TuijinJishu/Journal of Propulsion Technology
ISSN: 1001-4055
Vol. 46 No. 3 (2025)

Thulawa: Key-Aware Intra-Partition Parallelism extending Kafka Streams

Keshan Pathirana[1], Teshan Jayakody[2], Bihesha Dilshan[3], Manula Gunatilleke[4], Nuwan Kodagoda[5] and Samadhi Rathnayake[6]

[1,2,3,4,5,6]*Department of Computer Science and Software Engineering Sri Lanka Institute of Information Technology Malabe, Sri Lanka*

Abstract: -In the present, Event Stream Processing (ESP) is a critical process for real-time data analytics, which enables organizations to process and act on continuous streams of events. However, existing frameworks like Apache Kafka Streams face limitations in efficiently parallelizing event processing, particularly parallelizing beyond individual partitions, which can create performance bottlenecks in high-throughput scenarios. To overcome these limitations in Kafka Streams, our research introduces design enhancements through a parallel processing design that significantly enhances stream processing performance. Our approach enhances processing throughput and event scheduling by introducing parallel event processing and fair scheduling of events for processing, ensuring improved throughput in high-performance environments. To overcome Kafka Streams' processing limitations, we designed an architecture where scheduling, event submission, and execution are handled in parallel. A dedicated scheduling mechanism handles fair scheduling events for processing independently, while also event submission and execution occur parallelly, ensuring optimal resource utilization and improved throughput. These enhancements were implemented within an extended Kafka Streams library, Kafka-Thulawa. The findings show substantial improvement in processing throughput when compared to Kafka Streams. This study advances real-time stream processing by integrating parallel event execution within partitions, fair event scheduling, and improved design architecture, addressing key limitations in existing Kafka Streams.

Keywords*: Event Stream Processing, Apache Kafka Streams, Parallel Processing, Intra-Partition Parallelism, Key-Based Parallelism.*

1. Introduction

1.1. Background

ESP is a vital paradigm for real-time data analytics, which has become essential for organizations that need to process continuous streams of events efficiently [1]. The growth of Internet-of-Things (IoT) devices, cloud computing, and distributed systems and other business related use cases has increased the importance of ESP for handling and processing large amounts of streaming data in real-time, allowing organizations to respond dynamically to data patterns, anomalies, and real-world events [1] [2].

There are several ESP frameworks such as Apache Kafka Streams, Apache Flink, and Apache Pulsar and several others which have been developed to efficiently process real time data [3] with distributed and fault tolerant architectures enabling high throughput and low latency event processing [3].

Apache Kafka is an open-source distributed event streaming platform which can be used to process large amounts of real-time data. Kafka is based on the publish/subscribe model. Producers can send events to topics, and consumers can subscribe to those topics. Kafka is fault tolerant and highly available as it uses replication to distribute each partition across multiple brokers. The Kafka Streams library is a lightweight library for processing event data inside Kafka so developers can process event data without relying on external stream processing frameworks. It follows a partition-based processing model, where each partition is assigned to a single stream thread, ensuring scalability by enabling multiple threads to process partitions in parallel across distributed nodes. Furthermore, Kafka's distributed architecture provides a strong platform to build robust event-

driven applications for use cases like fraud detection, anomaly detection, real-time monitoring, and various other real-world use cases.

Kafka's partitioning mechanism is used to effectively distribute load between multiple consumers. When a producer sends an event, Kafka assigns it to a particular partition based on a key. If a key is provided, Kafka ensures that all events with the same key are delivered to the same partition, maintaining the order of events. If there is no key provided, Kafka distributes those events throughout partitions using a round-robin approach. Kafka Streams uses this partitioning model to provide parallel processing capabilities [4]. Kafka Streams instances have one 'StreamThread' by default, which can be increased by using num.stream.threads configuration if required. StreamThread runs sequentially for all processing tasks, such as scheduling, task submission, and execution which may cause bottlenecks.

Despite the advantages of ESP frameworks like Apache Kafka Streams, face challenges such as bottlenecks in stream processing, scalability limitations, and efficient resource utilization remain areas of concern [5]. This paper explores and provides design, architectural and algorithmic enhancements to address these challenges, specifically by exploring a key-based intra-partition parallel execution model and contributes to the advancement of real-time data analytics.

1.2. Problem Statement

Existing stream processing frameworks (e.g. Apache Kafka Streams, Apache Flink and Spark Structured Streaming) largely follow partition-based parallelism. In this design, processing units are mapped to partitions. This architecture is not necessarily a limitation, but one that can be further optimized through the processing nodes.

