International Journal of Leading Research Publication (IJLRP)
E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

Stream Processing Internals and Usecases

Arjun Reddy Lingala

arjunreddy.lingala@gmail.com

Abstract

Batch processing is widely used concept in data warehousing where many companies build analytical solutions deriving insights into their systems and building new products based on the analysis based on various aspects of the systems. The exponential growth of real-time data sources like IoT sensors, social media has necessitated systems capable of processing unbounded data streams with low latency, high throughput, and guaranteed correctness. Unlike batch processing, stream process- ing engines must handle continuous data flows with dynamic arrival patterns, out-of-order events, and variable workloads. The problem with daily batch processes is that changes in the input are only reflected in the output a day later, which is too slow for some use cases. To reduce the delay, we can run the processing more frequently. In the batch processing world, the inputs and outputs of a job are files may be on distributed file system like HDFS [1] or Amazon S3 [2]. Stream processing has emerged as a critical computational model that enables real-time ingestion, transformation, and analysis of continuous data streams. This paper presents a comprehensive exploration of the necessity for stream processing, identifying its use cases over batch processing and its suitability for latency-sensitive applications such as financial trading, fraud detection, and Internet of Things (IoT) systems. We begin by establishing the fundamental motivation behind stream processing, outlining key challenges associated with real-time data analytics, including data velocity, system scalability, and fault tolerance. The discussion highlights the lim- itations of traditional batch processing frameworks like Apache Hadoop and their inability to efficiently handle continuous data flows. In contrast, we analyze how stream processing frameworks such as Apache Kafka [3], Apache Flink [4], Apache Storm [5], and Spark Streaming [6] address these challenges by enabling near real-time event-driven computations.

Keywords: Stream Processing, Data warehouse, Windowing, Change Data Capture, Message Queues, Message Brokers, Kafka, Flink, Storm, Real-time

I. INTRODUCTION

The ever-growing volume of real-time data generated across industries such as finance and IoT has exposed the limitations of traditional batch-processing models. These conventional approaches, which process data in discrete intervals, struggle to keep up with the increasing need for instant insights, low-latency decision-making, and high-throughput analytics. In contrast, stream processing has emerged, allowing for continu- ous data ingestion, real-time computation, and rapid responses to events as they occur. This shift is essential for modern applications that require immediate data-driven actions, mak-ing stream processing an indispensable component of today's computing. Traditional batch-processing

frameworks, such as Apache Hadoop, are inherently unsuitable for scenarios that demand real-time responsiveness. They introduce delays due to their scheduled, periodic execution cycles, making them ineffective for applications where milliseconds matter. For instance, stock market fluctuations require immediate analysis to drive automated trading strategies, while cybersecurity systems must detect and mitigate threats in real-time to prevent potential breaches, bank transactions fraud detection must be detected in real-time to prevent scams. To overcome these constraints, stream processing frameworks such as Apache Flink [4], Apache Kafka Streams [3], Apache Storm [5], and Spark Streaming [6] have been developed to enable continuous, event-driven computations with low latency, high availability, and fault tolerance. The proliferation of connected devices, sensors, and digital interactions has resulted in an influx of continuous data that traditional batch processing models cannot efficiently handle. Industries such as fraud detection, predictive maintenance, network security, and algo- rithmic trading require immediate processing and response to data as it arrives. Modern stream processing solutions leverage distributed computing, ensuring efficient scalability and fault tolerance while handling vast, high-velocity data streams. Many modern applications rely on event-driven architectures where decisions are triggered by real-time events, requiring seamless data processing pipelines that react instantly. Stream processing frameworks are designed to integrate with existing big data architectures, including cloud data lakes, NoSQL databases, and enterprise data pipelines, enhancing operational efficiency.

II. STREAM PROCESSING INTERNALS

In continual processing methodology with low delays polling becomes expensive with traditional relational databases as they are not designed for these kind of use cases. Relational databases commonly have triggers, which can react to a change but they are very limited in what they can do and have been somewhat of an afterthought in database design. New specialized tools like messaging systems are developed where a producer sends a message containing the event, which is then pushed to consumers. A direct communication channel like a Unix pipe or TCP connection between producer and consumer would be a simple way of implementing a mes- saging system. However, most messaging systems expand on this basic model. In particular, Unix pipes and TCP connect exactly one sender with one recipient, whereas a messaging system allows multiple producer nodes to send messages to the same topic and allows multiple consumer nodes to receive messages in a topic.

