Université Libre de Bruxelles

INFO-F-415

Project report: Advanced Databases

Stream Databases – Apache Kafka & Apache Flink

Authors:
Grégoire Jean-Nicolas : 446638 (M-INFO)
Installé Arthur : 495303 (M-INFO)
Vanderslagmolen Moïra : 547486 (M-INFO)
Ze-xuan Xu : 541818 (M-INFO)

Date: December 2024

Contents

1 Introduction

The project we chose is stream database with two different tools: Apache Flink and Apache Kafka. We chose Kafka first because it is a widely used stream database and it is open source. It is now used in thousands of companies around the world. Then we decided to use Apache Flink, which is known to have an integration with Apache Kafka. In this report, we present a short introduction to the stream database. Then, we present Apache Flink and Apache Kafka and how stream databases are implemented in these tools. Finally, we show how we implemented the same application with three different tools :

- Apache Kafka
- Apache Flink
- Apache Kafka and Apache Flink working together

and we will compare these three tools using the benchmarking tools of Kafka and Flink.

2 Batch Processing vs Stream Processing

To understand how stream databases work, it is important to know the difference between stream (used by stream database) and batch processing (used by traditional database management systems). These are two ways to handle data that work very differently.

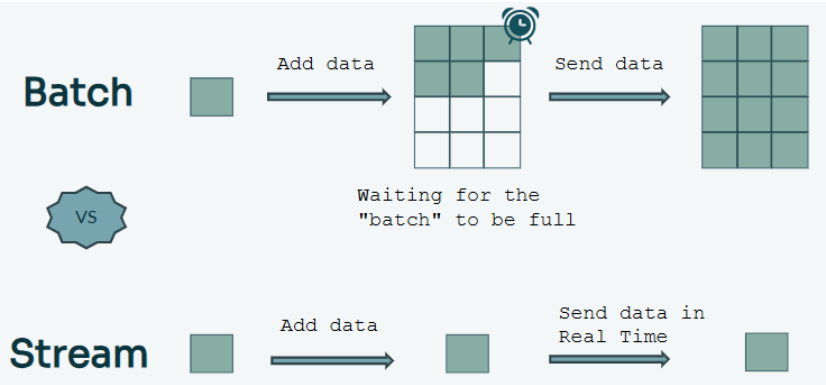

Figure 1: Comparison between stream database and batch processing

As we can see in Figure 1, batch processing is used by traditional database management systems. It works by collecting data until it reaches a certain number of data, after a certain time lapse or a size. It then processed it all at once as a batch. For example, analyzing sales data at the end of the day or running payroll calculations for an entire month are tasks typically handled with batch processing.

Stream database do the opposite of traditional database management systems (DBMS) and deal with data as it flows in real time. This means the data is processed continuously and almost instantly after it is generated. For example, we can imagine a live video feed or a real time financial transaction system.

3 Traditional database management system

3.1 Properties

- **Widely used:** Portable across different database systems and well documented.
- **Centralized storage:** Data are stored in tables with fixed schema. Optimized for structured data with strong consistency.
- **Batch processing:** Data are processed in bulk (collection of data), and traditional database management systems are not designed for real time or continuous data flow.
- **Query Optimization:** Most DBMS systems implement index research which optimizes queries.

- **Concurrency Control:** Implements concurrency control to prevent data corruption and data inconsistency.
- **BackUps:** DBMS facilitates utilization of replications of the data.

3.2 Strengths & Limitations

Since traditional database management systems have been used for storing and querying data for decades, they are reliable, well-tested, and mature. Systems such as PostgreSQL, MySQL, and Oracle Database provide strong guarantees of consistency, durability, and reliability. They are a natural choice for many applications because they established an ecosystem with rich tooling and community support.

One of the primary strengths of traditional database management systems is their suitability for predictable workloads, where the data and access patterns are fixed. These systems can handle complex queries and transactions easily, offering developers advanced capabilities such as indexing, joins, and stored procedures. These features make traditional database management systems ideal for e-commerce, inventory, and content management.

