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 databases with two different tools: Apache Flink and Apache Kafka. In this report, we present a short introduction to the stream databases. So 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 benchmarking.

2 Stream Databases

Stream databases are specialized to handle real-time data streams and huge amount of continuous query. It is designed to process, store and analyze real-time data streams unlike traditional databases which work on static data batches. We use stream databases in a large number of sectors, like chat, "Internet Of Things" (IoT), fraud detection in financial transactions, transport logistics, monitoring, etc.

2.1 How does stream databases work?

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

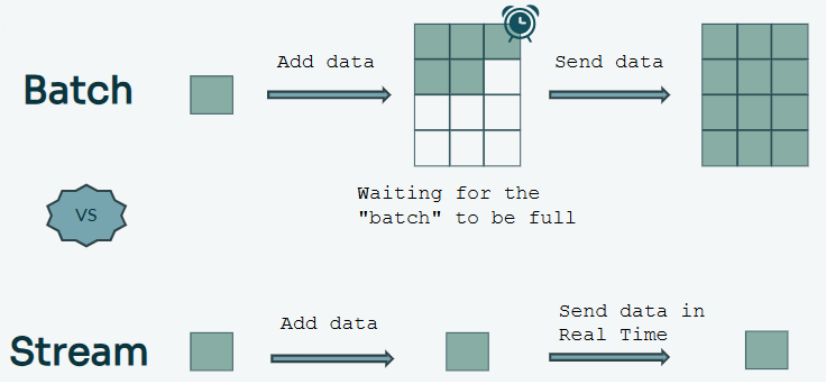

Figure 1: Comparison between stream databases and batch processing

As we can see in Figure 1, batch processing process the data by batches, which means that the processor waits until it has enough data to send it. Stream databases do the opposite and sends data directly.When data streams enter inside a stream database, it is immediately processed and try to get the next one.

3 Comparison

3.1 Traditional databases

3.1.1 Properties

- Centralized storage:
  - Data are stored in tables with fixed schema.
  - Optimized for structured data with strong consistency
- Batch processing:

  - Data are processed in bulk
  - Not designed for real-time or continuous data flow.

- Transaction Management:
  - Strong ACID (Atomicity, Consistency, Isolation, Durability) guarantees.
  - Suitable for applications like banking, e-commerce, and CRM.

- Query Optimization:
  - SQL is the dominant query language.
  - Indexing and normalization used for efficient query performance.

3.1.2 Strengths & Limitations

Since traditional databases have been used for storing and querying data for decades, they are reliable and mature. Systems such as PostgreSQL, MySQL, and Oracle Database ensure data's 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 databases 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 databases ideal for e-commerce, inventory and content management.

However, traditional databases have some limitation. For example, high latency can be an issue when scaling systems working with larger or more distributed workloads. Additionally, traditional databases may struggle to scale, particularly when the volume or velocity of data surpasses 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.

3.2 Stream database

3.2.1 Property

- Stateful Stream Processing:
  - Aggregations (sum, count, average, ...).
  - Statefull operation

- Continuous Query Processing:
  - Results are updated based on changes in data streams rather than recomputing everything from scratch
  - keep getting query and processing them one by one.

- Time-Ordered Processing:
  - Events are processed in temporal order
  - Support handling 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
  - Enabling real-time updates to alerts, updates, ...

3.2.2 Strengths & Limitations

Stream databases are designed for real-time analytics and event-driven architectures. They are suitable for 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 occurs. This allows them to receive updates without manual intervention.

Stream databases 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 databases to process data given that you have to plan carefully for things like adding historical data or reapplying transformations.

Another aspect of stream databases that can be limiting is that it's monitoring and debugging can be complex. This is due to the fact of the continuous nature of data flow and real-time processing introduces additional layers of complexity compared to batch-based systems.

