Real-time Stock Market Trend Analysis using Stream Processing Frameworks

Alekhya Pyla Rajasekhar Mekala Antriksh Ganjoo
{apyla, rmekala, ganjooa}@uci.edu

1. Introduction

In the dynamic and complex domain of the stock market, there are numerous factors that can influence its behavior, including news feeds, stock prices, social media sentiment, and economic indicators. Each of these factors generates its own data stream, with its own unique frequency of updates. Integrating and analyzing these disparate data streams in real time poses a significant challenge. However, by aggregating data from these diverse sources, we can uncover valuable insights into market trends, sentiment analysis, and potential investment opportunities.

Most organizations are facing an explosion of data coming from new applications, new business opportunities, and more. The ideal architecture envisioned is a clean, optimized system that allows businesses to capitalize on all that data. An event streaming platform can help solve these problems with a modern, distributed architecture rather than collecting information point-to-point. Instead, it offers a single platform to publish and subscribe, store, and process streams of data in real-time, at any scale, and in any environment. To address this challenge, we developed a stream processing framework that is designed to handle three key input streams: news feed, stock prices, and Tweets. These streams exhibit varying frequencies, with news feeds and stock prices being updated on an hourly or minute basis, while Tweets arrive sporadically and unpredictably. Our primary objective in developing this framework is to create a scalable system that is capable of processing large volumes of data in real-time while simulating real-world stream processing scenarios. This includes dealing with variable frequencies, diverse sources of data, variable input rates, and efficiently managing the system's load using techniques such as load-shedding. To achieve this, we have designed an architecture that can accommodate the continuous influx of data from the three input streams using Flink and Kafka. This architecture efficiently aggregates the data, performs computations and analysis, and generates meaningful visualizations.

2. Related Work

A. Introduction of Stream Processing

Stream analytics has become increasingly important in industries due to the explosion of data generated from various sources such as IoT devices, social media, sensors, and more. Real-time insights from these data sources help businesses make better decisions, improve customer experiences, and enhance operational efficiency. Recent demands for scalable and low latency have revealed various shortcomings of traditional batch processing systems such as MapReduce [1] and Hive [2]. One major disadvantage is the latency introduced by processing data in large batches at scheduled intervals. This delay is unsuitable for time-sensitive or real-time applications. Additionally, batch processing systems often struggle with limited freshness of data, as the data being processed may not be up-to-date. Scalability challenges arise as data volumes increase, resulting in longer processing times and resource inefficiencies.

Furthermore, batch processing systems lack built-in fault tolerance mechanisms, requiring entire batches to be rerun in case of failure.

In contrast, stream processing addresses these issues by enabling real-time or near-real-time data processing. With stream processing, data can be processed as it arrives, allowing for continuous analysis and processing of data streams, allowing organizations to make timely decisions and obtain up-to-date insights. Stream processing also offers low latency, enabling immediate actions and responses. Scalability is another advantage, as stream processing systems can efficiently handle high-volume data streams by distributing processing across multiple nodes. The continuous processing capability ensures a steady flow of insights and analysis, while built-in fault tolerance mechanisms enhance system reliability.

B. General Stream processing architecture

A general stream processing infrastructure refers to the underlying system or framework that enables the processing of continuous streams of data in real-time. It encompasses the necessary tools and components to ingest, process, analyze, and output data streams efficiently.

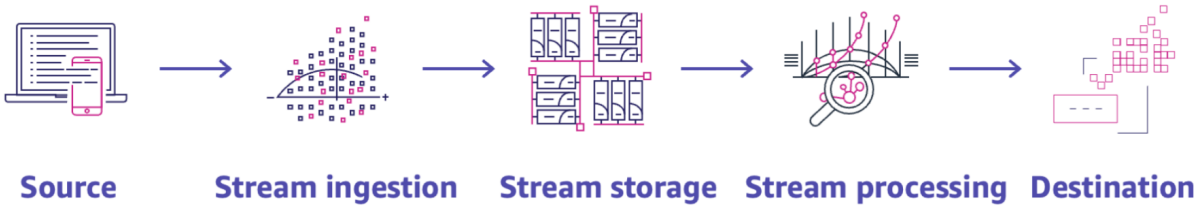

Figure 1: Stream Processing Framework

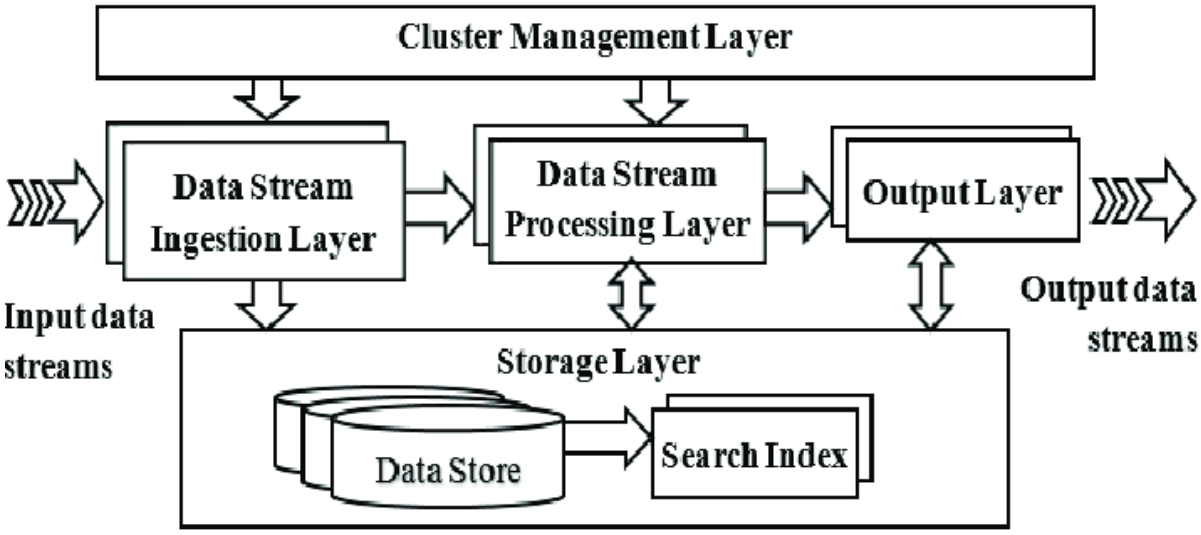

Figure 2: Stream Processing Infrastructure[3]

The several key components are as discussed below:

1. Data Source: The source of streaming data includes data sources like sensors, social media, IoT devices, and log files generated by using your web and mobile applications. The data generated can be structured, semi-structured and unstructured data and can be continuous streams at high velocity.
2. Ingestion Layer: This component handles the ingestion of data streams from various sources such as sensors, databases, or external APIs. It ensures the reliable and efficient collection of data into the stream processing system. The typical ingestion layer consists of message queuing systems such as Kafka or RabbitMQ.
3. Processing Layer: This is the core component that processes streaming data. It can perform various operations such as filtering, aggregation, transformation, enrichment, windowing, etc. A stream processing engine can also support Complex Event Processing (CEP) which is the ability to detect patterns or anomalies in streaming data and trigger actions accordingly. Some popular stream processing tools are Apache Spark Streaming, Apache Flink, Apache Kafka Streams, etc.
4. Storage Layer: A storage layer is responsible for storing, indexing, and managing both the raw data and the generated knowledge or insights. Data storage can be either persistent or ephemeral, relational or non-relational, structured or unstructured, etc. Some examples of data storage systems are Amazon S3, Hadoop Distributed File System (HDFS), Apache Cassandra, Elasticsearch, etc.
5. Cluster Management Layer: A cluster management layer coordinates the functions of distributed computing and storage resources, ensuring that the system can handle large volumes of data and scale as needed using actions such as auto-scaling.
6. Output Layer: An output layer directs the output data stream and generated knowledge to other downstream systems or visualization tools, where it can be used for further analysis or decision-making. It can consist of generated reports or dashboards based on predefined metrics or queries.