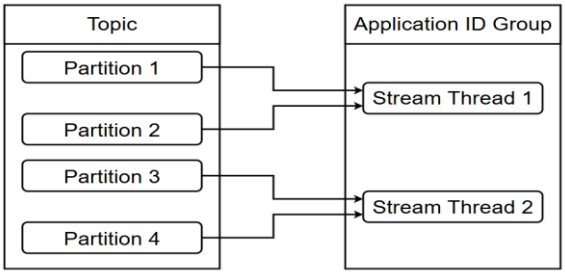
While alternative frameworks, such as Flink, support task-parallel execution, but by adding more overhead in scheduling and state management (especially in large-scale deployments), and vanilla Kafka Streams lacks built-in mechanisms for intra-partition parallelism and non-blocking execution. Therefore, Kafka Streams was chosen to implement our architecture of parallelism beyond the partitions.

Apache Kafka's capability to handle large-scale event data efficiently is primarily driven by its publish-subscribe messaging model and partitioning mechanism [6]. By sharing data into partitions, Kafka enables producers to write sequentially and consumers to read parallel across various brokers, leading to greatly improved throughput and scalability.

Kafka Streams, the Event Stream Processing library of Kafka uses this partition-based execution model where each partition is typically assigned to a single processing thread. This design is reasonably good because it guarantees order of processing within partitions and makes it easier to keep state clean. But with this approach the degree of parallelism is directly proportional and inherently restricted to the number of available partitions.

1.2.1. How Partitions are distribute in Kafka Streams

a) Instance 1



The Figure 1.1 shows an instance, of how partitions are distributed and assigned to Stream Thread or Kafka

Figure 1.1: Four Partition Distribution among two Kafka Streams application/Thread

Streams Processing application instance (A single Kafka Streams application instance has one Stream Thread, .

but it can be configured to have several by using the 'num.stream.threads' configuration in Kafka Streams). In the above case, since there are four partitions and two Stream Threads or application instances, each Stream Thread or application instance would get two partitions assigned each [6].

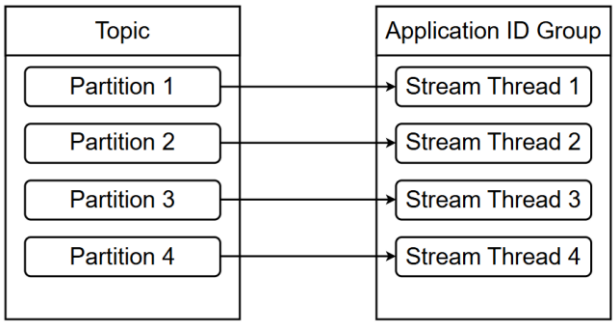


Figure 2.2: Four Partition Distribution among four Kafka Streams application/Thread

b) Instance 2

In Figure 1.2 instance, since there are four partitions and four Stream Threads or application instances, each Stream Thread or application instance will be assigned one partition each. Therefore, partitions will be equally distributed among Stream Threads or application instances.

c) Instance 3

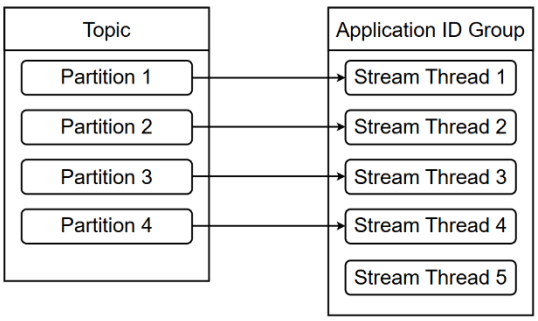


Figure 3.3: Four partitions with five or more Kafka Streams threads — surplus threads idle

In Figure1.3 instance, there are four partitions but now there are five Stream Threads or Application instances. In this scenario, only four partitions will do the processing, and the fifth partition will stay idle since it does not have a partition assigned to read from.

1.2.2. Limitations

a) Underutilized Parallelization

To increase processing throughput in Kafka Streams, users have two primary options:

1. Increase the number of partitions, thereby distributing the workload across more processing threads.
2. Scale Kafka Streams applications by either:
   - Increasing the number of application instances
   - Increasing the number of StreamThreads within an application instance.

While these approaches can improve performance, they introduce several challenges and drawbacks, and below are the two biggest drawbacks relating to our research:

- Under-utilized Parallelism - While Kafka Streams allows for multiple threads, tasks assigned to a single 'StreamThread' cannot leverage fine-grained parallel execution, limiting resource utilization. Meaning it will be ultimately limited to the number of available partitions and StreamThreads or Application instances created beyond the number of partitions would remain idle.
- Resource Utilization- Scaling up Streams applications or StreamThreads will increase competition for Central Processing Unit (CPU), memory, and network resources, leading to potential performance issues.

b) Blocking Execution Model (Specific to Kafka Streams)

In addition to the above-mentioned drawbacks, there is another important limitation in Kafka Streams which is the inherent blocking execution model.

