Comparison Between Flink and Kafka

CS651 Project 2023 Fall

Felix Jing
University of Waterloo
Student ID: 21012858

1.0 Abstract

This study undertakes an in-depth exploration of two prominent big data processing frameworks : Kafka and Flink. The primary focus is on comprehending the architecture of these systems and operation of Kafka and Flink. By doing do, I aim to gain a solid understanding in these two big data frames and how to evaluate any big data frameworks.

2.0 Learning Objectives

1. Acquire a comprehensive understanding of Flink as a novel framework.

2. Enhance knowledge of Kafka, diving into technical specifics not covered in lecture slides.

2.1 Methodology

The research methodology from the theoretical side involves a thorough review of peer-reviewed and highly cited academic papers pertinent to Kafka and Flink, sourced from Google Scholar. On the practical side, I utilized online resources such as YouTube, Medium, and various other websites for further information and understanding. In addition, I have installed the Kafka and Flink using brew on my mac and attached two simple java files that shows the difference of these two frameworks.

3.0 What is Apache Flink?

Apache Flink, or Flink for short, is an open-source system for processing real-time stream data [6]. It can also handle batch data but treat the batch data as a special case of stream data. It originates from high-performance cluster computing and data processing frameworks. The philosophy Flink build upon is that data processing applications can be expressed and executed as pipelined fault-tolerant dataflows [6]. This philosophy has great impact on many data engineering tools.

3.1 Flink Paradigm

Apache Flink operates on a unique paradigm that centralizes around data-stream processing. This paradigm embraces data-stream processing as the unifying model for real-time analysis, continuous streams, and batch processing both in the programming model and in the execution engine [6].

The beauty of this paradigm lies in its flexibility. Whether it's processing the latest real-time events, aggregating data in large windows, or analyzing terabytes of historical data, Flink makes no distinction in processing[6]. The only difference is where they start in the durable stream and the type of state they maintain during computation.

Flink's paradigm also includes a highly adaptable windowing mechanism. This allows for the computation of both early and approximate results, as well as delayed and accurate results, within the same operation. This means there's no need to juggle between different systems for different use cases.

Another key aspect of Flink's paradigm is its support for different notions of time, such as event-time, ingestion-time, and processing-time. This provides programmers with a high degree of flexibility when defining how events should be correlated.

The way Flink handles batch processing is also noteworthy. From its early days, Flink has adhered to the principle that "Batch is a Special Case of Streaming". This approach allows for a seamless transition between batch and stream processing, further enhancing its versatility and efficiency [10].

3.2 Flink's Runtime and APIs

Although Flink's of API continues to grow, in this report, I will be focusing on four main layers of Flink, and they are deployment, core, APIs, and libraries.

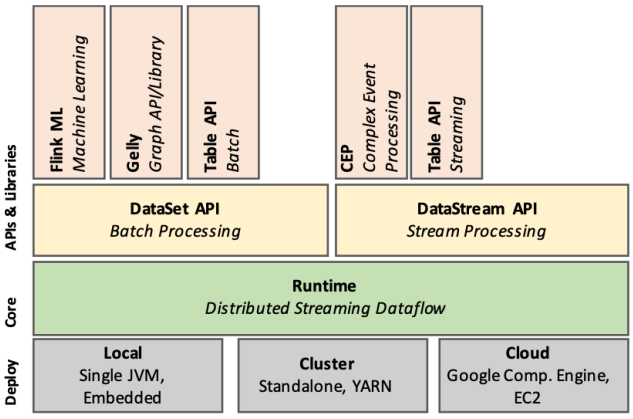

Figure 1: Flink Software stack from [6]

As show on the left-hand side, core is distributed streaming dataflow, which is also the philosophy of the Flink. This unit executes the dataflow programs. Then there are two essential APIs. The DataSet API, this also referred as batch processing and DataStream API, also known as stream processing. The engine serves as the common fabric to work with the batch and stream processing.

On the very top of the figure 1, it shows more domain-specific libraries and APIs that generate DataSet and DataStream API programs. FlinkML is for machine learning, Gelly for graph processing and Table API for SQL-like operations.