C. Existing Systems for Stream Processing

The adoption of stream analytics is expected to continue to grow as businesses increasingly rely on data to drive their operations and gain a competitive edge. However, finding the right technologies for stream analytics tasks can be challenging due to the complexity of processing which involves tools for ingesting, processing, storing, indexing, and managing streaming data. Based on the popularity and wide practical usage in the industry, stream processing technologies have arised such as Spark Streaming, Apache Flink, and Kafka Streams.

Apache Flink [4] is a powerful and popular open-source stream processing framework founded in 2014. Apache Flink is developed under the Apache License 2.0 by the Apache Flink Community within the Apache Software Foundation. Flink provides a unified programming model for both batch and stream processing, unlike previous systems. The previous batch processing approaches suffered from high latency and inaccuracy in results due to the splitting of the data basis into a time dimension. Flink aims to solve this problem such that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipelined fault-tolerant dataflows. Further, Flink has an advanced state management system that allows developers to maintain state across batch and stream processing. This was a significant improvement over previous systems, which required developers to maintain the state manually.

Flink sources generate data for a data stream while sinks consume data from a data stream such as Apache Kafka or local file systems. Flink can build end-to-end streaming applications where sources read data, apply processing logic, and output the results to sinks for storage or further analysis. The centerpiece of Flink is its distributed dataflow engine, designed to execute dataflow programs. Flink runtime programs consist of a directed acyclic graph (DAG) of stateful operators connected with data streams. Flink provides two key APIs: the DataSet API designed for batch processing and the DataStream API for stream processing.

Kafka Streams[5] is another important and popular stream processing library introduced in May 2016 (0.10 release) to build real-time streaming applications that can process and analyze data in real-time using Kafka as its source of data. Kafka Streams provides a simple and lightweight API for building streaming applications that can perform various operations such as filtering, aggregation, joining, and transforming data. Kafka Streams is tightly integrated with Apache Kafka, and it leverages its distributed architecture, fault-tolerance, and scalability features along with high-throughput and low-latency messaging for client applications.

D. Techniques for Improving Stream Processing

The concept of the 3Vs in big data, namely Volume, Velocity, and Variety, can also be applied to data stream processing. The 3Vs highlight the key characteristics that pose challenges in handling and analyzing continuous data streams. Volume refers to the large amount of data generated within data streams. Velocity represents the high speed at which data flows within streams. Variety relates to the diverse types and sources of data within streams. Handling the 3Vs in data stream processing requires specialized techniques to efficiently process and analyze the continuous data streams.

a. Load Shedding

In streaming data processing, stream processors face challenges when they have no control over the rate of incoming events, which can result in performance degradation when the input rates exceed the system's capacity. To mitigate this issue, load shedding is employed as a strategy to sustain the load by temporarily discarding excess tuples from inputs or intermediate operators in the streaming execution graph. Load shedding[6] involves a trade-off between result accuracy and sustainable performance, making it suitable for applications with strict latency constraints that can tolerate approximate results. In many stream processors, load shedding is limited to a predefined set of operators that do not modify tuples, such as filter, union, and join.

There are two main approaches: input-based shedding and state-based shedding. Input-based shedding strategies, such as Random input shedding (RI) and Selectivity-based input shedding (SI), discard events randomly or based on the query selectivity per event type. For example, Kafka implements Random input shedding. However, input-based shedding may not be suitable for complex event processing (CEP) queries, as the utility of a stream element heavily depends on the current state of query evaluation. To address this, state-based load shedding is introduced, which discards partial matches to maximize the recall of query processing during overload situations. Random state shedding (RS) and Selectivity-based state shedding (SS) are two strategies inspired by techniques used in approximate CEP.

b. Scalability in Stream Processing

Stream processing infrastructure should be scalable to handle high-volume and high-velocity data streams. It should be able to dynamically allocate resources and scale horizontally to accommodate increasing data processing demands. An ideal system should have elasticity, meaning it can dynamically acquire or release hosts to handle workload changes, including unexpected peaks and improve overall system utilization. Auto-scaling [7] technique is the key decision-making component in such an elastic scaling system. It determines when the system needs to scale in or out by comparing the current system utilization with predefined upper and lower thresholds. There are different methods for auto-scaling, including local thresholds, global thresholds, and reinforcement learning. Furthermore, stream processing systems have to be designed to be resilient to failures and ensure data integrity using mechanisms such as data replication, and checkpointing.

3. System Design and Architecture

Financial markets generate vast amounts of data from varied sources including real-time price feeds, news updates, and social media sentiment and with varied velocities respectively. Thus, stream processing in finance data analysis empowers financial institutions to make informed decisions and respond rapidly to market dynamics. It excels in applications such as fraud detection, algorithmic trading, real-time data monitoring, personalized recommendations and analytics.

Kafka and Flink are two powerful technologies for stream processing, each offering unique capabilities that complement the other in the data processing pipeline. Apache Kafka serves as a highly scalable and fault-tolerant distributed messaging system[5]. It acts as a reliable message queue, handling real-time data streams with low latency. Kafka's publish-subscribe model and persistent storage which ensure data availability and its ability to scale horizontally often make it a good choice as an ingestion tool. It also allows for data replayability, enabling the reprocessing of data streams.

Apache Flink, on the other hand, is a robust stream processing framework designed for complex event processing and analytics. It excels at high-throughput and low-latency stream transformations and stateful computations making it a good choice for a processing engine in stream processing architecture. Flink supports event time processing, handling out-of-order events and accurate time-based aggregations. Its fault tolerance mechanisms guarantee exact-once processing and state consistency.

Kafka and Flink together form a powerful combination for end-to-end stream processing. Kafka acts as a reliable and scalable message broker, handling the ingestion and distribution of data streams, while Flink performs the processing and analysis of these streams. Flink can consume data directly from Kafka topics, leveraging Kafka's parallelism and fault tolerance features. The integration between Kafka and Flink allows for seamless data flow and fault tolerance across the entire stream processing pipeline. When picking the data ingestion tool of Kafka, we had to make sure it can easily link up with data sources and destinations. Also, Flink provides the common transformations of data that are required for the analysis. Therefore, the scalability and reliability of Kafka and Flink make them a good choice for the project.