However, traditional database management systems have some limitations. For example, processing data in batches inherently introduces delays. High latency can be an issue when scaling systems working with larger or more distributed workloads. Additionally, traditional database management systems may struggle to scale horizontally (adding more servers), particularly when the volume or velocity of data exceeds their capacity. Finally, they can also be limited by their schema rigidity when adapting to unstructured or semi-structured data, such as JSON or XML formats.

Despite these limitations, traditional database management systems remain a solid choice for applications requiring strict consistency and complex querying capabilities.

4 Stream Databases

Stream databases are specialized to handle real time data as it flows in continuously, rather than working with fixed or pre-recorded data sets like traditional database management systems.

Unlike traditional database management systems that process data in batches after it has been collected, stream database work on data as it arrives. This allows them to process, store, and analyze real time information quickly and efficiently.

This capability makes stream database essential in many areas where immediate actions or decisions are required. For example, in chat applications, they help process and deliver messages instantly. In the "Internet of Things" (IoT), they enable smart devices to communicate and respond in real time. In financial systems, they are used for detecting fraudulent transactions as they happen, preventing potential losses. Similarly, they play a key role in transport and logistics by tracking shipments and optimizing routes in real time. They are also used in monitoring systems, such as detecting anomalies in industrial equipment or network traffic.

The key difference is timing: stream processing happens in real time, while batch processing happens after the data is collected. Stream database excel in situations where immediate responses are required, like detecting fraud as it happens or providing live updates in applications. Batch processing is more suited to scenarios where immediate results are not necessary, such as generating reports or conducting historical analysis.

4.1 Properties

- **Stateful Stream Processing**[1]: Can be useful for operations such as aggregations (sum, count, average, ...).

[1]Stateful processing means the system retains information about past data or computations to process incoming data more effectively.

- **Continuous Query Processing:** Results are updated based on changes (events) in data streams rather than recomputing everything from scratch. Stream database keep getting queries and processing them one by one.
- **Time-Ordered Processing:** Allows events to be processed in temporal order and handles late-arriving or out-of-order events.
- **Low Latency:** Designed to minimize the delay between data ingestion and query result generation.
- **Event-Driven architecture:** Each event triggers computations. It allows us enabling real time updates to alerts, updates, ...

4.2 Strengths & Limitations

Stream databases are designed for real time analytics and event-driven architectures. They excel in scenarios where data arrives continuously, and decisions need to be made almost instantaneously. Systems like Apache Kafka Streams, Apache Flink, and Amazon Kinesis let you process data quickly and support continuous querying. This is useful in industries like financial trading, IoT, and real time monitoring systems. In other words, they can process data as it comes in, offering low latency and real time analytics since they can react to changes or events as they occur. This grants them to receive updates without manual intervention.

However, stream database have some limitations. They often lack support for complex operations, such as multi-step joins or aggregations across large datasets. It can also be difficult to set up stream database to process data, given that you have to plan carefully for things like adding historical data or reapplying transformations.

Another aspect of stream database that can be limiting is that their monitoring and debugging can be complex. This is due to the fact that the continuous nature of data flow and real time processing introduces additional layers of complexity compared to batch-based systems. Stream database also consume a lot of CPU resources when it is running compared to DBMS.

5 Tools

5.1 Apache Kafka

Apache Kafka is a platform for streaming data in real time. It is widely used for building systems that process data streams efficiently and reliably. Kafka operates a cluster of servers, with key components:

1. *Brokers*: Servers that store event streams and manage data replication and coordinate the storage and retrieval of events. Brokers write the records they receive into topics, which are further divided into partitions for scalability. In a Kafka cluster, which is a collection of multiple brokers, each broker is uniquely identified by an ID. This allows for the distribution of data and ensures efficient management across the cluster.
2. *Producers*: Applications or systems that send events to Kafka topics. Producers take input from various sources, such as APIs or files (e.g., CSVs), and send data to brokers. The data can be sent in various formats, such as JSON or CSV, depending on the application's requirements. Producers determine the target topic and partition where the data will be written.
3. *Consumers*: Applications that read and process events from Kafka topics. After processing the information, consumers can send it to a sink, which could be another application, a database, or even a file storage system. Kafka ensures consumers can operate independently of producers, allowing seamless processing pipelines.
4. *Partitions*: Topics are split into smaller units called partitions, enabling parallel processing by allowing multiple producers and consumers to read and write simultaneously. This ensures scalability, and order for events with the same key are stored together in the same partition. It also allows to replicate the data if there is a crash of one partition.
5. *Replication*: Data in partitions is replicated across multiple brokers in the cluster to ensure reliability and availability even if servers fail.