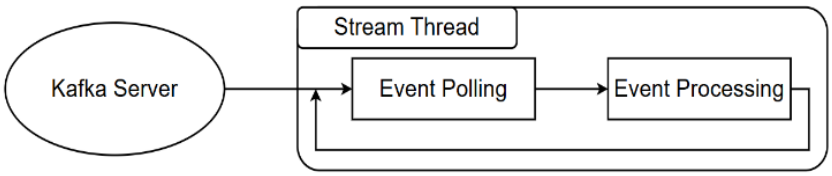


Figure 4.4: Blocking Execution Model of Kafka Streams

As depicted in Figure1.4, Kafka Streams' StreamThread handles event scheduling, task submission, and execution sequentially. This results in processing bottlenecks, particularly under high-throughput workloads. For example, in a scenario like making calls for an external API for processing of a certain event and the processing of other events will also be halted until the response arrives.

Therefore, this research aims to address these limitations by implementing an execution architecture upon the traditional partition-based model to improve parallelism across processing nodes. For practical implementation, we have chosen Kafka Streams as the stream processing library, and we will be extending Kafka Streams library with our own library called the Kafka-Thulawa. Our approach enhances parallel execution, resource utilization, and non-blocking processing while also maintaining the benefits of the existing partitioning mechanism.

1.3. Research Objectives

The primary objective of this research is to improve the execution architecture of partition-based stream processing frameworks or libraries by enhancing parallelism, resource utilization and non-blocking execution, while providing fair execution of arriving events on all processing nodes. While partition-based stream processing libraries like Apache Kafka Streams, offer scalability provided by partition-based parallelism, they remain inherently limited in terms of fine-grained intra-partition parallel execution. Therefore, it results in under-utilized processing capacity, inefficient resource allocation, and potential bottlenecks in high-throughput workloads.

To address the challenges discussed above, this research proposes an execution architecture that operates as an extension of the partition-based model to optimize the parallel processing in streaming applications. The key objectives of this research are outlined below:

- Design an Execution Architecture for Enhanced Parallelism- Develop a new and improved execution model that enables fair and efficient parallel execution of events within partitions, to increase overall performance and throughput when compared to the traditional approach.
- Implement a Fair and Efficient Asynchronous Processing Architecture – Efficiently process arriving events asynchronously, where an event with a certain key will not have to remain idle until an event with another key is still processing. Furthermore, preserving Key Ordering and events with the same key are not processed out of order.

- Integrate the Architecture within Kafka Streams for Practical Applicability – Integrate and extend the Kafka Streams Library with our architecture with our own library for practical applicability. Additionally, upgrade Kafka Streams Library with Thulawa Library to Fetch, Schedule and Process records in an asynchronous manner.
- Evaluate Throughput and Gains over Existing Stream Processing Models – Test the implementation of the proposed architecture against use cases and observe and validate the impact of the architecture.

Through these objectives, this research aims to contribute a scalable, efficient, and practical approach to stream processing, addressing the inherent limitations of partition-based architectures while maintaining their strengths.

2. Literature Review

The field of real-time data analytics depends heavily on stream processing which enables which enables continuous ingestion and processing of events. Frameworks and libraries such as Apache Kafka Streams and Apache Flink and Apache Spark Streaming have appeared to satisfy requirements of high-speed low-latency data processing. This literature review explores existing literature to identify current capabilities, limitations, and areas for improvement in stream processing architectures.

Apache Flink, Spark Streaming, and Kafka Streams, are comparable, but they are built with different architectures to suit different use cases. Apache Kafka Streams is lightweight by natures and can be seamlessly integrated with Apache Kafka and facilitates real-time analytics without necessitating a separate processing cluster [7]. Furthermore, different stream processing libraries have different strengths and weaknesses due to the architecture being different. Apache Flink excels in low-latency, real-time processing, Spark Streaming for heavy workloads and Kafka Streams for Kafka Integrations and event-driven applications that prioritize simplicity [7].

Most stream processing architectures, frameworks like Apache Flink and Apache Kafka are often used together to leverage their complementary strengths. Kafka serves as a distributed event streaming platform, providing durable and scalable storage for real-time data streams. Apache Flink and Kafka Streams, on the other hand, may work as either sink, source or processing nodes, enabling complex event processing and analytics. By using and combining these technologies, organizations can build efficient, real-time data pipelines and these types integration facilitates scalable, fault-tolerant, and high-throughput stream processing solutions [8]. Furthermore, not only on Kafka and Flink are together, but other technologies like Spark Streaming are also leveraged in the modern Stream processing environment [9].