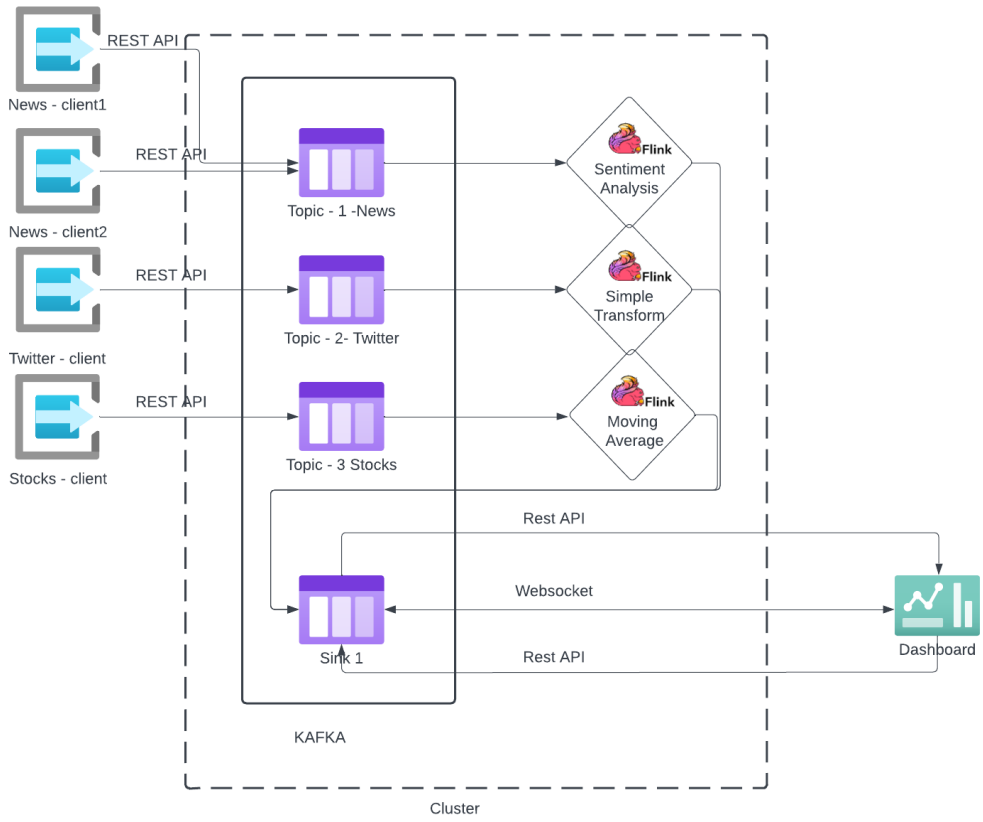

Figure 3: Architecture

The design of the system consists of multiple blocks or components. First is the sources. There are three different data sources, news , twitter, and stock data respectively and simulated different velocity scenarios for each with varied frequencies of daily, sporadic, and minute. Further, we have considered scenarios that involved significant data volumes, such as considering sporadic data bursts from news. Also, we incorporated varied sources of input to simulate scenarios involving diverse data types and sources. The sources for extracting the data from these APIs and sending into Kafka topics via REST API are written in python and NodeJs. For every type of data source, we have created one topic to ingest that particular information, since topics are a way of categorizing data. The news data is generated and is flowing into the system on an hourly basis. There are two sources of data for news, googleNewsAPI and YahooFinance APIs respectively. Similarly for Twitter, we have used TwitterAPI for fetching the tweets related to the company. Also, for stocks, we have used publicly available datasets for fetching the stock information. We have simulated generating the data with variable frequency and have added mechanisms to tackle this like load shedding.

Zookeeper and Kafka have been set up which receives the ingested data from above sources. The traditional configurations of both systems have been kept the same in our case. Each Kafka topic receives data from one source. In our application, Flink has the above-mentioned kafka topics as the input. These functionalities are implemented in Java. Flink performs the following operations on the data using DataStreams. From the news articles, it takes the headlines and

article information and performs sentiment analysis, which is represented as a highly positive to highly negative bar. For the stock data, it calculates the moving average of the stock.

After the resulting transformations are done, the output is saved in a different Kafka topic created for just persisting the results. The dashboard extracts the information from the Kafka and visualizations are shown. The frontend of the dashboard is built using ReactJs. The dashboard reads and interacts from Kafka sink using Rest APIs and Websocket.

To simulate the cases where the input rates can be too high than the processing capacities of Flink, we have implemented input-based shedding given that it is simpler and is generally more efficient. We performed experiments by randomly dropping 30 percent of samples when the input is high using filtering in Flink. To ensure the application is scalable, we have tried setting it up on AWS cluster, but given that it requires more processing power and the free tier micro of AWS wasn't sufficient in our case. Thus, we had to set up an ICS Cluster. Future work can be done to enable auto-scaling in this case.

4. Visualization Results

The following are the data visualizations through our dashboard which shows the real time stock market trend analysis using 3 different data sources Stock, Tweets and News.
For the stock UI we have shown the 30-day moving average using a bar plot on the top of the page as shown in Figure 4. Using the scrollbar, we can visualize the independent data by selecting one of the tabs. For stocks, the existing companies can be seen or a new one can be selected using Add New. Stock relevant data such as the company-wise stock percentage change, market capitalization and volume of stocks can be visualized.

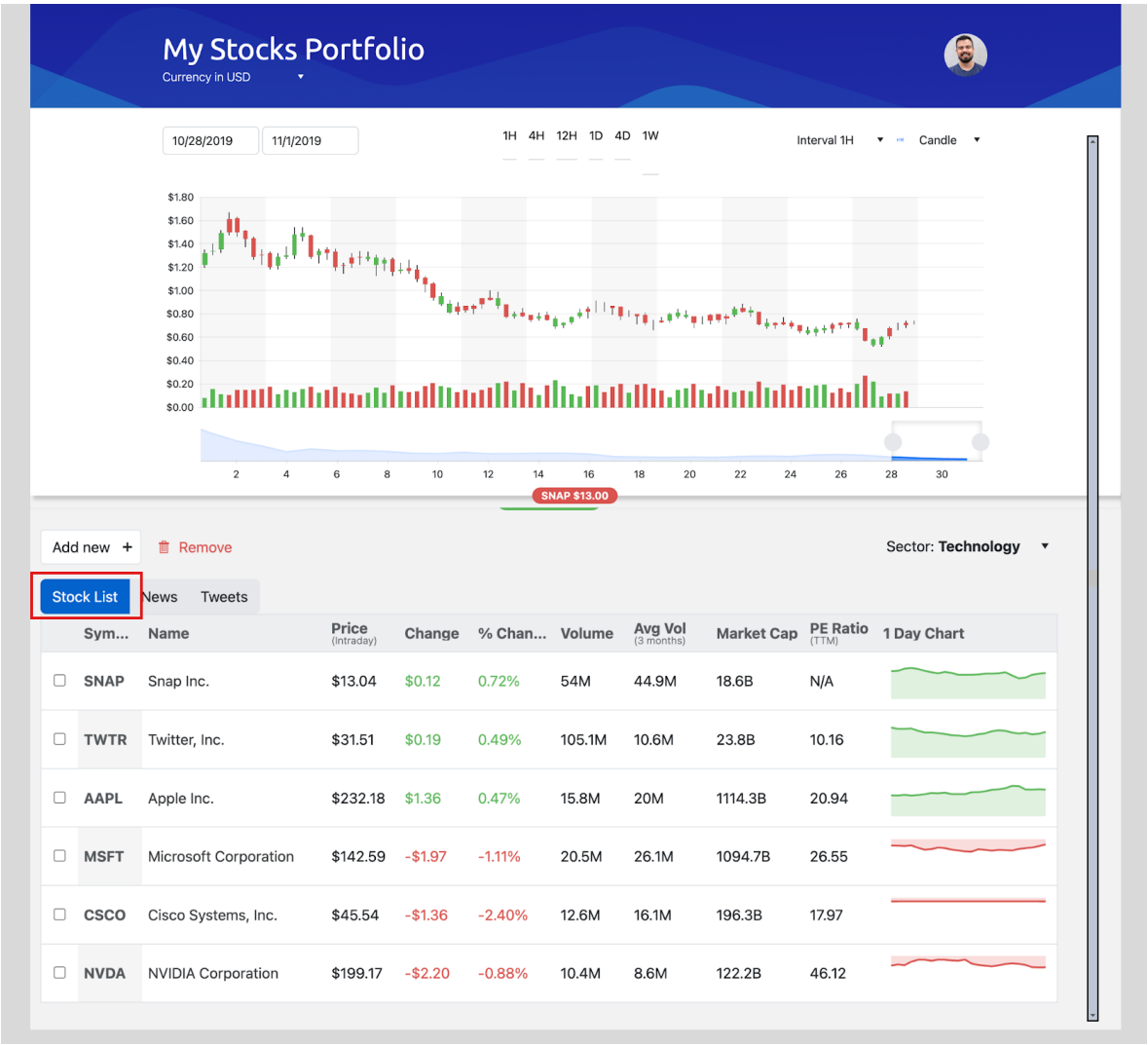

Figure 4: Dashboard with Stocks UI

The recent news data about a company can be visualized by changing the tab to news tab as shown in Figure 5 . The news article with links is shown sorted based on time and sentiment analysis is performed on the content. The sentiment helps investors decide whether to buy stocks of a particular company or to avoid it. The bar represents the sentiment range from negative (red), neutral (yellow) and positive (green).