We can see Apache Kafka as a processing engine that uses stream processing. It uses a TCP-Network protocol to communicate between the server (*Brokers*) and clients (*Producers*). Those clients publish (*write*) events, and consumers are those that subscribe to (*read & process*) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other. For example, producers never need to wait for consumers.

The main advantages of Kafka are first is: permanent storage as it can be configured for every record to stored and replicated in the topic. Also, As we've seen earlier, Kafka can handle a massive and infinite volume of records per second compared to traditional databases. With Traditional DB, the latency can grow to minutes and hours while with Kafka, the latency is usually around seconds or milli-seconds.

Kafka also ensure exactly-once semantics, which means that when the producer send one record, Kafka ensure that the consumer receive this record exactly once.

The other advantage of Kafka is that it can have hundreds of sources such as CSV, API, Socket and hundreds of sink such as Postgresql, etc.

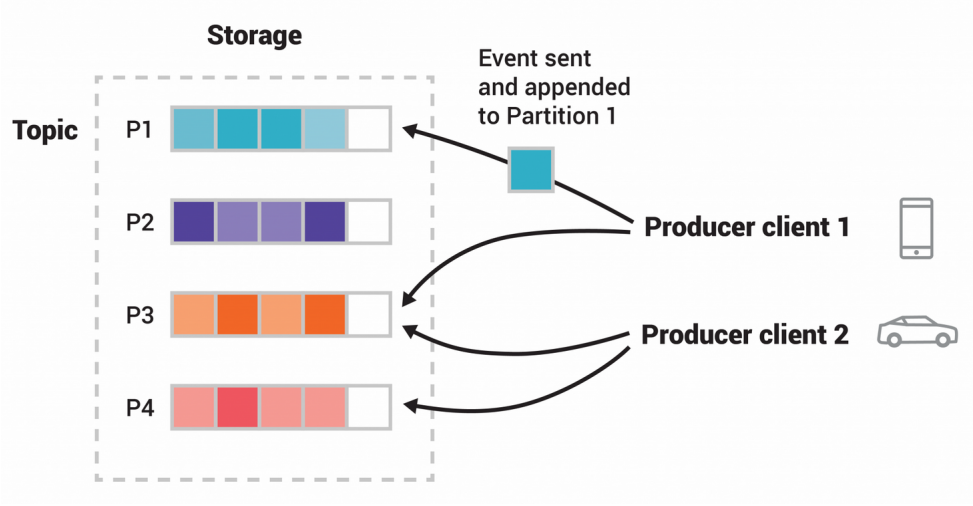

Figure 2: This example topic has four partitions $P_1 - P_4$. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate. [2]

It is also important to note the key roles of Apache Zookeeper in Apache Kafka. Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services [4]. In Apache Kafka, Zookeeper keeps track of the configurations of brokers and topics. It also elects the leader of the partitions, who is responsible for handling read and write operations on the partitions.

5.2 Apache Flink

Apache Flink is a framework and distributed processing engine. It uses stream processing but can also be used to do batch processing.

Those are the important concepts to understand in Flink:

1. *Source:* Connects external systems to ingest data into the application.
2. *Processing:* Refers to the core functionality where raw data is analyzed and transformed into meaningful outputs.
3. *Sink:* Endpoint where the processed data is written or sent. Sinks allows to store or forward results to external systems.

4. *Job:* Represent the complete data processing pipeline, from source to sink. It can transition through multiple states; it starts in the *created* state, it then moves to the *processing* state while processing tasks, and finally, once all tasks are completed, it transitions to the *finished* state. Some more complex states like *failing* or *canceling* also exist.