A. Message Queues

A widely used tool for sending messages is via a message broker which is also known as a message queue which is essentially a kind of database that is optimized for handling message streams. It runs as a server, with producers and con- sumers connecting to it as clients. Producers write messages to the broker, and consumers receive them by reading them from the broker. By centralizing the data in the broker, these systems can more easily tolerate clients that connect, disconnect, and crash, and broker handles the problem of durability. Some message brokers only keep messages in memory, while others write them to disk so that they are not lost in case of a broker crash. Faced with slow consumers, they generally allow unbounded queueing, although this choice may also depend on the configuration. A consequence of queueing is also that consumers are generally asynchronous: when a producer sends a message, it normally only waits for the broker to confirm that it has buffered the message and does not wait for the message to be processed by consumers. The delivery to consumers will happen at some undetermined future point in time, but sometimes significantly later if there is a queue backlog.

Databases usually keep data until it is explicitly deleted, whereas most message brokers automatically delete a message when it has been successfully delivered to its consumers. Such message brokers are not suitable for long-term data storage. This is the traditional view of message brokers, which is encapsulated in standards like JMS [7] and implemented in software like RabbitMQ, and Google Cloud Pub/Sub [8].

When multiple consumers read messages in the same topic, two main patterns of messaging are used. One is load balancing where each message is delivered to one of the consumers, so the consumers can share the work of processing the messages in the topic. The broker may assign messages to consumers randomly. This pattern is useful when the messages are expensive to process, and so you want to be able to add consumers to parallelize the processing. Other one is fan out model where each message is delivered to all of the consumers. Fan-out model allows multiple independent consumers to each subscribe to the same broadcast of messages without affecting each other. Consumers may crash at any time, so it could happen that a broker delivers a message to a consumer but the consumer never processes it, or only partially processes it before crashing. In order to ensure that the message is not lost, message brokers use acknowledgments. A client must explicitly tell the broker when it has finished processing a message so that the broker can remove it from the queue. As the acknowledgment timing may vary consumer to consumer and broker may end up sending the older messages to the consumers again which result in missing the order of mes- sages. Even if the message broker otherwise tries to preserve the order of messages, the combination of load balancing with redelivery inevitably leads to messages being re-ordered which can be avoided by maintaining a separate queue for each consumer. In case of message queues, a new consumer joining at a later stage will only receive messages sent after it is registered and prior messages cannot be recovered from the new consumer standpoint. This problem is addressed using Log-based message brokers.

B. *Log-based Message Brokers*

A log is simply an append-only sequence of records on disk. The same structure can be used to implement a message broker where a producer sends a message by appending it to the end of the log, and a consumer receives messages by reading the log sequentially. If a consumer reaches the end of the log, it waits for a notification that a new message has been appended. In order to scale to higher throughput than a single disk can offer, the log can be partitioned. Different partitions can then be hosted on different machines, making each partition a separate log that can be read and written independently from other partitions. A topic can then be defined as a group of partitions that all carry messages of the same type. Brokers assign a monotonically increasing number within each partition called as offset. Apache Kafka [3] Amazon Kinesis Streams [9], and Twitter's DistributedLog [10] are log-based message brokers that work like this. Google Cloud Pub/Sub [8] is architecturally similar but exposes a JMS-style API rather than a log abstraction. Consuming a partition sequentially makes it easy to tell which messages have been processed, all messages with an offset less than a consumer's current offset have already been processed, and all messages with a greater offset have not yet been seen. Thus, the broker does not need to track acknowledgments for every single message it only needs to periodically record the consumer offsets. The reduced bookkeeping overhead and the opportunities for batching and pipelining in this approach help increase the throughput of log-based systems. If a consumer node fails, another node in the consumer group is assigned the failed consumer's partitions, and it starts consuming messages at the last recorded offset. If the consumer had

processed subsequent messages but not yet recorded their offset, those messages will be processed a second time upon restart.

If we only ever append to the log, you will eventually run out of disk space. To reclaim disk space, the log is actually divided into segments, and from time to time old segments are deleted or moved to archive storage. This means that if a slow consumer cannot keep up with the rate of messages, and it falls so far behind that its consumer offset points to a deleted segment, it will miss some of the messages. Effectively, the log implements a bounded-size buffer that dis- cards old messages when it gets full, also known as a circular buffer or ring buffer. However, since that buffer is on disk, it can be quite large. The throughput of a log remains more or less constant, since every message is written to disk anyway. This behavior is in contrast to messaging systems that keep messages in memory by default and only write them to disk if the queue grows too large and such systems are fast when queues are short and become much slower when they start writing to disk.