3.3 Flink cluster and processes

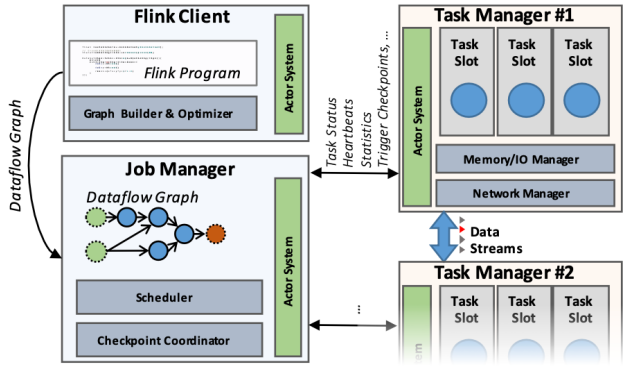

Figure 2: The Flink process model from [6]

Flink integrates with all common cluster resources managers, such as Hadoop, and Kubernetes. It can also be setup as. Standalone cluster or even as a library [8]. There are three types of processes: the client, the Job Manager, and the Task Manager. Task Manager needs to have at least 1. The client takes the program code and transform it to a dataflow graph, then submits to the JobManager. The JobManager manages the distributed execution of the dataflow. It keeps tracks the state and progress of each operator and stream, whenever needed it schedule new operator and coordinate the checkpoints and recovery. In a high-availability situation, the JobManager persists a minimal set of metadata at each checkpoint to fault-tolerant storage. So, the secondary or standby JobManager can reconstruct the checkpoint and recover the dataflow execution. In the attached SimpleFlinkJob.java file.

The **Client** is responsible for setting up the execution environment and executing the job. This is represented by the

StreamExecutionEnvironment.getExecutionEnvironment()

and

env.execute("Simple Flink Job")

The **JobManager** is responsible for managing the distributed execution of the dataflow. This is represented by the

DataStream<String> dataStream = env.fromElements("Hello", "World")

and

transformedStream.print()

The **TaskManager** is responsible for executing the tasks (i.e., transformations) that make up the job. This is represented by the

DataStream<String> transformedStream = dataStream.map(...)

The follow line enables checkpointing and sets the interval between checkpoint to 1000 milliseconds. The other lines configure various aspects of checkpointing, such as the checkpointing mode, he minimum pause between checkpoints, the checkpoint timeout, the maximum number of concurrent checkpoints, and the behavior when the job is cancelled.

env.enableCheckpointing(1000);

env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);

env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);

env.getCheckpointConfig().setCheckpointTimeout(10000);

env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);

I want to add a note that in real-world scenario, the Flink job would be much more complex. Also there will be additional consideration for fault tolerance and high availability, which I didn't implement in the java files.

3.4 What is the dataflow graph?

All Flink programs eventually compiles to a common representation: the dataflow graph [6]. It then gets executed by Flink's runtime engine or core. From the following figure, the dataflow graph can be depicted as a directed acyclic graph (DAG). We can see two important components, stateful operators (SRC1 and SRC2) and data streams.

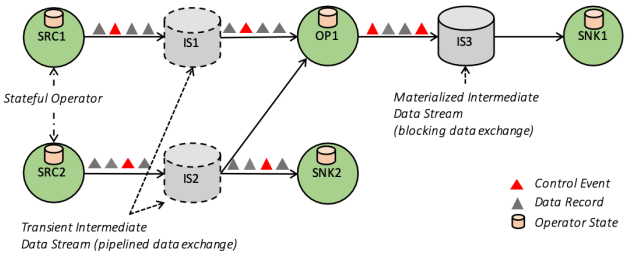

Figure 3: Example of data flow graph, from [6]

Data streams represents the data produced by an operator and are available for consumption by other operators. Streams distribute data between producing and consuming operators in various patterns, such as point-to-point, broadcast, re-partition, fan-out and merge.

4.0 What is Apache Kafka?

From the lecture 8 slides, we got a brief introduction Kafka as "A distributed pub-sub broker designed for resilience and availability", when it is introduced, Kafka is one of the popular data brokers and how it has been adopted by users. In short, it was open sourced in 2011 through the Apache Software Foundation and has since become one of the most popular event streaming planforms.