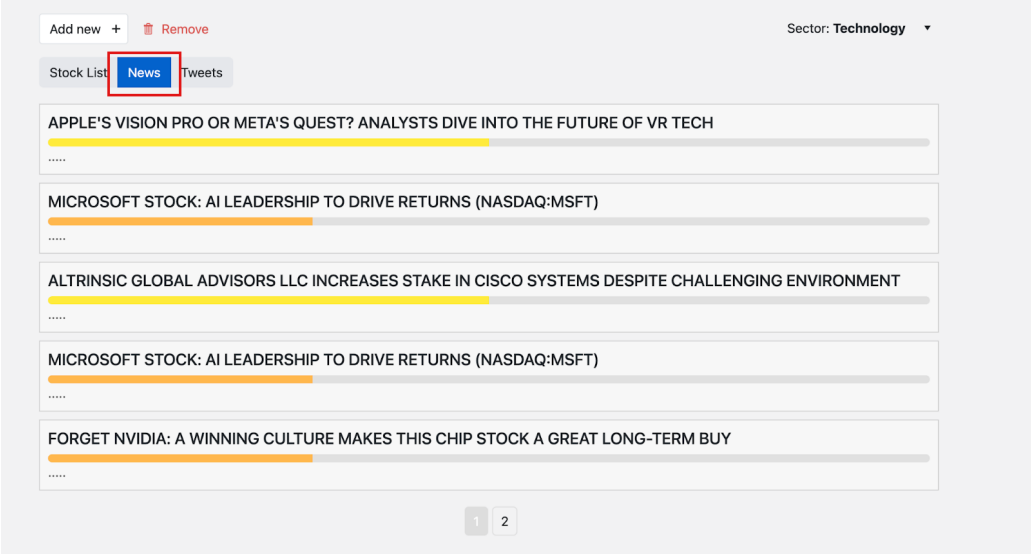

Figure 5: News UI

Similarly the recent tweets about the company can be observed by selecting the Tweets tab.

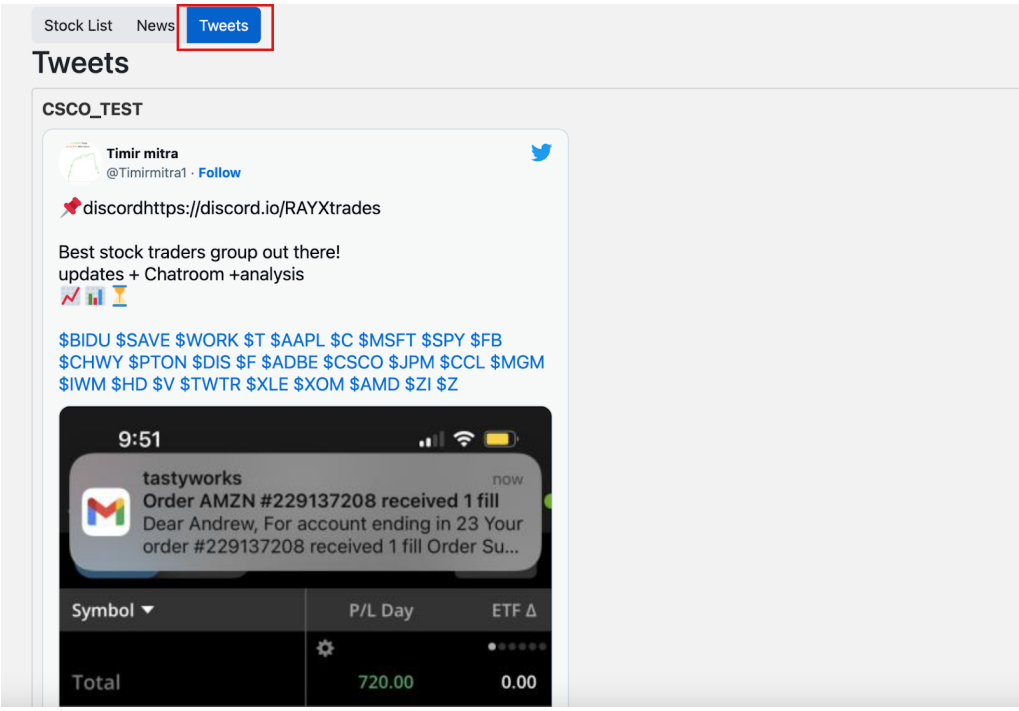

Figure 6: Tweets UI

5. Evaluation Experiments

The following are the results of our evaluation tests which show the latency and total data flow through the network achieved in the system. We have done a comparative analysis of latency achieved with and without load shedding by configuring the load shedding parameter. The input is the number of news articles ingested with the output being the latency time in seconds. As we can see the latency levels are similar for smaller input sizes but as the input size rises the latency in case of load shedding is drastically reduced. Similarly, as we increase data to be shredded, the latency decreases further. Also, for the data flow through the system we can observe that with increase in input traffic (burst traffic) , similar patterns are visible. With higher data shedded, less data flows through the network.

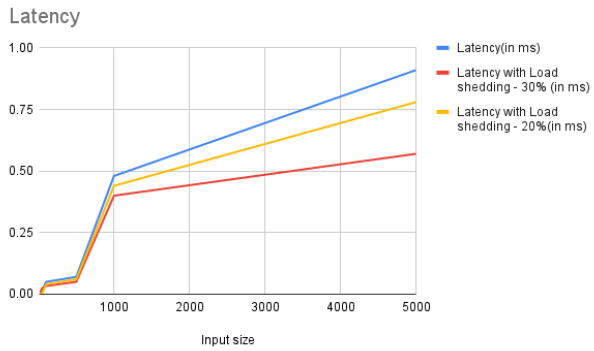

Figure 7: Latency

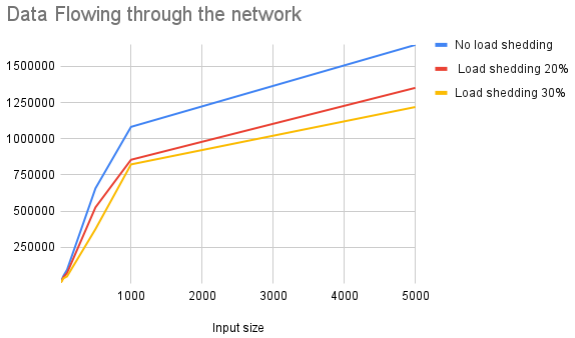

Figure 8: Data Flowing

6. Conclusion and Future Work

In summary, Apache Flink and Apache Kafka with their complementary features and capabilities make them a powerful combination for building robust and scalable real-time data processing systems. Together, they enable us to build a system that can handle the challenges of real-time data processing at scale such as varied frequencies, diverse sources and huge bursts of input data. By leveraging these technologies, we have created a dynamic and responsive dashboard that provides timely information and analysis, empowering users to make informed decisions based on the latest market trends.

The future work in this direction can be to implement more load shedding techniques such as selectivity based load shedding. Analysis can be performed on the balance between the amount of data that can be shredded without losing out much on the overall stock sentiment information. Further, the sentiment analysis of multiple news articles and tweets can be combined to show overall sentiment of the stock. The ability to filter information based on sentiment can also be done. Right now, we only have simulated data for stocks, moving ahead, we can try to incorporate real time actual data. Also, elasticity techniques can be implemented for scalability.