Apache Kafka is a high-throughput distributed messaging system which is able to process millions of records per second. As a messaging system Apache Kafka is highly scalable, durable and reliable in high performance scenarios [10] of the modern world and is often times the go to choice in the modern streaming environment.

Furthermore, in these types of high velocity environments, ensuring the correctness guarantees like handling out of order data while balancing the performance and cost is a big challenge, To achieve this different architectures use different mechanisms. For example Apache Flink uses the Watermark concept for temporal completeness in infinite data streams [11]. Watermarks serve as a tool for reasoning about the completeness of data, allowing Flink to handle out-of-order events by indicating when certain time windows have received all their expected data.

In contrast Kafka Streams the completeness and consistency are guaranteed through its persistent log architecture. Even under unexpected failures and out-of-order data scenarios, Kafka Streams ensures correctness by integrating stream processing with its persistent logging. It uses idempotent and transactional write protocols to guarantee exactly once semantics, ensuring that each record is processed once and only once, even in the event of unexpected failures.

It should be clear now that different stream processing frameworks and libraries have different architectures and are generally suitable for different use cases and handle concepts like completeness and correctness using different mechanisms. Furthermore, often in modern use cases they are integrated together to satisfy different

needs. Therefore, improving areas like the performance individually within a framework or library will in turn improve the overall performance of the concrete implementations used within the industry.

There are several studies conducted to balance the workload and improve performance and throughput in Stream Processing. Key-based approach is one approach which is used to achieve the above target. Several studies achieve the above target by effectively managing and distributing load based on the keys among partitions [12] [13] [14]. There has been less focus on parallelizing and distributing the workload within the partition and within the processing nodes. Therefore, this research goes into parallelizing and distributing processing within the processing nodes, thereby trying to increase the throughput in stream processing.

In conclusion, stream processing frameworks such as Apache Kafka Streams, Apache Flink, and Spark Streaming each offer unique advantages tailored to different real-time data processing needs. While existing research has explored key-based partitioning for workload distribution across partitions, there is limited focus on optimizing parallel execution within partitions. This research addresses that gap by introducing intra-partition parallelism, aiming to enhance throughput and efficiency in stream processing architectures.

3. Methodology

During the initial phase of this research, we thoroughly investigated existing stream processing architectures, with a particular focus on partition-based execution models implemented in Libraries or Frameworks such as Apache Kafka Streams, Apache Flink, and Spark Structured Streaming. These Libraries or Frameworks use most notably partition-based parallelism with each partition assigned to a particular processing thread. This design architecture effectively helps distribute workload across multiple nodes when storing events and for processing applications to read events in a scalable manner. What we mainly observed was that parallelism can be extended beyond the partitions within those processing instances. Additionally, we analyzed alternative architectures that go beyond partition-based execution, such as task-parallel models and operator-based architectures of Flink and Spark. From this comparative study, we identified key limitations and areas where improvements could be made.

Upon analyzing several architectures, we found one key area of improvement: in Kafka Streams and similar partition-based frameworks, parallelism is pre-defined to the number of partitions. This creates inefficiency and can be optimized further, particularly in cases where a partition handles high-velocity event streams that could benefit from further parallel processing within a single processing node. Therefore, to address this, we propose an extension of parallelism beyond partition-based execution to the use of key-based execution in a processing instance handling a particular partition. Thereby, increasing the parallelism within the partition and the overall performance and the throughput of the processing instance.

Instead of designing a stream processing framework or library from the ground up, we decided to leverage an existing open-source framework or a library. The rationale behind this decision was to avoid reinventing the wheel while ensuring that our proposed architecture could be integrated into real-world streaming systems and to test it against the already existing systems. After evaluating several frameworks, we settled on Apache Kafka Streams as the library for our implementation due to the following reasons:

- Kafka Streams lacks an inherent key-based parallelism model – Kafka Streams inherently lacks the key-based execution model that we propose. Therefore, we could implement our solution as an extension of it and verify.
- Lightweight Client-Side library – Kafka Streams is a lightweight library. Furthermore, Kafka Streams provides a Processor API for custom processing without requiring a separate distributed processing framework.
- Part of the Kafka Ecosystem – Kafka Streams is a Kafka project which is well supported by an active open-source community and is used by many organizations around the world.

Once Kafka Streams was chosen as the library, we will be implementing our solution on, we performed a deep dive into its architecture, analyzing its execution model in detail. Apart from its partition-based parallelism limitation, we identified some other areas that we can optimize as well:

- Sequential Execution Model – Kafka Streams does fetch events and processing events in a sequential manner within a given stream thread. Therefore, this design could be further optimized.
- Lack of Asynchronous processing – Kafka the traditional execution model requires blocking operations for external calls (e.g., database queries, network requests), leading to inefficient processing which can be further optimized.