After more of the digging into the Kafka original introduction paper, I obtained a deeper understanding. Basically, Kafka is a versatile platform that excels at scalable, real-time data streaming for modern architectures. It was originally developed for collecting and delivering high volumes of log data with low latency [7]. It is a distributed streaming platform for building real-time data pipeline and streaming applications at massive scale.

Kafka Architecture

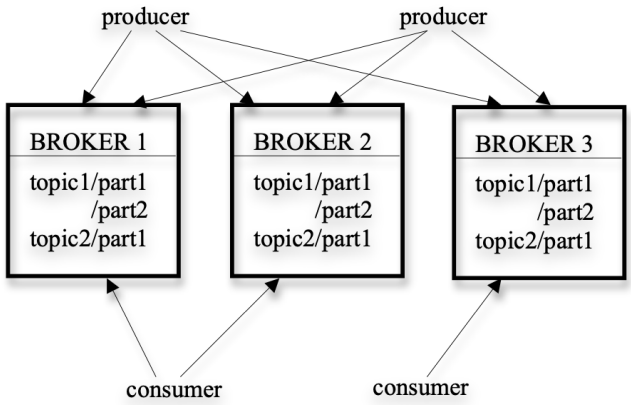

Figure 4. Kafka Architecture

Kafka organized the stream of messages of type as topic. A producer can publish message to a topic. The published message is then stored at a set of servers called brokers. The consumer can subscribe to one or

more topics from the broker and consume the subscribed messages by pulling data from the brokers. Here I present some sample code to show how the API is used.

Sample producer code:

```
producer = new Producer(…)

message = new Message(“Test Message str”.getBytes())

set = newMessageSet(message)

producer.send(“topic1”,set)
```

Sample consumer code:

```
streams[] = Consumer.createMessageStreams(“topic1”, 1) for (message : streams[0]) {

bytes = message.payload();
// process the message

}
```

To subscribe to a topic, a consumer first creates one or more message streams for the topic. The message published to that will be evenly distributed into these sub-streams. Each message stream provides an iterator interface over the continual stream of messages being produced. The consumer then iterates over every message in the stream and process the payload of the message. Unlike the traditional iterators, the message stream iterator never terminates [7]. If there are no current message to consume, the iterator blocks until new messages are published to the topic. Kafka supports both point-to-point delivery model in which multiple consumers jointly consume a single copy of all messages in a topic and publish/subscribe model which multiple consumers retrieve their own copy of a topic.

4.1 How Kafka distributed the messages?

Now I will describe how the producers and consumers behave in a distributed setting.

Its core queuing and messaging features power an array of critical applications and workloads. It can be considered as the answer to problems faced by distribution and scaling of messaging systems.

Event stream are organized into topics that are distributed across multiple servers called brokers. This ensures data is easily accessible and resilient to system crashes. Applications that feed data into Kafka are called producers, while those application that consume data are consumers. Kafka’s strength lies in its ability to handle massive amounts of data, flexibility to work with diverse applications, and its fault tolerance. This sets apart from simpler messaging systems. Kafka has become a critical component of modern system architectures due to its ability to enable real-time, scalable data streaming.

4.2 What is Kafka Streams

Kafka Streams is a data processing and transformation library within Kafka. It is capable of perform complex processing but doesn’t support batch processing. It is a General-purpose Data Stream Processing Engines (GDSPE) [10]. A stream which represents an unbounded and immutable data record, this is an important abstraction in Kafka Streams [10]. Kafka Streams offers two ways to define processing topology. The first one being Kafka Streams Domain Specific Language (DSL). It provides the common data transformation operations, such as, mapping, filtering, join and aggregation out of the box. The second way is Process API. This enables the definition and creation of the any customized the processors to interact with state stores and windowing operations. In many aspects, this can be compared with Flink, and I will list the comparison in section 5.

4.2 Kafka Use cases?

Here are a few Kafka’s most common and impactful use cases.