4 Tools

4.1 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. These events are stored and distributed across partitions within topics for scalability.
2. *Producers*: Applications or systems that send events to Kafka topics.
3. *Consumers*: Applications that read and process events. Kafka lets consumers process data without affecting producers.
4. *Partitions*: Topics are split into parts, allowing multiple reads and writes. 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 brokers to ensure reliability and availability even if servers fail.

We can see Apache Kafka as a processing engine which 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.

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 keep tracks of the configurations of brokers and topics. It also elects the leader of the partitions, which is responsible for handling read and write operations on the partitions.

4.2 Apache Flink

Apache Flink is a framework and distributed processing engine. It uses stream processing but can also be used to do batch processing. In Flink, a data stream represent a collection of data. Data Streams are created from various sources such as files, databases, http server, 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.

4.3 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 and the consistency while Kafka provides the durability with the topics and the brokers.

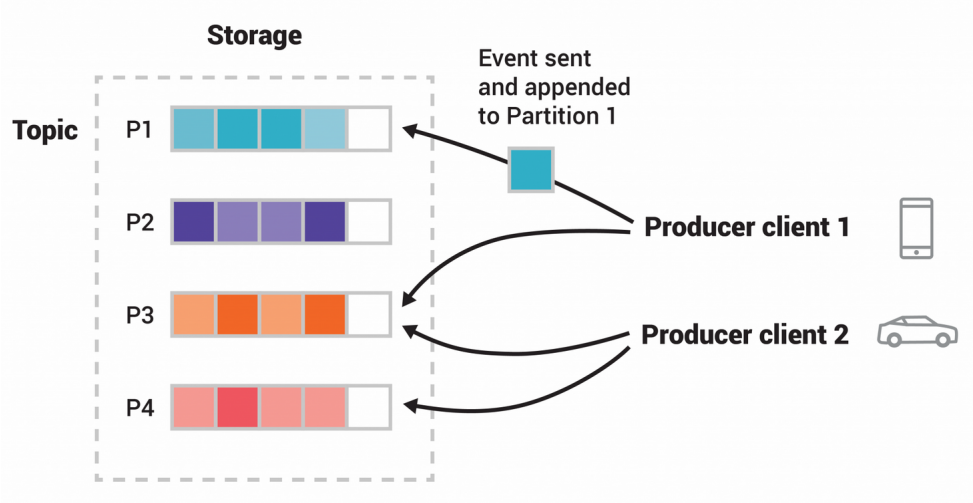

Figure 2: This example topic has four partitions $P_1 - P_4$. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written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

5 Implementation

The application we implemented is a live chat that moderates automatically. Each message is processed and analyzed when it pass 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 are send to other users in a server-based architecture such as Twitch.

The language we chose for this application is 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 consumers (Kafka, Flink and Kafka-Flink) is implemented in Java for consistency for the benchmark. Each producers is implemented in python because Kafka has a great library for Python called kafka-python-ng and for simplicity.

5.1 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.

5.2 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).

5.2.1 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.

5.2.2 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 setup output function for the pipeline.

5.3 Flink

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

5.3.1 Source

For the source, we had a lot of troubles trying to implementing 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.

5.3.2 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 implement a second sink that stores messages.

5.4 Kafka-Flink

To implement Kafka-Flink, we decided to use Kafka as producer and Flink as consumer and message 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 message from producers.

5.5 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 an Hashset that is going to store all the banned users.

Then, the function *processMessage*, implemented in the class Utils, process 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.

6 Benchmark

6.1 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.