7. References

[1] Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), pp.107-113.
[2] Camacho-Rodríguez, J., Chauhan, A., Gates, A., Koifman, E., O'Malley, O., Garg, V., Haindrich, Z., Shelukhin, S., Jayachandran, P., Seth, S. and Jaiswal, D., 2019, June. Apache hive: From

MapReduce to enterprise-grade big data warehousing. In Proceedings of the 2019 International Conference on Management of Data (pp. 1773-1786).
[3] Isah, H., Abughofa, T., Mahfuz, S., Ajerla, D., Zulkernine, F. and Khan, S., 2019. A survey of distributed data stream processing frameworks. IEEE Access, 7, pp.154300-154316.
[4] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. and Tzoumas, K., 2015. Apache Flink: Stream and batch processing in a single engine. The Bulletin of the Technical Committee on Data Engineering, 38(4).
[5] Kreps, J., Narkhede, N. and Rao, J., 2011, June. Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (Vol. 11, No. 2011, pp. 1-7).
[6] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. PVLDB, 12(2):120--139, 2003.
[7] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14). Association for Computing Machinery, New York, NY, USA, 318–321. https://doi.org/10.1145/2611286.2611314
[8] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. PVLDB, 10(12):1718--1729, 2017.
[9] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. 2003. Aurora: a data stream management system. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03). Association for Computing Machinery, New York, NY, USA, 666. https://doi.org/10.1145/872757.872855
[10] Zhao, Bo & Hung, Nguyen & Weidlich, Matthias. (2020). Load Shedding for Complex Event Processing: Input-based and State-based Techniques. 1093-1104. 10.1109/ICDE48307.2020.00099.
[11] Fragkoulis, M., Carbone, P., Kalavri, V., & Katsifodimos, A. (2020). A Survey on the Evolution of Stream Processing Systems. ArXiv, abs/2008.00842.

Attaching the survey paper:

Survey on Distributed Data Stream Processing Middleware

Alekhya Pyla Rajasekhar Mekala Antriksh Ganjoo
{apyla, rmekala, ganjooa}@uci.edu

1. Abstract

Stream analytics has become increasingly important in industries due to the explosion of data generated from various sources such as IoT devices, social media, sensors, and more. Real-time insights from these data sources help businesses make better decisions, improve customer experiences, and enhance operational efficiency. Recent demands for scalable and low latency have revealed various shortcomings of traditional batch processing systems such as MapReduce [5] and Hive [6]. Batch processing systems suffer from latency problems due to the need to collect input data into batches before it can be processed. Thus, Stream processing allows for quicker data analysis, reducing downtime and improving overall reliability. Further, the adoption of stream analytics is expected to continue to grow as businesses increasingly rely on

data to drive their operations and gain a competitive edge. However, finding the right technologies for stream analytics tasks can be challenging due to the complexity of processing which involves tools for ingesting, processing, storing, indexing, and managing streaming data. Additionally, selecting the appropriate tools and platforms to integrate the various data sources can be a daunting task. To address this issue, in this paper, we present a study of distributed data stream processing frameworks of Spark Streaming, Apache Flink, and Kafka Streams along with systems of RabbitMQ and Kafka which are used in the data ingestion part of streaming applications.

2. Data Stream Processing System

We start by looking at the architecture of a stream processing system which consists of various components as shown in Figure 1.

- A data stream ingestion layer responsible for accepting streams of data
- A data stream processing layer, which processes and analyses data, typically in one or more steps, such as filtering, aggregating, or enriching the data.
- A storage layer is responsible for storing, indexing, and managing both the raw data and the generated knowledge or insights.
- A cluster management layer coordinates the functions of distributed computing and storage resources, ensuring that the system can handle large volumes of data and scale as needed.
- An output layer directs the output data stream and generated knowledge to other downstream systems or visualization tools, where it can be used for further analysis or decision-making.

In this paper, we focus on the famous technologies used in the Ingestion layer coupled with the streaming frameworks available.

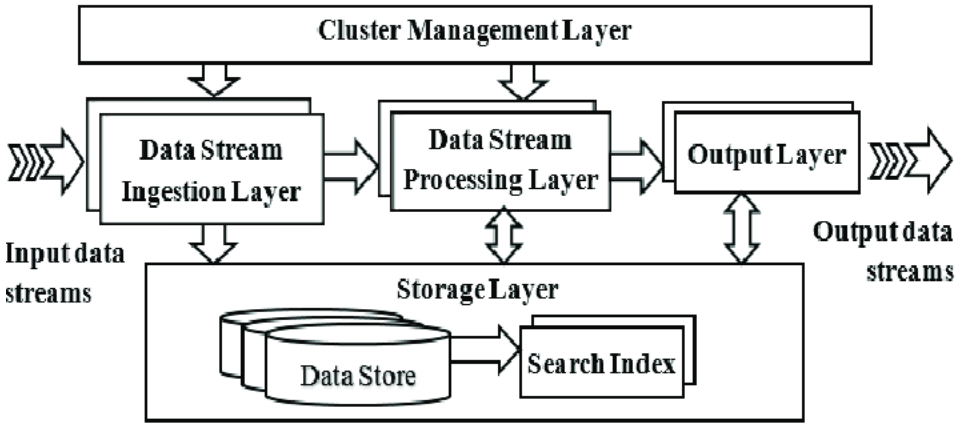

Figure 1. The architecture of a Data Stream Processing System (DSPS)[1].

2. Message Queues for Data Ingestion Layer

The process of receiving data streams from its source to the processing or storage system is known as data ingestion. It is crucial to perform this process efficiently and correctly. The data ingestion layer is responsible for ensuring that data is distributed across the architecture in a scalable, resilient, and fault-tolerant manner from multiple input data streams. Additionally, it decouples the input data sources, providing better management and control. When selecting a data stream ingestion layer, it is important to consider factors such as scalability, reliability, ease of integration, and support for various data formats and protocols. Message queues are often

used as a means of data ingestion in data stream processing systems. In this section, we look into the message queue architectures used in today's world, Apache Kafka and RabbitMQ.

2.1 RabbitMQ:

RabbitMQ was first released in 2007 by Rabbit Technologies Ltd. RabbitMQ is a messaging broker, which provides a mechanism for different applications to communicate with each other by sending and receiving messages. It is an open-source implementation of the Advanced Message Queuing Protocol (AMQP) and is written in the Erlang programming language. RabbitMQ works by providing a central message broker, which acts as a mediator between the different applications. The message broker is responsible for receiving messages from one application and routing them to the appropriate destination application. It also provides several features, such as message queuing, message persistence, and message routing, which help to ensure that messages are delivered reliably and efficiently.

One of the key features of RabbitMQ is its support for different messaging patterns. These patterns define how messages are exchanged between different applications. Some common messaging patterns include publish/subscribe, request/reply, and point-to-point messaging. RabbitMQ supports all of these patterns, and developers can choose the pattern that best suits their application's needs. Another important feature of RabbitMQ is its support for message queuing. In a message queuing system, messages are stored in a queue until they can be processed by a receiver. This allows messages to be sent even if the receiver is not currently available. When the receiver becomes available, it can process the messages in the order that they were received. This helps to ensure that messages are not lost or dropped if the receiver is temporarily unavailable.

In summary, RabbitMQ is a messaging broker that provides a way for different applications to communicate with each other by sending and receiving messages. Its features such as message queuing and message persistence ensure that messages are delivered reliably and efficiently. RabbitMQ is widely used in a variety of industries, including finance, healthcare, and telecommunications, and is an essential component of many distributed application architectures.

2.2 Apache Kafka:

Kafka[8] is a distributed streaming platform that is used to build real-time data pipelines and streaming applications. It was originally developed by LinkedIn in 2011 and is now maintained by the Apache Software Foundation. At its core, Kafka is a distributed publish-subscribe messaging system. It allows producers to publish messages on a topic, and consumers to subscribe to one or more topics and receive notifications from them in real time. Kafka is designed to handle large volumes of data and high throughput, making it ideal for use cases such as stream analytics in real-time, log aggregation, and event-driven architectures.

Kafka is based on a few key concepts, including topics, partitions, producers, and consumers. A topic is a category or feed name to which messages are published, and consumers can subscribe to one or more topics to receive messages. Topics are divided into partitions, which allow for parallel processing and scalability. Producers are responsible for publishing messages about topics, while consumers are responsible for consuming messages from topics.