III. STREAMING FOR DATABASES

Though databases are developed very prior to streaming systems, we have some use cases where streaming systems can be used to support databases. A replication log is a stream of database write events, produced by the leader as it processes transactions. The followers apply that stream of writes to their own copy of the database and thus end up with an accurate copy of the same data. Infact there is no single system that can satisfy all data storage, querying, and processing needs. Most applications need to combine several different technologies in order to satisfy their requirements. For example, using an OLTP database to serve user requests, a cache to speed up common requests, a full-text index to handle search queries, and a data warehouse for analytics. Each of these has its own copy of the data, stored in its own representation that is optimized for its own purposes. These different representations of data that appears in different places needs to be kept in sync with one another for responding to requests accurately. Change data capture is a mechanism for ensuring that all changes made to the system of record are also reflected in the derived data systems so that the derived systems have an accurate copy of the data. Essentially, change data capture makes one database the leader, and turns the others into followers. A log-based message broker is well suited for transporting the change events from the source database, since it preserves the ordering of messages. Database triggers can be used to implement change data capture by registering triggers that observe all changes to data tables and add corresponding entries to a changelog table. However, they tend to be fragile and have significant performance overheads. LinkedIn's Databus [11], Facebook's Wormhole [12]. Like message brokers, change data capture is usually asynchronous where the system of record database does not wait for the change to be applied to consumers before committing it.

Similarly to change data capture, event sourcing involves storing all changes to the application state as a log of change events. In change data capture, the application uses the database in a mutable way, updating and deleting records at will. The log of changes is extracted from the database at a low level, which ensures that the order of writes extracted from the database matches the order in which they were actually written, avoiding the race condition. In event sourcing, the application logic is explicitly built on the basis of immutable events that are written to an event log. In this case, the event store is append-only, and updates or deletes are discouraged or prohibited. Events are designed to reflect things that happened at the application level, rather than low-level state changes.

IV. STREAM PROCESSING USECASES AND IMPLEMENTATION

Streaming data can be taken as events and write it to a database, cache, search index, or similar storage system, from where it can then be queried by other clients. Database can be kept in sync with changes happening in other parts of the systems especially if the consumer is the only client writing to the database. We can push the events to users in some way, for example by sending email alerts or push notifications, or by streaming the events to a real-time dashboard where they are visualized. We can process one or more input streams to produce one or more output streams. Streams may go through a pipeline consisting of several such processing stages before they eventually end up at an output. Stream processing has long been used for monitoring purposes, like fraud detection systems need to determine if the usage patterns of a credit card have unexpectedly changed, and block the card if it is likely to have been stolen. Manufacturing systems need to monitor the status of machines in a factory, and quickly identify the problem if there is a malfunction. Various types of use cases of stream processing have emerged over time. Complex event processing systems often use a high-level declarative query language like SQL, or a graphical user interface, to describe the patterns of events that should be detected. These queries are submitted to a processing engine that consumes the input streams and internally maintains a state machine that performs the required matching. Another area stream processing is used heavily is for analytics on streams. Some usecases for analytics on streams include measuring the rate of some type of event, calculating the rolling average of a value over some time period.

Stream processors often need to deal with time, especially when used for analytics purposes, which frequently use time windows. In a batch process, the processing tasks rapidly crunch through a large collection of historical events. If some kind of breakdown by time needs to happen, the batch process needs to look at the timestamp embedded in each event. There is no point in looking at the system clock of the machine running the batch process, because the time at which the process is run has nothing to do with the time at which the events actually occurred. Many stream processing frameworks use the local system clock on the processing machine to determine windowing. This approach has the advantage of being simple, and it is reasonable if the delay between event creation and event processing is negligibly short. There are many reasons why processing may be delayed like queueing, network faults, a performance issue leading to contention in the message broker or processor, a restart of the stream consumer, or reprocessing of past events while recovering from a fault or after fixing a bug in the code. Once you know how the timestamp of an event should be determined, the next step is to decide how windows over time periods should be defined. The window can then be used for aggregations. Commonly used windows in stream processing are ***Tumbling Window*** - A tumbling window has a fixed length, and every event belongs to exactly one window. Tumbling Window cab be implemented by a 1-minute tumbling window by taking each event timestamp and rounding it down to the nearest minute to determine the window that it belongs to, ***Hopping Window*** - A hopping window also has a fixed length, but allows windows to overlap in order to provide some smoothing. It can be im- plemented this hopping window by first calculating 1-minute tumbling windows, and then aggregating over several adjacent windows, ***Sliding Window*** - A sliding window contains all the events that occur within some interval of each other. A sliding window can be implemented by keeping a buffer of events sorted by time and removing old events when they expire from the window, ***Session Window*** - Unlike the other window types, a session window has no fixed duration. Instead, it is defined by grouping together all events for the same user that occur

closely together in time, and the window ends when the user has been inactive for some time.