5. *Windowing:* Refers to chunking the data into windows. It can be very interesting in operations such as aggregations (sums, counts, ...). Windows can be data-driven (every 2 elements, count the number of visitors) or time-driven (every 10 seconds, calculate the sum of the last elements).

6. *Checkpoint:* Checkpointing periodically save the state of Flink and is crucial to ensure data consistency and exactly-once semantics[2].

7. *Stream Barriers:* Stream barriers are a fundamental part of checkpointing. They ensure that the state of the application is correctly checkpointed (consistent) by adding markers into the datastream. The markers indicate where the application needs to be checkpointed.

8. *Watermark:* In event-driven systems, a watermark indicates the last event that has arrived to be processed. A watermark with a timestamp t means that all events with timestamps less than or equal to t have been processed or are expected to have arrived. It helps Flink to manage out-of-order events or windows. For example, if a watermark with a timestamp of *20:00:05* is generated, it means that Flink assumes all events with timestamps less than or equal to *20:00:05* have been processed or are expected to have arrived. Thus, Flink can safely trigger computations for windows or other time-based operations that end at *20:00:05* or earlier.

In Flink, a data stream represents a collection of data. Data streams are created from various sources such as files, database, HTTP servers, socket streams, etc. The data imported from the source can be finite or unbounded. A Flink job is a job that will read from the data source and process it immediately and send it to the sink. A Flink cluster is a cluster of jobs.

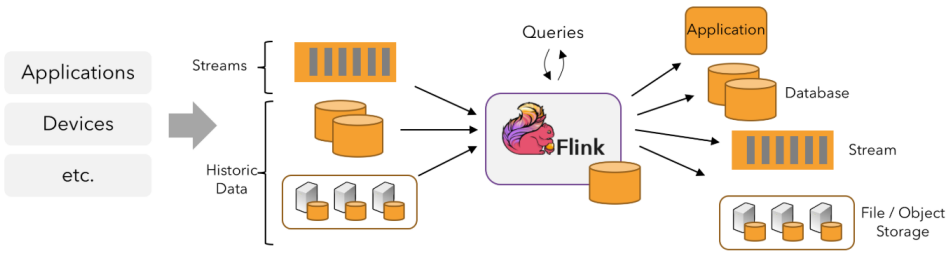

Figure 3: Data Flow in Apache Flink Stream Processing. [1]

5.3 Apache Kafka and Apache Flink

Apache Kafka and Flink are very often used together. The reason behind this cooperation is that Flink provides the computation, fault-tolerance (checkpointing) and the consistency while Kafka provides a large number of sources.

6 Implementation

The application we implemented is a live chat that moderates automatically. Each message is processed and analyzed when it passes through the stream database. If the message contains "illegal" words, we output that the message contains illegal words and is therefore not published. We chose a live chat because each message needs to be computed very fast, and there can be huge amounts of messages. We want to mimic the constant flux that is sent to other users in a server-based architecture such as Twitch.

The languages we chose for this application are Java and Python. We chose Java for our consumers because Flink is implemented in Java and Scala and has the best documentation for Java code. Each

[2]Exactly-once semantics means that the record sent to the Flink job is sent only once

consumer (Kafka, Flink and Kafka-Flink) is implemented in Java for consistency for the benchmark. Each producer is implemented in Python because Kafka has a great library for Python called kafka-python-ng and for simplicity.

6.1 Data

The data we used comes from this dataset. It contains 1.6 millions tweets extracted from the Twitter API with the following fields: the target (a scale from 0 to 4 representing sentiment, 0 being negative), the id, the date, the flag, the user, and the message content. The format of the data is a CSV file.

6.2 Apache Kafka

The first tool we implemented is Apache Kafka. For Apache Kafka, we need a topic, a producer (which will send the messages), a consumer (which will receive the messages), and a sink (the output of the messages).