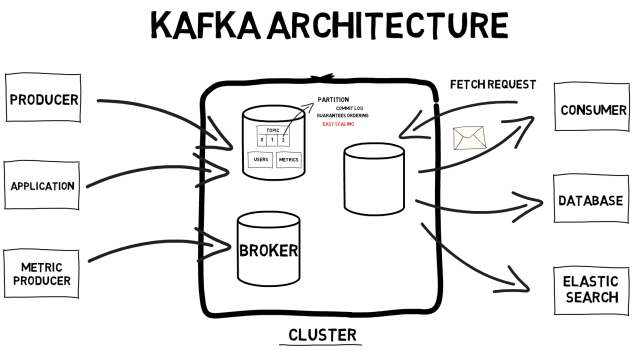

Figure 2: Kafka architecture Block Diagram[11]

Kafka improved on RabbitMQ in several ways. The main advantage of Kafka is its support for stream processing. Kafka provides APIs for processing data streams in real time, allowing developers to build real-time applications that can respond to events as they occur. Kafka also integrates with popular stream processing frameworks such as Apache Spark, Apache Flink, and Apache Samza, making it easy to build complex streaming applications. Further, improvement is the ability to handle large volumes of data at high velocity. Kafka is designed to be horizontally scalable, meaning that it can handle large volumes of data by adding more machines to the cluster. It also supports fault tolerance through replication, where multiple copies of data are stored across different machines to ensure that data is not lost in the event of a failure.

Kafka has a number of use cases across a variety of industries. For example, in finance, it is used for real-time trading and risk management applications, and in e-commerce, it is used for real-time inventory management and recommendation engines.

Both Kafka and RabbitMQ have their strengths and weaknesses and the choice of message broker depends on the specific requirements of the application. Kafka is a better choice for high-throughput, low-latency applications that require fault tolerance and scalability, while RabbitMQ is a better choice for applications that require more complex messaging patterns and resource efficiency.

3. Data Stream Processing Engines:

Data stream processing engines (DSPEs) are software frameworks that enable the processing of continuous data streams in real-time or near real-time. In this section, we look at some representative Data Stream Processing Engines(DSPEs) selected based on their popularity and wide practical usage in the industry, namely Spark Streaming, Apache Flink, and Kafka Streams.

3.1 Spark Streaming

Apache Spark Streaming[7], initially released in 2013 is a scalable fault-tolerant streaming processing system that natively supports both batch and streaming workloads. Spark Streaming is an extension of the core Spark API. It allows developers to build stream processing applications using the same programming model and APIs as batch processing with Apache Spark, by dividing the data into small batches and processing them using Spark's powerful

distributed computing engine. This enables developers to build complex stream processing applications, as they can leverage their existing knowledge of Apache Spark.

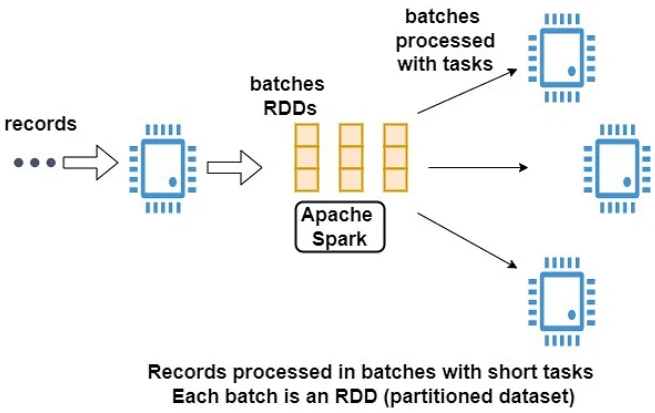

Figure 3: Spark Streaming architecture[9]

Spark Streaming architecture is based on the concept of Discretized Streams (DStreams), which is a sequence of RDDs (Resilient Distributed Datasets). Each RDD in a DStream represents a batch of data that is processed in parallel. Spark Streaming ingests data from various sources, such as Kafka, Flume, HDFS, and TCP sockets, and divides the data into small batches, which are processed by Spark's batch processing engine. The processed results are then sent to the output sink, such as HDFS, Cassandra, or JDBC. This allows Spark Streaming to seamlessly integrate with any other Spark components like MLlib and Spark SQL. Its Single execution engine and unified programming model for batch and streaming lead to some unique benefits over other traditional streaming systems.

Spark Streaming supports both event-time and processing-time semantics, which allows developers to choose the appropriate time mode based on their use case. Spark Streaming also provides support for exactly-once processing semantics, which guarantees that each input record is processed exactly once.

3.2 Apache Flink

Apache Flink [4] is a powerful and popular open-source stream processing framework founded in 2014. Apache Flink is developed under the Apache License 2.0 by the Apache Flink Community within the Apache Software Foundation.

Flink provides a unified programming model for both batch and stream processing, unlike previous systems. The previous batch processing approaches suffered from high latency and inaccuracy in results due to the splitting of the data basis into a time dimension. Flink aims to solve this problem such that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipelined fault-tolerant dataflows. Further, Flink has an advanced state management system that allows developers to maintain state across batch and stream processing. This was a significant improvement over previous systems, which required developers to maintain the state manually.

Flink sources generate data for a data stream while sinks consume data from a data stream such as Apache Kafka or local file systems. Flink can build end-to-end streaming applications where sources read data, apply processing logic, and output the results to sinks for storage or further analysis. The centerpiece of Flink is its distributed dataflow engine, designed to execute dataflow programs. Flink runtime programs consist of a directed acyclic graph (DAG) of stateful operators connected with data streams. Flink provides two key APIs: the DataSet API designed for batch processing and the DataStream API for stream processing.

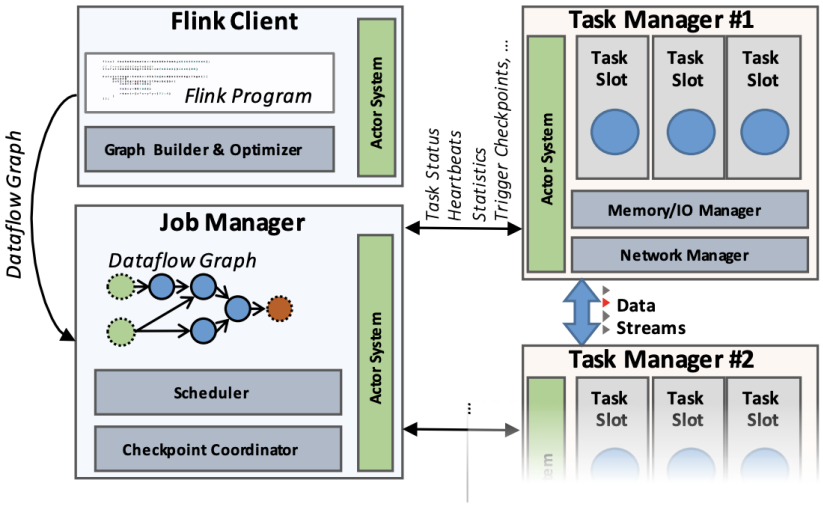

Figure 4. Apache Flink Data Flow[4]

Flink provides reliable execution with strict exactly-once-processing guarantees through checkpointing and partial re-execution. Data sources must be persistent and replayable, such as files and message queues like Kafka. It supports three-time modes event time, ingestion time, and processing time. By supporting different time modes, Flink allows developers to choose the appropriate time mode based on their use case, and achieve the desired level of correctness and performance in their data processing applications. Flink has numerous applications across industries including finance, healthcare, retail, and telecommunications. Its common use cases include real-time analytics, fraud detection, recommendation systems, and IoT data processing.

There are several ways in which Flink improved on Spark Streaming such as its capability to process both batch and streaming data providing a flexible model compared to Spark Streaming, based on a micro-batch processing model, which can result in higher latency and reduced throughput. Flink has better integration with other systems such as Kafka and Hadoop. Flink's connectors for these systems are more mature and offer better performance than Spark Streaming's connectors.