6.2 Results

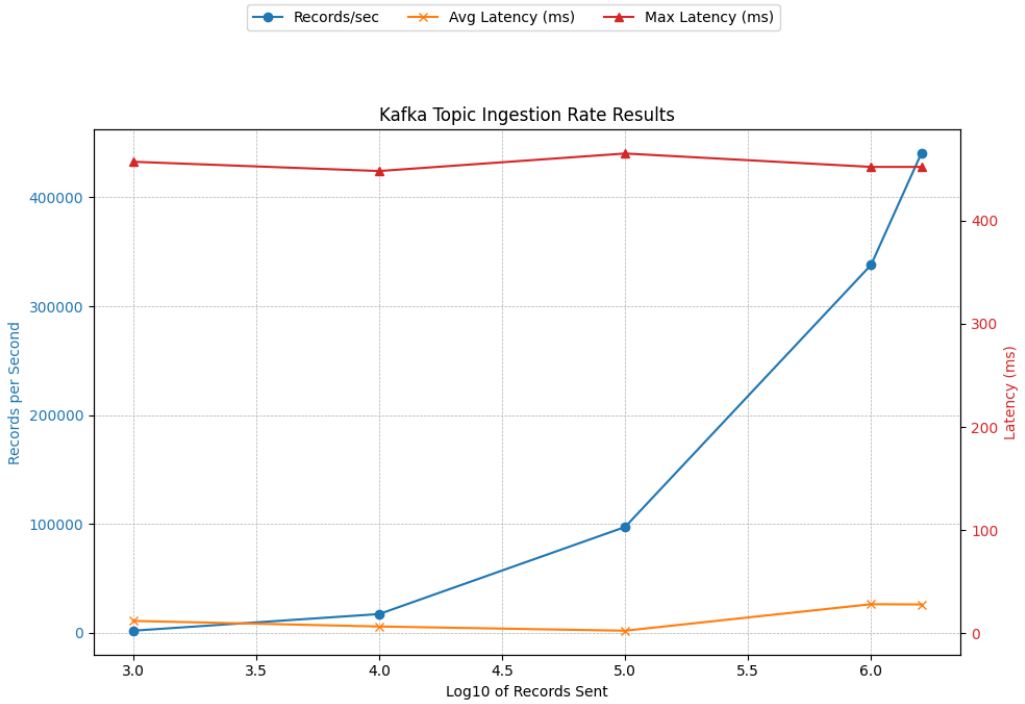

Figure 3: Benchmark of the ingestion rate of a topic

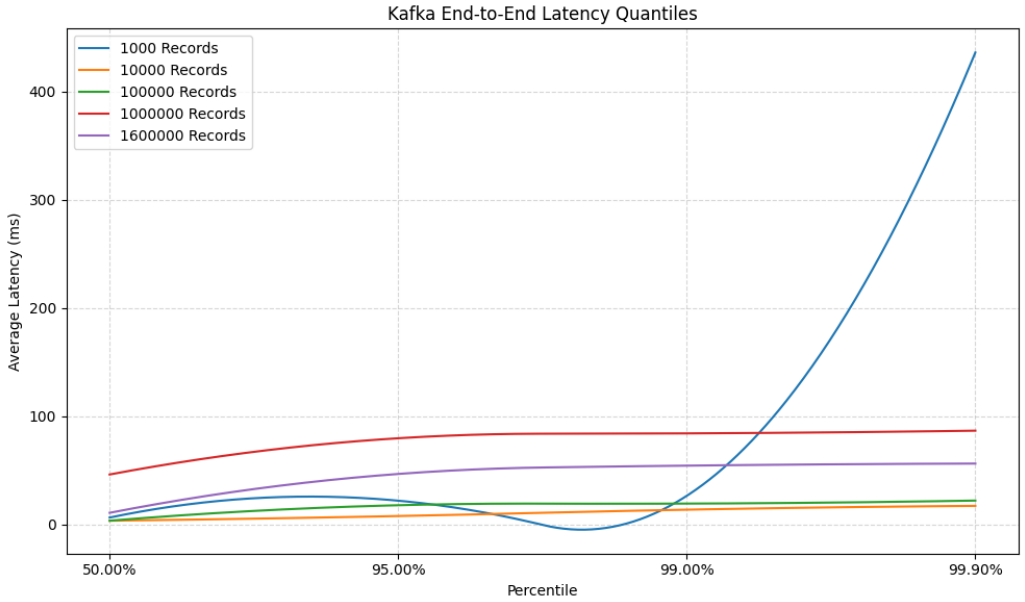

Figure 4: Kafka End-to-End Latency Quantiles

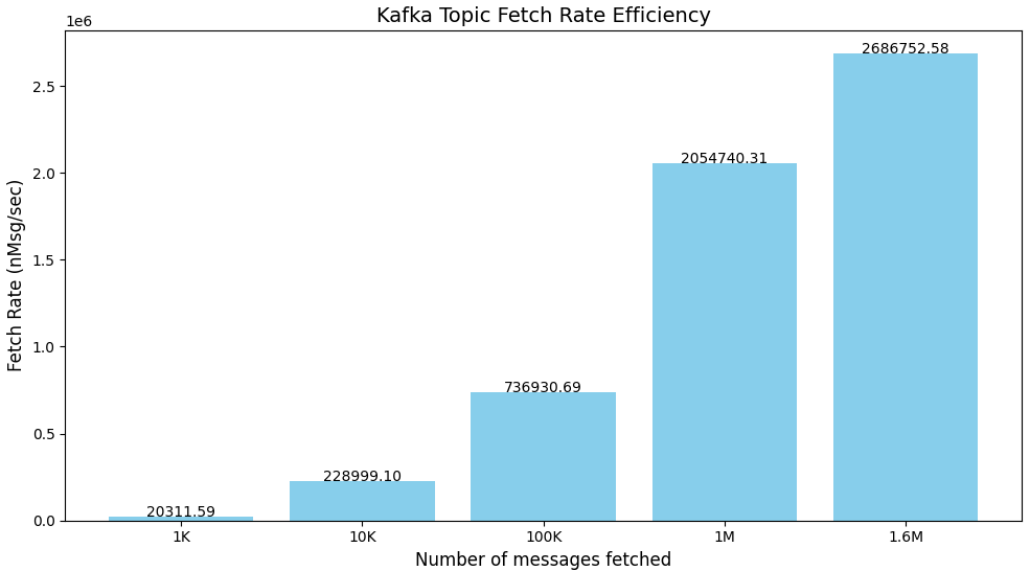

Figure 5: Benchmark from the topic to a consumer

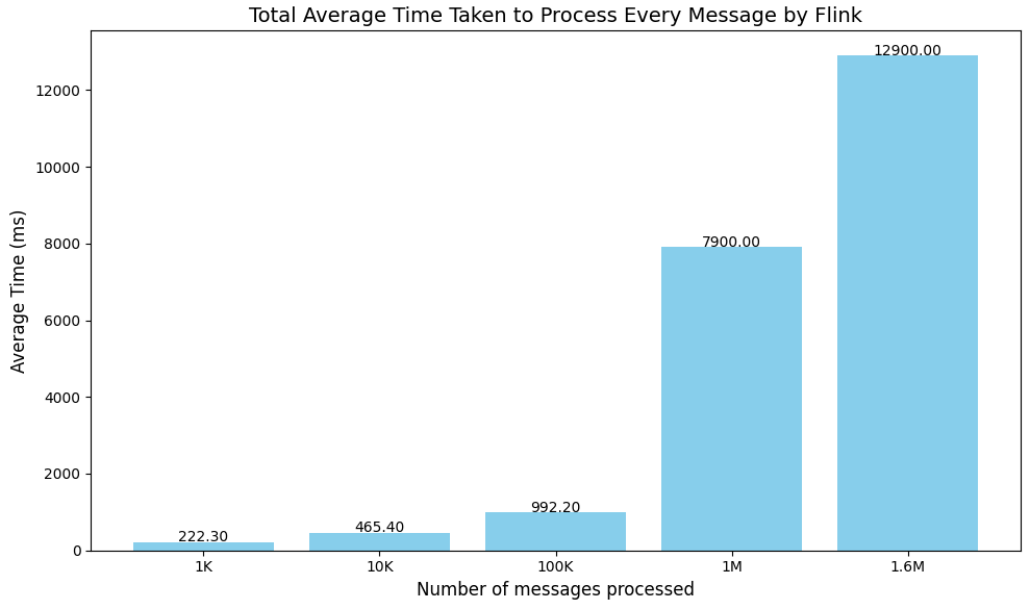

Figure 6: Flink Benchmark

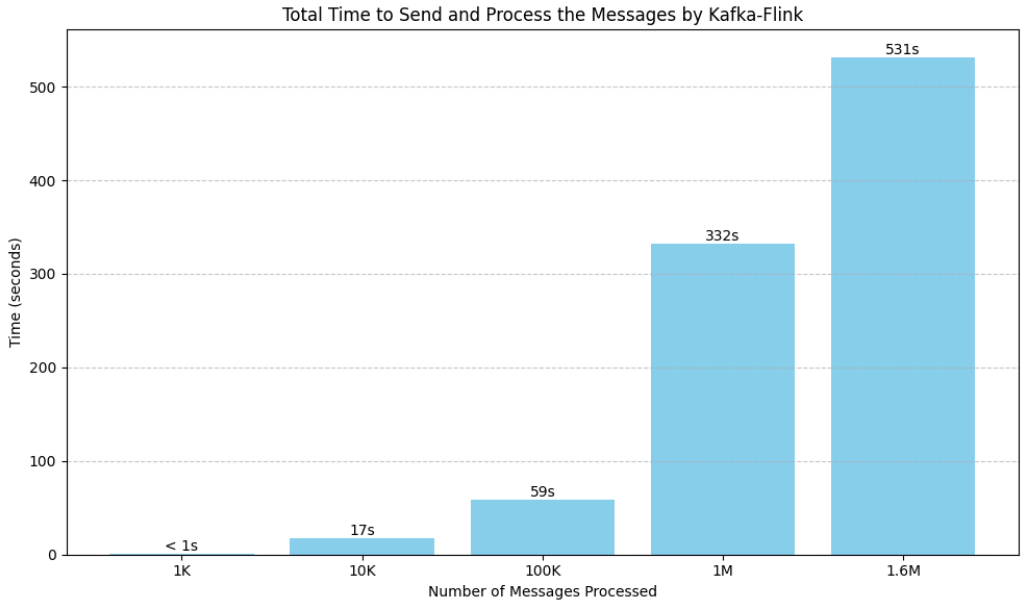

Figure 7: Kafka-Flink Consumer Benchmark

6.3 Analysis of the results

On the Figure 3, we notice that the topic increases his ingestion speed as the number of messages increases. So the latency between sending a message and its retrieval and storage in the topic is consistently small.

Moreover we also notice on the Figure 5, the topic increases his sending rate as the number of messages asked increase so sending 1 million of messages is as quick as sending only 1 thousand messages.

The Figure 4 shows that Kafka has a consistent and stable latency for larger record siez (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 6 shows that the time taken to process messages is linear. Directly getting messages from the source without passing by a producer, 1 million and 6 hundreds thousands messages only take few seconds to be processed by the Flink.

On the Figure 7, we notice that the growth for the time taken by the sending and processing of message by Kafka-Flink consumer is linear. Indeed, the growth between each 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 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 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 sends quick enough. Thus to send lots of messages, we need to partition the messages between multiple producers to send messages to the topic.

7 Conclusion

As we seen in the figures, the main advantage of stream databases is the rapidity of execution. Kafka works well on its own and is consistent and have stable latency. Even with data as big as millions of records, it takes only a couple of seconds to be ingested by a topic (for Kafka) and being send to a consumer (For Kafka). But we had a lot of issues implementing standalone sources with Flink. The main advantage of Flink is that it acts very well as processor, it can process a lot of messages very quickly. The combination of Kafka and Flink 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].