First, Kafka serves as a highly reliable, scalable message queue. It decouples data producers from data consumers, which allows them to operate

independently and efficiently at scale. A major use case is activity tracking. Kafka is ideal for ingesting and storing real-time events like clicks, views and purchases from high traffic website and applications. Uber and Netflix use Kafka for real-time analytics of user activity. For gathering data from many sources, Kafka can consolidate disparate streams into unified real-time pipelines for analytics and storage. This is extremely useful for aggregating internet of things and sensor data. In microservices architecture, Kafka serves as the real-time data bus that allows different services to talk to each other.

Kafka is also great for monitoring and observability when integrated with the ELK stack. It collects metrics, application logs and network data in real-time, which can then be aggregated and analyzed to monitor overall system health and performance. Lastly, Kafka enables scalable stream processing of big data through its distributed architecture. It can handle massive volume of real-time data streams. For example, processing user click streams for product recommendations, detecting anomalies in IOT sensor data, or analyzing financial market data. Kafka has some limitations too. It is quite complicated. It has a steep learning curve. It requires some expertise for setup, scaling, and maintenance. It can be quite resource-intensive, requiring substantial hardware and operational investment. This might not be ideal for smaller start-ups. It is also not ultra-low-latency applications, like high frequency trading, where microseconds matter.

4.3 Why is Kafka fast?

Firstly, we need to clarify what does it even mean that Kafka is fast. Are we talking latency? Are talking throughput? It is fast compared to what tools?

Kafka is optimized for high throughput. It is designed to move many records in a short amount of time. Think of it as a very large pipe moving liquid. The bigger the cross-section area of the pipe, the larger volume of the liquid can be moved through it. So, when people generally saying Kafka is fast, which really saying is that Kafka allow data more efficiently.

4.4 Design choice made Kafka more efficient.

There are many design choices that made Kafka fast. I think the following contributing the most weight: Kafka's reliance on sequential input output, and it focus on efficiency [9].

There is common misconception that disk access is slower than memory data access. But this is largely depending on the data access patterns. There are two major data access patterns, random and sequential.

For hard drives, it takes time to physically move the arm to different locations on the magnetic disks. This is what makes random access slow. For sequential access, since the arm doesn't need to jump around, it is much faster to read and write blocks of data one after the other.

Kafka takes advantages of this by using an append-only log as its primary data structure. An append-only log adds new data to the end of the file. This access pattern is sequential. Now let's bring this idea home with numbers. On modern hardware with an array of these hard disks, sequential writes reach hundreds of MB/secs, while random write are measured in hundreds of kB/sec. Sequential access is several order magnitudes faster. Using hard disks has its cost advantages too. Compared to SSD, hard disks come at 1/3 of the price but with about 3 times the capacity[9]. Giving Kafka a large pool of cheap disk space without any performance penalty, means that Kafka can cost effectively retain messages for a long period of time, a feature that was uncommon to messaging systems before Kafka.

The second design choice that gives Kafka its performance advantage is its focus on efficiency. Kafka moves a lot of data from network to disk, and from disk to network. It is critically important to eliminate excess copy when moving pages and pages if data between the disk and the network. This is where zero copy principle comes into the picture, Modern UNIX operating systems are highly optimized to transfer data from disk to network without copying data excessively. Let's dive deeper to see how this is done. Firstly, we look at how Kafka sends a page of data on disk to the consumer when zero copy is not used at all.

Firstly, the data loaded from disk to the OS cache. Secondly, the data is copied from the OS cache into the Kafka application. Third the data is copied from Kafka to the socket buffer. And forth the data is copied from the socket buffer to the network interface card buffer. And finally, the data is sent over the network to the consumer.

Now this is clearly inefficient. There are four copies and two system calls. Now let's compare this to zero copy. The first step is the same. The data page is loaded from the disk to the OS cache. With zero copy, the Kafka application uses a system called sendfile() to tell the operation system to directly copy the data from the OS cache to the network interface card buffer. In this optimized path, the only copy is from the OS cache into the network card buffer. With a modern network card, this copying process is done with DMA, which stands from direct memory access. When the DMA is used, the CPU is not involved, making it even more efficient.