This analysis led to the expansion of our attention beyond only key-based intra-partition parallelism to the improvement of execution efficiency via non-blocking processing mechanisms.

3.1. High-Level Architecture

The idea of overall architecture is simple, the main optimization lies in the architecture that allows parallel processing within a partition, based on event keys.

As shown in Figure3.1, the architecture's event processing flow is planned to optimize parallelism while

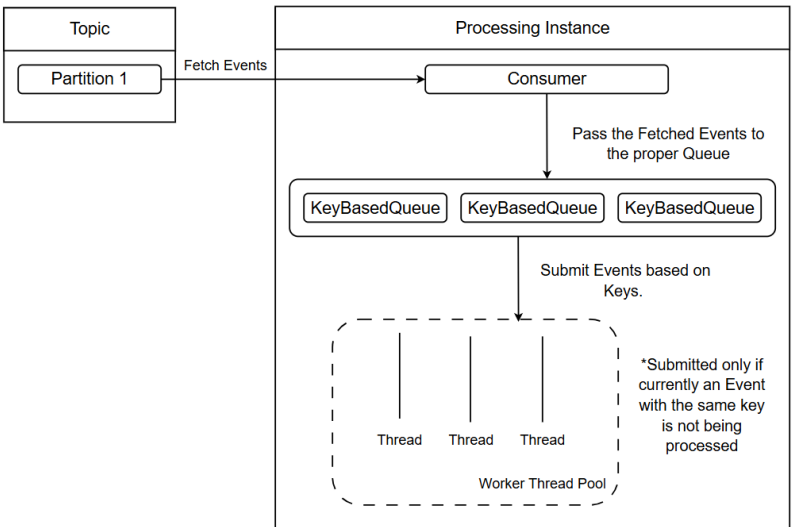


Figure 3.1: High Level Architecture of the proposed library

maintaining key ordering guarantee. Events are to be first ingested, categorized, and stored according to the key that corresponds to them. When an event arrives, the event must be first assigned to a Key-Based Queue. Events with a particular key are placed in a queue for that, ensuring that they maintain their original order relative to that key. If an event arrives without a key, it is to be placed in a common queue, which handles unkeyed events separately. This method ensures that key-based events can be processed in a structured manner and allows them to store the events categorically based on the keys and makes lookups based on the key easier.

Next an event is to be selected based on a certain scheduling mechanism and in this case prioritized by the arrival time and is to be submitted for a thread pool for processing. In this case it is important not to submit an event, if another event with the same key is currently being processed to make sure the key ordering guarantees are met. This ensures that the events with the same key are processed strictly based on the order of the arrival time. If an event with the same key is still being handled by a thread, the other events relating to that particular key will remain in respective queue until prior one completes. This controlled submission mechanism maximizes parallelism while preventing out-of-order processing.

Furthermore, this can be further improved by adding a Micro-Batching component.