6.2.1 Producer

We implemented the producer of Kafka in Python. We initialize a Kafka producer that connects to a port and serializes with JSON. Then, we iterate through a file, and for each line in the file we send with the producer to the topic chat the message in format JSON. The Kafka topic temporarily stores the message and waits until a consumer connects to the topic. Each consumer will always receive all the messages retained in the topic from the start.

6.2.2 Consumer

The consumer, implemented in Java, receives the messages sent by the producers. It connects to the Kafka broker with the function "subscribe" to a certain topic. While it is connected to the Kafka broker, it processes each record it gets from the broker with the class Utils. A record is in JSON format. This function, implemented in the class Kafka, will connect to a fixed broker with the topic "chat". It also set up the output function for the pipeline.

6.3 Flink

To implement Flink, we need a source, which will send messages, and a sink, which will receive the messages.

6.3.1 Source

For the source, we had a lot of trouble trying to implement it in Python. We tried to implement a socket and an HTTP server but it did not work well, so we decided to directly implement it in Java with the function *StreamExecutionEnvironnement.readTextFile* which creates a data stream from a file. Then we process messages by overriding the *processElement* function from the *Broadcast Process Function* class.

6.3.2 Sink

To create the sink, we need to create an instance of a SinkFunction where we override the processElement function. If we wanted to store the messages, we could have implemented a second sink that stores messages.

6.4 Kafka-Flink

To implement Kafka-Flink, we decided to use Kafka as the producer and Flink as the consumer and the messages will be processed using Utils. The advantage of this alliance is that Kafka already provides a broker. The class Kafka-Flink will create a Kafka consumer source for Flink to use to receive messages from producers.

6.5 Shared library

We created a class, Utils, that contains functions used to process each message received by the chat and output it.

First we create a Hashset that is going to store all the banned users.

Then, the function *processMessage*, implemented in the class Utils, processes each message received:

- Check if the user of the message is banned; if true, then it will tell that the message of the user is blocked with the reason: "User Banned".
- Check if the message contains banned words; if true, then it will block the message with the reason "Message contains banned word" and will ban the user of the message.
- If the message is not blocked, then we will show the message on the chat.

7 Benchmark

7.1 Setup

To benchmark our application, we used the Flink monitoring. We also used the Kafka Producer Benchmark and the Kafka Consumer Benchmark

The tests were run on this machine:

- Computer: Zenbook UM3402YAR_UM3402YA 1.0
- CPU: AMD Ryzen 7 7730U with Radeon G
- Memory: 15383MiB
- OS: EndeavourOS Linux x86_64
- Kernel: 6.12.1-zen1-1-zen

Each test was run 10 times and we took the average of it.