4.5 Kafka Usage at LinkedIn

In the lecture 8 slide page 42, we know that LinkedIn is one of the users. Here I will describe more how LinkedIn use Kafka. LinkedIn has been using Kafka for both online and offline applications [7]. LinkedIn has on Kafka cluster collocated with each datacenter where their user facing services run [7]. The frontend services generate different kinds of log data and publish it to the local Kafka brokers in batches. Then it relies on a load-balancer to distribute the requests to the set of Kafka brokers evenly. The online consumers of Kafka run in services within the same datacenter. Below figure shows a simplified version of the Kafka deployment in LinkedIn.

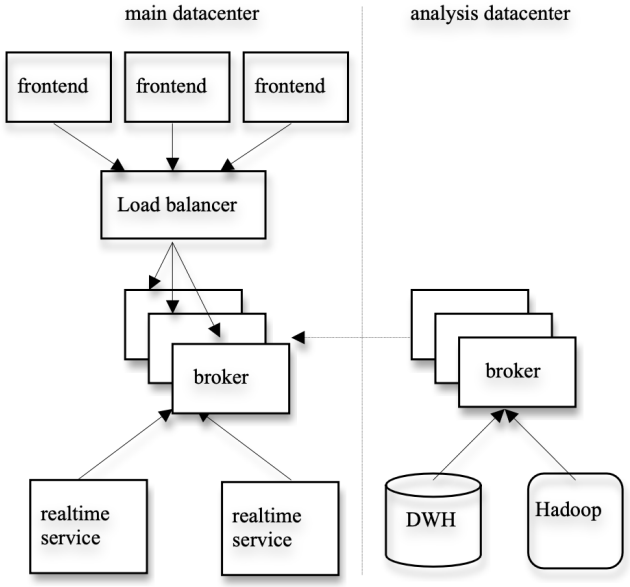

Figure 5. Simplified Kafka Deployment at LinkedIn, from [7]

They also deployed a cluster of Kafka in a separate datacenter for offline analysis. This analysis cluster shows on the right-hand side of the figure, is located closer to their Hadoop cluster and other data warehouse infrastructure [7]. This analysis cluster pull the data from the main datacenter. Then the data is loaded to Hadoop and data warehouse (DWH in the figure). They also use this Kafka cluster for prototyping and run scripts against the raw event streams for any ad hoc querying [7]. The end-to-end latency for the complete pipeline is about 10 seconds, which is amazing. With the introduce of the rebalance process, they can automatically redirect the consumption when brokers started or stopped for software or hardware maintenance.

LinkedIn also introduced an auditing system to verify no data is lost along the pipeline [7]. Each message carries the timestamp and the server's name when they are generated. The producer has been instructed to periodically generates a monitoring event, which record the number of the messages published by that producer for each topic within a fixed time window. The producer publishes the monitoring event to Kafka in a separated topic. Then the consumers can count the number of messages they received from a specific

topic and validate with the number from the monitoring events.

5.0 Comparison Between Flink and Kafka Stream

Having explored the fundamental system architectures of both Flink and Kafka, it's clear that these two frameworks exhibit significant differences. In this section, I will highlight the key distinctions in deployment and management aspects between Kafka Streams and Flink [1][3].

	Flink	Kafka Streams
Deployment	It is a cluster framework, and can take care of deploying, either in standalone or with docker, Kubernetes, etc.	Library within any Java application, hence, don't have a deployment method
Where the code run	As a job in the Flink cluster	Inside their own application
Ownership	Data Infra team	Whoever manage the application
Coordination	JobMangaer	Kafka cluster performs the coordination, load balancing, etc.
Fault-tolerance	By the dedicated master node, which implements its own high availability mechanisms based on ZooKeeper	Using Kafka core primitives. Each instance of the application acts independently

Figure 6: Summary of difference between Flink and Kafka Streams

5.1 Comparison Between Flink and Kafka

In the preceding sections, I have conducted a comparative analysis of Apache Flink and Kafka Streams, a library within Apache Kafka. This comparison should have shed light on their differences. Now, let's encapsulate the key distinctions between Flink and Kafka as the conclusion.

Apache Kafka is known for its ability to consume and produce data into streams, databases, or the stream processor itself. It's commonly used with Kafka as the underlying storage layer but operates independently of it. Kafka's strengths lie in its scalability, reliability, durability, flexibility, and large ecosystem of tools and libraries. However, it can be complex to set up and manage, may exhibit high latency for some use cases, can be expensive to deploy and operate, and if not properly configured, can pose a security risk.