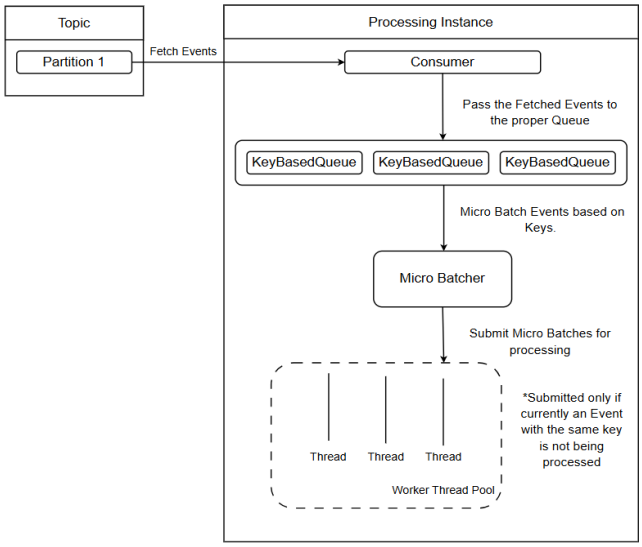


Figure 3.2: High Level Architecture with Micro Batcher

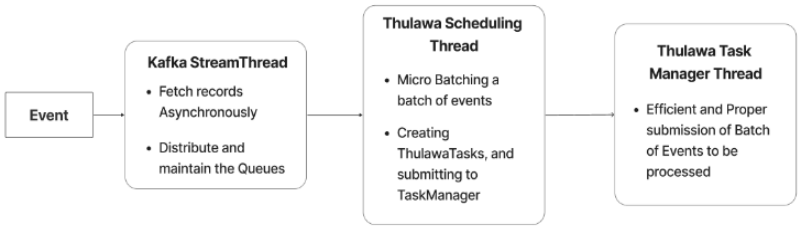


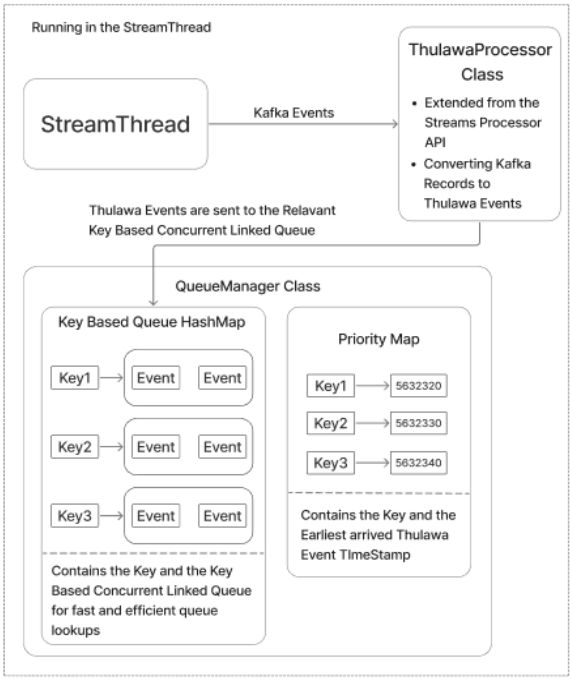
Figure 3.3: Asynchronous Event Processing flow in Thulawa Kafka Library

Including micro-batching can improve the processing efficiency and throughput compared to the earlier design in Figure 3.1 With the Micro Batcher in place, multiple events with the same key can be grouped together and processed as a single batch, reducing the number of submissions and lowering synchronization contention. This not only improves CPU utilization but also reduces per-event processing latency.

3.2. Implementation with Kafka Streams

During the implementation phase there were several challenges to overcome. The first of which was how to integrate the above solution within Kafka Streams. Kafka Streams provides an Interface called the Processor API which can be implemented and extended to suit processing use cases. Furthermore, the KafkaStreams class was extended to pass configurations to the applications and to set up the processing environment, for example exposing application related metrics like the application processing throughput.The next step in the implementation phase was to extend Kafka Streams in a way in which the polling of the events and the processing of the events are done asynchronously. To achieve this, a multithreaded approach was used.

The above diagram shows the multithreaded approach used to separate the processing from the fetching layer. In the next section, for each thread described in Figure 3.3, the purpose and the inner implementation working will be introduced.



3.3. StreamThread

In Thulawa implementation, the main task of the Kafka Stream Thread is to manage the flow of events to the relevant Key-Based queue. Therefore, in this implementation the Stream Thread is released off processing duties and other tasks and solely focused on distributing the events and maintaining the priority map.

The main components are:

1. Key-Based Queues – The events will be stored in separate First in First out (FIFO) queues according to their key. The events that arrive without a key would be stored in a common queue.
2. Priority Map – A map data structure that contains the key of a particular queue as the key of the map, and the value contains the timestamp of the event that arrived earliest to that particular queue.

At its core, the Stream Thread receives and passes them to the ThulawaProcessor class which was created by extending the Processor API of Kafka Streams. The ThulawaProcessor class converts the Kafka Events to Thulawa Events (A wrapper around the Kafka Events and additionally includes the processing logic as a runnable process). After converting to a Thulawa Event, the event is passed to the appropriate Key-Based concurrent linked queue managed by the QueueManager class.

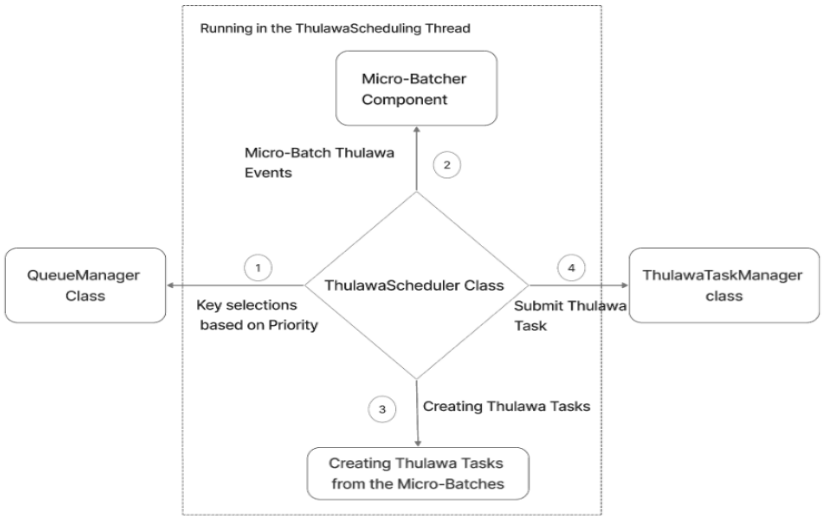
The QueueManager class maintains two essential data structures, a HashMap linking each key to its dedicated concurrent linked queue (ensuring events with the same key are processed in order), and a second HashMap (Priority Map) tracking the earliest event timestamp for each key.