3.3 Kafka Streams

Kafka Streams[8] is another important and popular stream processing library introduced in May 2016 (0.10 release) to build real-time streaming applications that can process and analyze data in real-time using Kafka as its source of data. Kafka Streams provides a simple and lightweight API for building streaming applications that can perform various operations such as filtering,

aggregation, joining, and transforming data. Kafka Streams is tightly integrated with Apache Kafka, and it leverages its distributed architecture, fault-tolerance, and scalability features along with high-throughput and low-latency messaging for client applications. It also supports stateful processing, which enables developers to maintain state across processing instances.

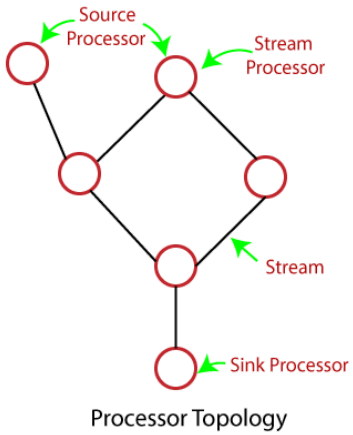

Figure 5. Kafka Stream Processing Topology[10]

The core architecture of the Kafka Streams API includes the stream processor, which is a node in a processing topology. A processing topology is a graph of stream processors (as shown above) that specifies how input streams are transformed into output streams. In addition, Kafka Streams also provides support for stateful stream processing, which allows stream processors to maintain local states and perform complex data processing operations. This is achieved through the use of a state store, which is essentially a local key-value store that is maintained by each stream processor. Kafka Streams also supports windowing operations for aggregating data over some time. Overall, Kafka Streams' architecture provides a flexible and scalable platform for building real-time stream processing applications. The key idea behind Kafka Streams is that it processes data in real-time by leveraging the Kafka topic as a durable and fault-tolerant storage layer. This means that all data processing is performed directly on the Kafka topic, without the need for any additional storage layers or external dependencies.

Similar to Flink, Kafka Streams also provides exactly-once processing guarantees through its internal architecture. Kafka Streams uses a combination of processing-time and record-time timestamps to achieve exactly-once semantics. When a record is processed by a Kafka Streams application, it is marked as processed and its offset is committed back to Kafka. This ensures that in case of any failure, the same record is not processed again. Kafka Streams also uses a transactional write-ahead log, where all state updates are written to a transactional topic in Kafka. In case of a failure, the application can restore its state from the transactional topic. Kafka Streams also provides fault tolerance by using standby replicas, which are kept in sync with the active application instance. In case of a failure, the standby replica can take over the processing without any data loss. The internal state of Kafka Streams is also partitioned and distributed across the application instances, ensuring scalability and fault tolerance.

4. Auto-scaling and elasticity in stream processing systems

Challenges in industries like financial trading require data stream processing systems with low latency and high throughput. Existing solutions, such as Apache Flink and Stream processing,

need to scale out across multiple hosts to meet these challenges. An ideal system should have elasticity, meaning it can dynamically acquire or release hosts to handle workload changes, including unexpected peaks and improve overall system utilization. The auto-scaling technique is the key decision-making component in such an elastic scaling system. It determines when the system needs to scale in or out by comparing the current system utilization with predefined upper and lower thresholds. There are different methods for auto-scaling, including local thresholds, global thresholds, and reinforcement learning.

In Demo[12], a system for operator placement in elastic scaling was proposed where operators are monitored for CPU, RAM, and network consumption. The auto-scaling technique assesses if the system is overloaded or underloaded based on these measurements. For an overloaded host, a subset of its operators is selected and reassigned to another non-overloaded host. An underloaded host is released by redistributing its operators to other hosts.

In threshold-based approaches, the auto-scaling technique is triggered when the system utilization exceeds the upper threshold for a certain number of consecutive measurements, marking the system as overloaded. Conversely, if the utilization drops below the lower threshold for a consecutive number of measurements, a host is marked as underloaded. To avoid frequent scaling decisions, certain actions are implemented, such as a grace period after each scaling decision and scaling decisions made only after a certain number of consecutive threshold violations. During scale-out decisions, the load to move is chosen to reduce the utilization below the target level.

Alternatively, the global threshold-based approach calculates the average load of all running hosts. If this average load surpasses an upper limit, the system is marked as overloaded, and each host is assigned a portion of the load to be moved. If the lower threshold is exceeded, the host with the minimal load is released, and its operators are redistributed.

Another approach implemented is based on reinforcement learning, which models the system using states and possible actions. The algorithm selects the action with the highest expected reward, learned through online learning algorithms. While simple threshold-based approaches may appear intuitive, incorporating adaptive techniques allows for maximizing system utilization and achieving comparable end-to-end latency.

5. Load shedding in Stream Processing systems

In streaming data processing, stream processors face challenges when they have no control over the rate of incoming events, which can result in performance degradation when the input rates exceed the system's capacity. To mitigate this issue, load shedding is employed as a strategy to sustain the load by temporarily discarding excess tuples from inputs or intermediate operators in the streaming execution graph. Load shedding involves a trade-off between result accuracy and sustainable performance, making it suitable for applications with strict latency constraints that can tolerate approximate results. Accurately detecting overload situations is crucial to prevent unnecessary degradation of results. Load-shedding components rely on execution statistics to make informed decisions. In many stream processors, load shedding is limited to a predefined set of operators that do not modify tuples, such as filter, union, and join. A load-shedding road map (LSRM) can be utilized, which is a pre-computed table containing prioritized load-shedding plans based on the expected amount of shedding they will cause.

In traditional data stream processing, load shedding is commonly achieved by discarding stream elements based on their estimated utility for the query result. There are two main approaches: input-based shedding and state-based shedding. Input-based shedding strategies, such as Random input shedding (RI) and Selectivity-based input shedding (SI), discard events randomly or based on the query selectivity per event type. For example, Kafka implements Random input shedding. However, input-based shedding may not be suitable for complex event processing (CEP) queries, as the utility of a stream element heavily depends on the current state of query evaluation. Complex event processing (CEP) is an efficient and scalable paradigm for processing event streams. To address this, state-based load shedding is introduced, which discards partial matches to maximize the recall of query processing during overload situations. Random state shedding (RS) and Selectivity-based state shedding (SS) are two strategies inspired by techniques used in approximate CEP.

While state-based shedding offers finer control compared to shedding input events, input-based shedding is generally more efficient because discarded events are not processed at all, whereas partial matches have already incurred computational effort. Therefore, to strike the right balance between result quality and efficiency for specific applications, a hybrid approach that combines state-based and input-based load shedding has been proposed. This approach, outlined in [14], introduces a cost model for hybrid load shedding that quantifies the utility of partial matches, directly linked to the utility of input events. The problem can be formulated as a knapsack problem, and shedding strategies are designed based on its solution.

In related work, a lightweight and efficient load-shedding strategy called eSPICE [15] is proposed. This strategy utilizes a probabilistic model to determine the importance of events within a window, taking into account the dependencies between events of the same pattern as well as their order in the pattern and input event streams. The importance of an event is influenced by its type and its relative position within the window. The core idea behind eSPICE is that events occurring at specific positions within a window, which contribute to constructing a complex event in one window, are also likely to contribute to complex events in other windows. To maintain a given latency bound, the authors provide an algorithm that estimates the number of events to be dropped and the intervals at which these drops should occur.
By considering event importance and leveraging the correlations between events across windows, eSPICE offers an effective approach to determine which events can be safely discarded while preserving the desired latency constraints.

6. Control Plane and Data Plane

In distributed stream processing systems, there are two main components: the control plane and the data plane. The control plane manages the system's operation, handling tasks like job submission, resource allocation, load balancing, fault tolerance, and scaling. It makes high-level decisions to optimize resource utilization and performance based on system monitoring and metrics. On the other hand, the data plane processes the data streams by executing computational tasks on distributed processing nodes. It performs real-time data transformations, filtering, aggregations, and other specified operations. The data plane ensures the timely delivery of results.