On the other hand, Apache Flink is a fully-stateful framework known for its low-latency processing capabilities, making it ideal for use cases that require real-time analysis and processing of data. It can store the state of the data during processing, making it ideal for applications that require complex calculations or data consistency. Flink scales well both horizontally and vertically, allowing it to handle larger data volumes and complex processing tasks. It's also more user-friendly, especially for developers familiar with Java or Scala, and offers a wide range of APIs for seamless integration with other systems. However, Flink can be complex to set up and manage for large-scale deployments. It has a steeper learning curve than other streaming frameworks due to its complexity and wide range of features. Compared to other streaming frameworks, Flink has a smaller ecosystem of tools and libraries, which can make finding the right tools for a specific use case more challenging.

6.0 Conclusion

In wrapping up, my exploration of both Flink and Kafka has led me to appreciate their unique strengths and challenges. There are many existing and emerging applications that require processing of high volume of data. The choice between these two would largely depends on the specific context and situation and related functional and non-function requirements. This project has been a valuable journey in enhancing my understanding of big data frameworks. I've come to realize that many other frameworks may share

similarities with Flink and Kafka in certain aspects. This knowledge will undoubtedly be beneficial in navigating the landscape of big data.

6.1 Open Challenges

The increasing volume and highly irregular nature of the data rates pose new challenges to big data professional, but also offered job opportunities. It would be great if there is better benchmarking for big data tool, attributed includes but not limited to, performance, security, cost and integration with other tools and libraries. The ranking and recommendations for big data tools remain an active research area [2]. Kafka and Flink are among the cutting-edge data streaming processing engines, but many other application or function can be and needs to be extended [2]. In addition, educating the users and decision makers would also contribute no less than the technology improvement itself.

Acknowledgements

I would like to express my deepest gratitude to Daniel Holtby for his exceptional teaching in CS451/651. His dedication to imparting knowledge and his energetic teaching style have greatly enriched my learning experience. Thank you, Professor Daniel Holtby!

References

[1] Flink vs Kafka Streams - Comparing Features - Confluent. https://www.confluent.io/blog/apache-flink-apache-kafka-streams-comparison-guideline-users/.

[2] Vyas, S., Tyagi, R. K., Jain, C., & Sahu, S. (2021, July). Literature review: A comparative study of real time streaming technologies and apache kafka. In *2021 Fourth International Conference on Computational Intelligence and Communication Technologies (CCICT)* (pp. 146-153). IEEE.

[3) Apache Flink and Kafka Stream: A Comparative Analysis. https://medium.com/@BitrockIT/apache-flink-and-kafka-stream-a-comparative-analysis-f8cb5b946ec3.

[4] Apache Flink vs Kafka: Unraveling the Differences | DoubleCloud. https://double.cloud/blog/posts/2023/06/kafka-vs-flink/.

[5) Apache Kafka vs. Apache NiFi vs. Apache Flink Comparison - SourceForge. https://sourceforge.net/software/compare/Apache-Kafka-vs-Apache-NiFi-vs-Flink/.

[6] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, *38*(4).

[7] Kreps, J., Narkhede, N., & Rao, J. (2011, June). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (Vol. 11, No. 2011, pp. 1-7).

[8] *Flink architecture.* Flink Architecture | Apache Flink. (n.d.). https://nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/

[9] YouTube. (2022, June 29). System design: Why is Kafka Fast?. YouTube. https://www.youtube.com/watch?v=UNUz1-msbOM&ab_channel=ByteByteGo

[10] Wysakowicz , D. (2021, March 11). A rundown of Batch Execution Mode in the datastream API. Apache Flink. https://flink.apache.org/2021/03/11/a-rundown-of-batch-execution-mode-in-the-datastream-api/

[11] Isah, H., Abughofa, T., Mahfuz, S., Ajerla, D., Zulkernine, F., & Khan, S. (2019). A survey of distributed data stream processing frameworks. *IEEE Access*, 7, 154300-154316.