One common solution for fault tolerance in stream process- ing frameworks is to break the stream into small blocks, and treat each block like a mini batch process. This approach is called microbatching, and it is used in Spark Streaming [6]. The batch size is typically around one second, which is the result of a performance compromise. Smaller batches incur greater scheduling and coordination overhead, while larger batches mean a longer delay before results of the stream processor become visible. A variant approach, used in Apache Flink [4], is to periodically generate rolling checkpoints of state and write them to durable storage. If a stream operator crashes, it can restart from its most recent checkpoint and dis- card any output generated between the last checkpoint and the crash. The checkpoints are triggered by barriers in the message stream, similar to the boundaries between microbatches, but without forcing a particular window size.

V. CONCLUSION

The surge in real-time data generation across various indus- tries has necessitated a shift from traditional batch processing to stream processing, which enables continuous data flow han- dling with minimal latency. This paper has comprehensively analyzed why stream processing is essential, highlighting its advantages over batch-oriented methods, especially in scenar- ios requiring instant decision-making, high-speed analytics, and event-driven architectures. In this paper we have discussed event streams, what purposes they serve, and how to process them. In some ways, stream processing is very much like the batch processing, but done continuously on unbounded streams rather than on a fixed-size input. We also discussed types of message brokers, few challenges faced from message brokers and how log-based message queues address these challenges. This paper has also provided an in-depth exploration of stream processing internals, discussing essential concepts such as data ingestion, partitioning, event-time vs. processing-time semantics, windowing techniques, stateful processing, and fault tolerance mechanisms. These architectural elements are crucial for building efficient, scalable, and resilient stream pro- cessing systems. The discussion also extended to distributed computing strategies, including parallelism, checkpointing, fault recovery, and scalability, which ensure the robustness of modern stream processing solutions. Looking toward the future, the evolution of stream processing will continue to be shaped by emerging trends such as the fusion of artificial intelligence (AI) and machine learning (ML) with real-time data analytics. These integrations will enhance automation, predictive insights, and adaptive decision-making capabilities.

REFERENCES

[1] D. Borthakur, ”The Hadoop Distributed File System: Architecture and Design,” The Apache Software Foundation, 2007. [Online]. Available: https://hadoop.apache.org.

[2] Amazon Web Services, ”Amazon Simple Storage Service (S3): Devel- oper Guide,” Amazon Web Services (AWS), 2023. [Online]. Available: https://docs.aws.amazon.com/s3/.

[3] Apache Software Foundation, ”Kafka Streams: Stream Processing with Apache Kafka,” Apache Kafka Documentation, 2023. [Online]. Avail- able: https://kafka.apache.org/documentation/streams/.

[4] Apache Software Foundation, ”Apache Flink: Scalable Stream and Batch Data Processing,” Apache Flink Documentation, 2023. [Online]. Available: https://flink.apache.org/.

[5] Apache Software Foundation, ”Apache Storm: Distributed Real-time Computation System,”

Apache Storm Documentation, 2023. [Online]. Available: https://storm.apache.org/.

[6] Apache Software Foundation, ”Spark Streaming: Scalable and Fault- Tolerant Stream Processing,” Apache Spark Documentation, 2023. [On- line]. Available: https://spark.apache.org/streaming/.

[7] M. Hapner, R. Burridge, and G. Sharma, ”Java Message Service (JMS): A Distributed Messaging Standard,” Proceedings of the JavaOne Con- ference, San Francisco, CA, USA, 2002.

[8] Google Cloud, ”Pub/Sub: A Scalable Messaging Middleware for Event- Driven Architectures,” Google Cloud Whitepaper, 2023. [Online]. Avail- able: https://cloud.google.com/pubsub/docs/.

[9] Amazon Web Services, ”Amazon Kinesis Streams: Real-Time Data Streaming,” Amazon Web Services (AWS) Documentation, 2023. [On- line]. Available: https://aws.amazon.com/kinesis/streams/.

[10] Twitter, ”Distributed Log: Building and Maintaining a High-Throughput Messaging Platform at Twitter,” Twitter Engineering Whitepaper, 2016. [Online]. Available: https://engineering.twitter.com/.

[11] S. Chintalapudi, S. N. Chockalingam, and N. S. V. S. Srinivas, ”Databus: A Distributed Change Data Capture System for Large-Scale Data Replication,” Proceedings of the 2014 ACM International Conference on Management of Data (SIGMOD), Snowbird, UT, USA, 2014, pp. 1201-1212.

[12] A. Shriram, D. K. Ghosh, and T. S. Choi, ”Wormhole: Real-Time Event Streaming for Facebook’s Infrastructure,” Proceedings of the 2016 ACM International Conference on Distributed Systems (ICDCS), Nanjing, China, 2016, pp. 250-261.