Together, the control plane and data plane enable efficient and effective stream processing. The control plane handles management and coordination, while the data plane carries out the actual

data processing tasks. This separation allows for scalability, fault tolerance, and optimized resource usage in distributed stream processing systems.

The Chi control-plane design [16] addresses the challenges of configuring and tuning modern streaming systems in dynamic environments. By embedding control-plane messages within data-plane channels, Chi achieves a low-latency and flexible control plane. It introduces a reactive programming model that allows for asynchronous execution of control policies, eliminating the need for global synchronization. Dataflow controllers in the system monitor data flow behavior and external changes, triggering control operations when necessary. Users define and submit control operations to these controllers, which follow a three-phase control loop. This enables supporting important streaming requirements such as frequent reconfigurations without compromising on ease of use.

The control loop involves the controller making control decisions and creating unique control operations for each operator in the data flow. These operations are serialized into control messages and broadcast to the source operators. Control messages travel through the data flow, triggering control actions at each operator and allowing for the attachment of additional data. Sink operators send the control messages back to the controller for post-processing. This approach enables continuous monitoring, feedback, and dynamic reconfiguration, catering to various production use cases.

6. Back Pressure and Flow Control

Back Pressure and flow control play crucial roles in distributed stream processing systems, ensuring efficient data processing by addressing the mismatch between data producers and consumers. Backpressure is a feedback mechanism where overwhelmed downstream components send signals to slow down or reduce data production, maintaining system stability. It propagates through the processing pipeline, reaching upstream operators and even the data sources. To prevent data loss, persistent input message queues and sufficient storage space are necessary.

Flow control manages data flow to prevent congestion and optimize resource utilization. It regulates the rate of data transmission between operators by buffering data, applying rate-limiting techniques, or dynamically adjusting transmission rates based on system capacity and workload. Buffer-based back pressure controls data flow indirectly by monitoring buffer availability and gradually filling up buffers along the dataflow path of bottleneck operators.

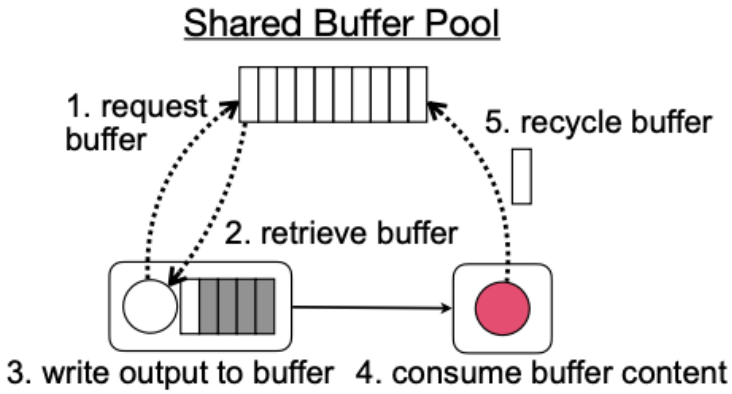

Fig 6: Buffer-based backpressure [17]

Credit-based flow control (CFC), derived from ATM network switches, is a per-channel congestion control technique that uses a credit system to signal buffer availability from receivers to senders. This technique, employed in modern stream processors like Apache Flink, effectively manages loads and improves performance in highly parallel processing systems.

By leveraging back pressure and flow control mechanisms, distributed stream processing systems can handle varying workloads, prevent bottlenecks, and ensure controlled and efficient data processing throughout the system.

7. Summary:

In summary, the three systems discussed above have some strengths and weaknesses to offer. While Data partitioning in Spark Streaming distributes data automatically into partitions, Flink has one or more partitions and can use partitioning strategies such as hashing or random re-partitions. Kafka Streams uses partitioning to enable data locality, scalability, high performance, and fault tolerance. State management in Structured(Spark) Streaming and Flink use external storage systems, while Kafka Streams provides state stores that can be used to store and query data. Processing guarantees for Spark Streaming are exactly-once processing using a micro-batch processing engine. Flink guarantees exactly-once state updates, while Kafka Streams added end-to-end exactly-once processing semantics in the latest versions. Fault tolerance for Structured Streaming is through checkpointing and a write-ahead log, Flink uses a combination of stream replay and checkpointing, and Kafka Streams leverages the fault-tolerance capability offered by the Kafka consumer to handle failures and automatically restarts tasks in the remaining instances of the application.

For efficient and reliable stream processing, the current applications implement techniques of load management such as auto-scaling, load-shedding, backpressure, and flow control and control planes. Research is also done extensively on the fault-tolerance and replication capabilities of these systems. The future directions not discussed in these papers can be investigating privacy and security issues in stream processing, including techniques for protecting sensitive data and detecting anomalies and security breaches in real time. Also, oftentimes understanding these complex frameworks is challenging. So, work can be done in

the direction to simplify the development and usage of stream processing applications for non-expert users.

8. References:

1. Isah, H., Abughofa, T., Mahfuz, S., Ajerla, D., Zulkernine, F. and Khan, S., 2019. A survey of distributed data stream processing frameworks. IEEE Access, 7, pp.154300-154316.
2. Dobbelaere, P. and Esmaili, K.S., 2017, June. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In Proceedings of the 11th ACM international conference on distributed and event-based systems (pp. 227-238).
3. Sharvari, T. and Sowmya Nag, K., 2019. A study on modern messaging systems-Kafka, rabbitmq, and nats streaming. CoRR abs/1912.03715.
4. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. and Tzoumas, K., 2015. Apache Flink: Stream and batch processing in a single engine. The Bulletin of the Technical Committee on Data Engineering, 38(4).
5. Dean, J. and Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), pp.107-113.
6. Camacho-Rodríguez, J., Chauhan, A., Gates, A., Koifman, E., O'Malley, O., Garg, V., Haindrich, Z., Shelukhin, S., Jayachandran, P., Seth, S. and Jaiswal, D., 2019, June. Apache hive: From MapReduce to enterprise-grade big data warehousing. In Proceedings of the 2019 International Conference on Management of Data (pp. 1773-1786).
7. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J. and Ghodsi, A., 2016. Apache spark: a unified engine for big data processing. Communications of the ACM, 59(11), pp.56-65.
8. Kreps, J., Narkhede, N. and Rao, J., 2011, June. Kafka: A distributed messaging system for log processing. In Proceedings of the NetDB (Vol. 11, No. 2011, pp. 1-7).
9. https://www.projectpro.io/article/spark-streaming-example/540
10. https://kafka.apache.org/0102/documentation/streams/core-concepts
11. https://finematics.com/apache-kafka-explained/
12. Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS '14). Association for Computing Machinery, New York, NY, USA, 318–321. https://doi.org/10.1145/2611286.2611314
13. N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stone- braker. Load shedding in a data stream manager. In VLDB, 2003
14. Zhao, Bo & Hung, Nguyen & Weidlich, Matthias. (2020). Load Shedding for Complex Event Processing: Input-based and State-based Techniques. 1093-1104. 10.1109/ICDE48307.2020.00099.
15. Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2019. ESPICE: Probabilistic Load Shedding from Input Event Streams in Complex Event Processing. In Proceedings of the 20th International Middleware Conference (Middleware '19). Association for Computing Machinery, New York, NY, USA, 215–227. https://doi.org/10.1145/3361525.3361548
16. Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: a scalable and programmable control plane for distributed stream

processing systems. Proc. VLDB Endow. 11, 10 (June 2018), 1303–1316. https://doi.org/10.14778/3231751.3231765
17. Fragkoulis, M., Carbone, P., Kalavri, V., & Katsifodimos, A. (2020). A Survey on the Evolution of Stream Processing Systems. ArXiv, abs/2008.00842.