The Priority Map is used to fairly and efficiently manage scheduling and prioritization. The earliest timestamp of an event in each Key-Based queue is maintained through this map, this ensures that the events are processed

Figure 3.4: StreamThread component design

fairly based on their arrival order, preventing starvation, where some keys might accumulate a higher load of events while some other keys would not accumulate as much resulting events being unprocessed. Since the oldest timestamp for each key is updated dynamically, the system will be able to make scheduling decisions accurately, so that no key is indefinitely delayed while optimizing the throughput.

3.4. Thulawa Scheduling Thread



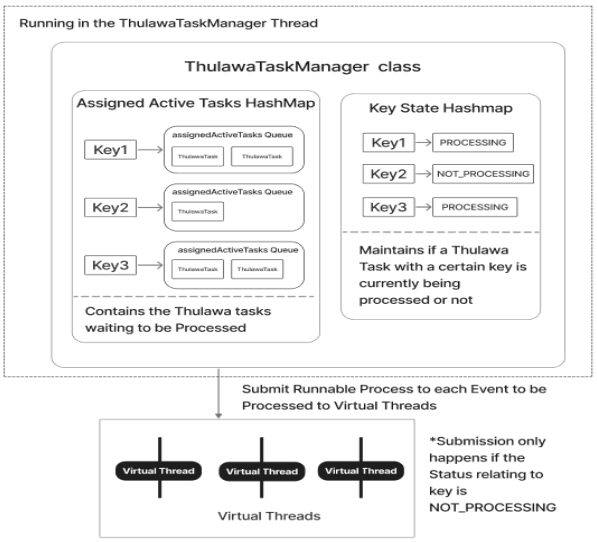
The main components are:

Figure 3.5: Thulawa Scheduling Thread component design

1. Micro-Batcher – A micro-batching component that creates a batch of events of a certain key rather than processing events individually.
2. Thulawa Scheduler – The Thulawa Scheduler selects the key with the highest priority for batching and submits tasks for Assigned Active Tasks Queues in the Thulawa Task Manager thread in Figure 3.6.

Before processing, the Micro-Batcher batches similar events of the same key into small batches. This micro-batching step allows multiple events of the same key to be processed in a single execution cycle within a single thread, increasing CPU efficiency and reducing the overhead of repeated context switching. Therefore, avoiding computational overhead and overall reduced system throughput.

The Thulawa Scheduler orchestrates efficient event processing, by operating within its own dedicated scheduling thread. It begins by identifying the key with the earliest timestamp from the QueueManager class. It .



then invokes the Micro-Batcher to fetch a batch of events associated with that key. These batched events are converted into Thulawa tasks, which are subsequently submitted to the ThulawaTaskManager class. By batching tasks per key, the scheduler improves CPU efficiency and also most importantly ensures ordered, non-overlapping processing of key-based workloads.

3.5. Thulawa Task Manager Thread

The main components are:

1. Assigned Active Tasks Queues – The batch of events batched by the Micro-Batching component to be stored in FIFO queues.
2. Key State Map - A map data structure containing the key of the batched records as the key and the value contains if any batch of events from that key is currently being processed or not.
3. Executor Thread Pool – The pool of threads where the batch of events will be submitted for processing.

Following batching by the Scheduling Thread, the batch of events are stored in Key-Based Batched queues called the Assigned Active Tasks queues. By doing so, when the system accepts a batch for processing, it can easily and efficiently retrieve the batch of events based on the key before submission. This type of queue acts as a buffer between event arrival and execution, allowing the system effectively to store and look up event batches before they are submitted for processing in threads.

The Key State map maintains the key and whether a batch of events with that key is being currently processed or not. Maintaining the Key State Map is vital to prevent several threads from attempting to process events of the same key in parallel, which could lead to data inconsistencies. Furthermore, this mechanism provided strict per-key processing guarantees, by ensuring that only one batch with a given key is being processed at a given time.

The actual processing happens inside an Executor Thread Pool, where batches of events are submitted to be processed. Furthermore, only a batch of events with a key currently not in the processing state should be

Figure 3.6: Thulawa TaskManager thread component design

submitted to the Executor Thread Pool for processing. Specifically in Java, we can use Virtual Threads instead of Platform Threads to do the processing since they follow a Thread-per-Task model and are lightweight, therefore less costly and are also efficiently managed by the JVM.