7.2 Results

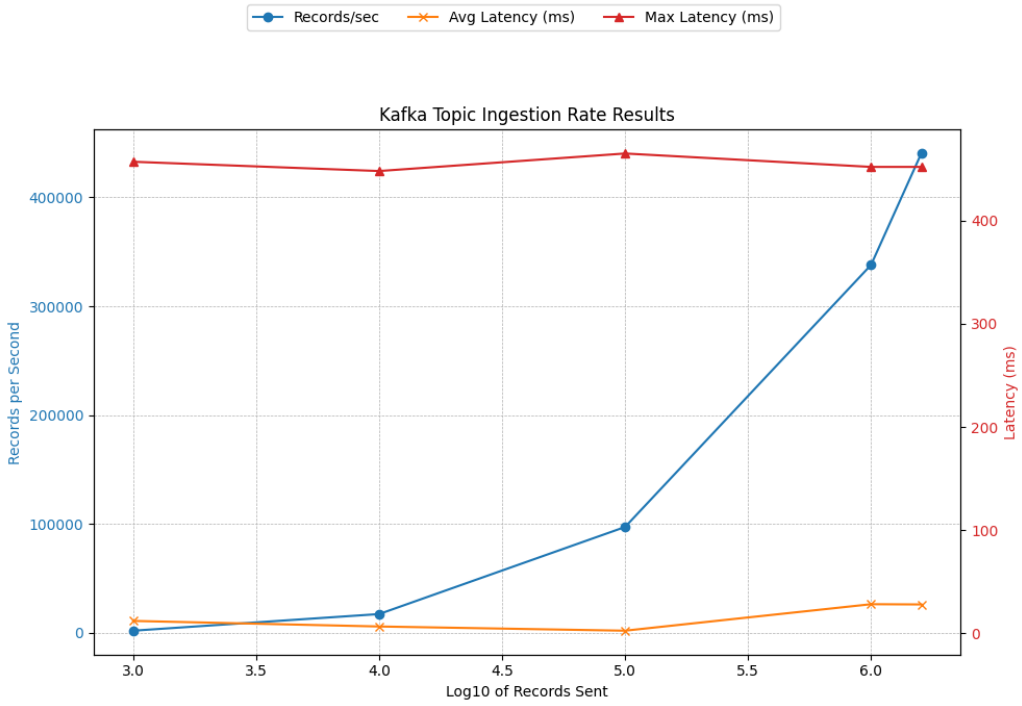

Figure 4: Benchmark of the ingestion rate of a topic

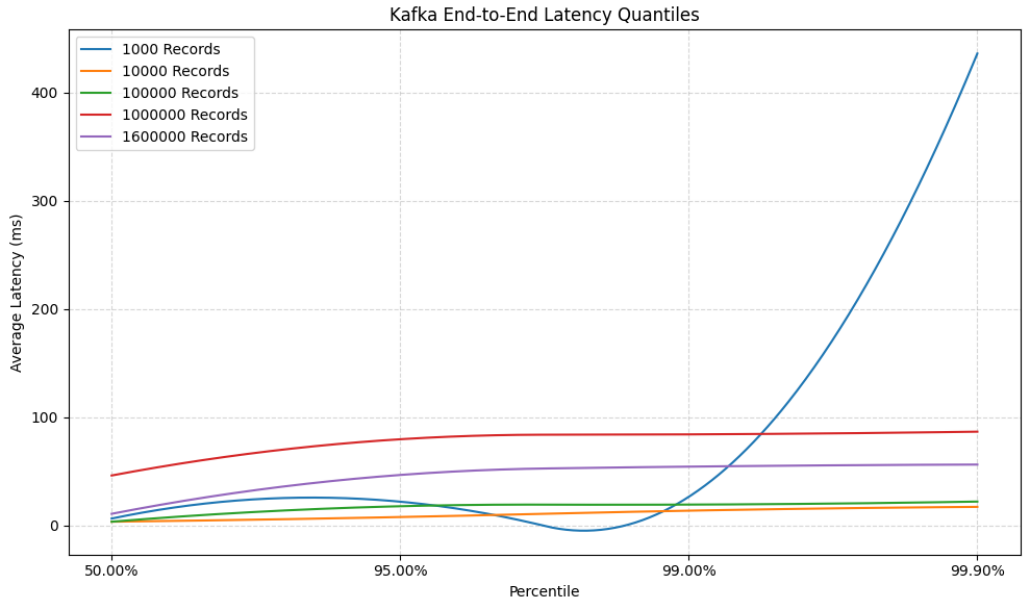

Figure 5: Kafka End-to-End Latency Quantiles

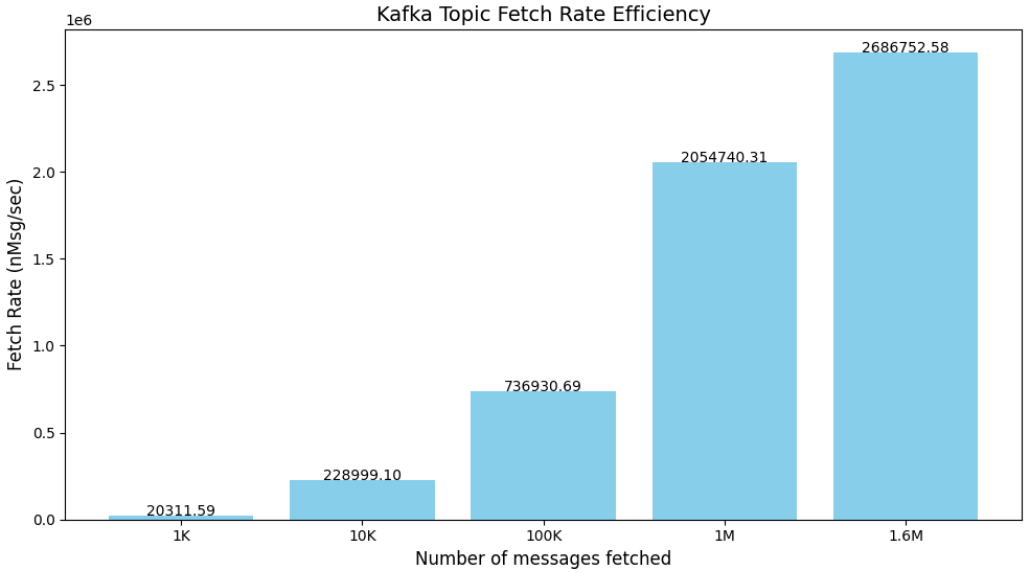

Figure 6: Benchmark from the topic to a consumer

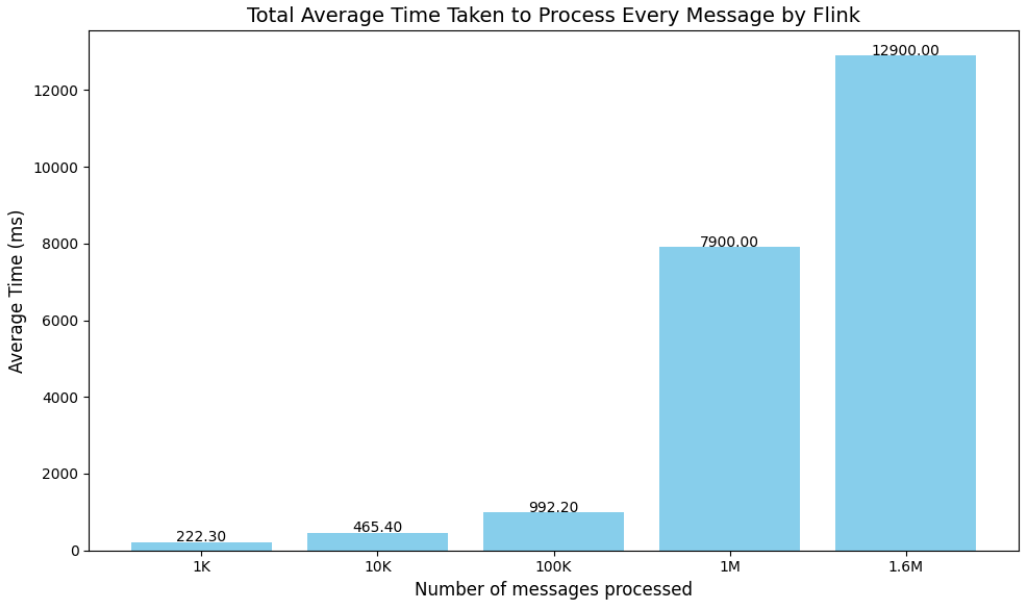

Figure 7: Flink Benchmark

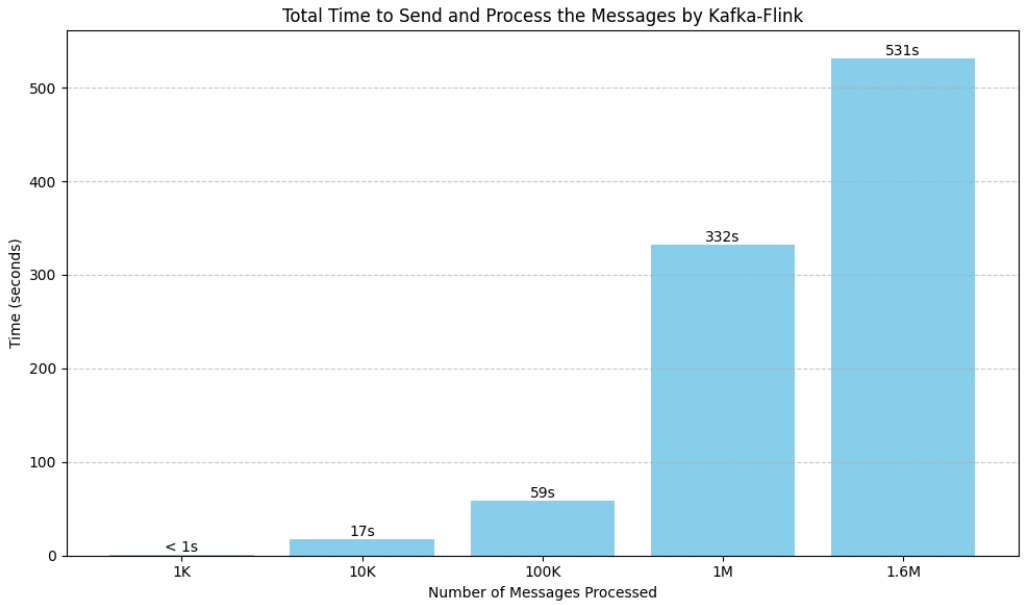

Figure 8: Kafka-Flink Consumer Benchmark

7.3 Analysis of the results

On the Figure 4, we notice that the topic ingestion rate (records/sec) increases significantly with the logarithm of records sent. The average latency remains low across all records count but starts increasing slightly as the number of records growths. Finally, the maximum latency is relatively consistent with minimal variation, indicating stability in maximum delays.

Moreover, we notice in Figure 6, Kafka's fetch rate improves as the number of messages requested increases. Obviously, we see that fetching 1M+ is not as quick as fetching 10K messages, but Kafka's efficiency scales as the number of messages growths.

The Figure 5 shows that Kafka has a consistent and stable latency for larger record sizes (1M+), with average latency remaining the same across all percentiles. However, it seems that smaller record