecture, which means that applications can consume data whenever it is needed. It outperforms conventional messaging systems in terms of throughput.

Sources

#	Source
1	Ankita Bhutani, Preeti Wadhwani (09.2019). Data mining emerges as a valuable process for business growth, Retrieved 23.03.2020 from https://www.gminsights.com/industry-analysis/data-warehousing-market
2	Jun Rao (11.01.2011). Open-sourcing Kafka, LinkedIn's distributed message queue, Retrieved 23.03.2020 from https://blog.linkedin.com/2011/01/11/open-source-linkedin-kafka
3	Kafka 2.0 Documentation. Topics and Logs, Retrieved 23.03.2020 from https://kafka.apache.org/20/documentation.html
4	IBM Garage Event-Driven Reference Architecture. Kafka Overview, Retrieved 21.03.21 from https:/ibm-cloud-architecture.github.io/refarch-eda/technology/kafka-overview/
5	Buckenhofer-DWH03-Architecture.pdf
6	B. R. Hiraman, C. Viresh M. and K. Abhijeet C., "A Study of Apache Kafka in Big Data Stream Processing," 2018 International Conference on Information , Communication, Engineering and Technology (ICICET), Pune, India, 2018, pp. 1-3, doi: 10.1109/ICICET.2018.8533771.
7	Tobias Otte (14.2.2020). Wie Apache Kafka ein real-time-DWH unterstützt, Retrived 21.03.2020 from https://blog.viadee.de/wie-apache-kafka-ein-real-time-dwh-unterstuetzt