sizes (e.g., 1K) show significant variability, especially at the 99th and 99.9th percentiles, where latency spikes due to overhead and inefficiencies in handling small batches. Optimal performance is achieved with moderate to large batch sizes, reducing network and processing overhead.

The Figure 7 shows that the total time taken to process messages by Flink increases non-linearly as the number of messages grows. Processing 1M messages takes approximately 8 seconds, while 1.6M messages require around 13 seconds. Although the processing time grows with workload size, Flink seems to efficiently scale for large message volumes

On the Figure 8, the time required to send and process messages by the Kafka-Flink consumer is linear as the volume of messages increases. Indeed, the growth between each result factor of 10 for the number of messages varies from a factor of less than 2 to 8. However, the time needed is much longer than Flink without a producer, we have already proved that the topic and Flink are not the issue here; hence the issue comes from our producer that sends messages to the topic too slowly.

Having a single producer to send messages to the topic is not enough. The topic can ingest messages very quickly, but a single producer does not send them quickly enough. Thus, to send a larger amount of messages, we need to partition the messages between multiple producers to finally send them to the topic.

8 Conclusion

As seen in the figures, the main advantage of stream database is the rapidity of execution. Kafka handles higher message throughput efficiently without significant deterioration in latency. Even with higher workloads, it takes only a couple of seconds to be ingested by a Kafka topic, and to be sent to a Kafka consumer. The main advantage of Flink is its exceptional performance to process the incoming data. Finally, the Kafka and Flink combination is in our opinion, the implementation that works the best. It is easy to implement and very well documented. The integration is commonly used and deeply developed.

References

[1] *Flink Documentation*. `https://nightlies.apache.org/flink/flink-docs-release-1.20/docs/dev/datastream/overview/`. [Online; accessed 10-December-2024].

[2] *Kafka Documentation*. `https://kafka.apache.org/documentation/`. [Online; accessed 10-December-2024].

[3] Robert Metzger. *Guide to Apache Kafka + Flink*. `https://www.ververica.com/blog/kafka-flink-a-practical-how-to`. [Online; accessed 10-December-2024]. 2015.

[4] *Apache Zookeeper*. `https://zookeeper.apache.org/`. [Online; accessed 15-December-2024].