This structured flow ensures that parallelism is not limited by the number of partitions but rather by the number of unique keys in the system. This significantly improves system efficiency in high-load scenarios, where a large number of keys exist. Therefore, this architecture is well-suited for high-throughput, real-time processing environments.

4. Results and Discussion

The performance of the proposed Kafka Thulawa, the key-based intra-partition parallelism architecture was evaluated under a series of simulated load conditions designed to closely reflect real-world streaming workloads. To make sure the results were both rigorous and reproducible, the experiments were conducted using both Confluent Cloud and locally hosted Kafka servers. In the Confluent Cloud setup, we used the Confluent DataGen connector to generate a stream of synthetic events. Additionally, in local environments, we experimented with custom producers to simulate different simulation scenarios. Through the next part of the paper we will in detail go through testing scenario which was conducted using the Confluent Cloud environment and the Confluent DataGen connector along with the results associated with it.

4.1. Confluent Cloud Based Simulation

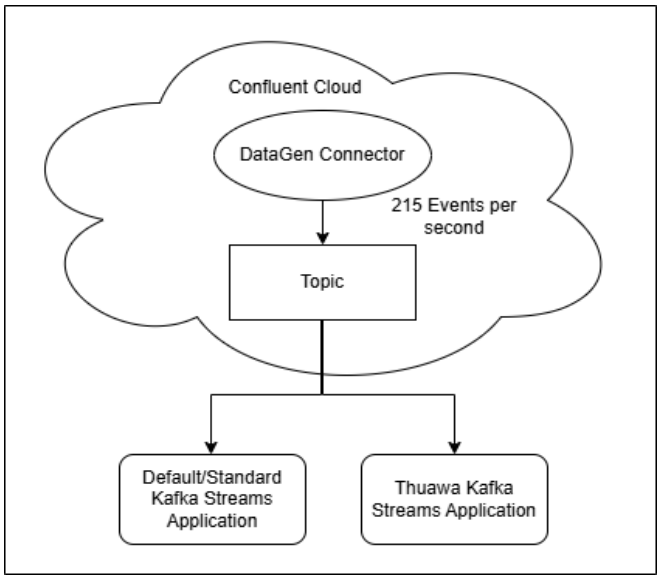


Figure 4.1: Confluent Cloud Simulation Workflow for Throughput Comparison

The central goal of these tests was to measure throughput improvements provided by the Thulawa library extension with the proposed architecture over the standard Kafka Streams processing model when tasked with external API enrichment operations and in this scenario, GeoIP lookups.

The stream processing logic in both the Thulawa Streams application and Default or Standard Stream application remained the same. The processing logic was designed to consume pageview events generated by the Confluent DataGen connector through the Confluent Cloud to the Stream Processing applications. Each event contained an IP address field as the key of the event and then these events were then enriched with geographical metadata such as city, country, and time zone by querying an externally hosted GeoIP enrichment service. The high-level test design is depicted in Figure 4.1.

To simulate realistic data ingestion scenarios, the Confluent DataGen connector was configured to generate synthetic pageview events using the Pageview Schema provided by the Confluent DataGen connector. During peak load simulations, the DataGen connector was tuned to emit approximately 215 events per second. The tests were run over a 40-minute period to allow JVM warm-up effects to stabilize and to capture steady-state behavior.

The above-described scenario was conducted in three stages. This setup was useful in understanding the impact of external API latency on throughput between the two applications, we conducted the tests using three distinct deployment setups of the GeoIP enrichment service:

1. Local Deployment: The enrichment service was hosted locally on the same machine as the stream processor.
   - Observed average latency: ~5 ms
2. Same-Network Deployment: The enrichment service was hosted on a different device within the same local network.
   - Observed average latency: ~20 ms
3. External Network Deployment: The enrichment service was hosted on a remote device in a different network.
   - Observed average latency: ~250 ms

Table 1: List of the Testing Scenarios

Test Scenario	Enrichment Service Location	Observed Latency (ms)
1	Localhost (same machine)	~5
2	Same network (different device)	~20
3	External network (remote Server)	~250

4.2. Results

The Result of the comparative tests showed that the Thulawa Streams instance was able to consistently outperform the Traditional Kafka Streams application instance, under identical conditions and was able to significantly outperform the Traditional Kafka Streams instance when the request response latency is high.

4.2.1. Test Scenario 1

In this setup, the GeoIP enrichment service was hosted locally on the same machine as the stream processing application. This minimized network latency, simulating a low-latency enrichment environment.

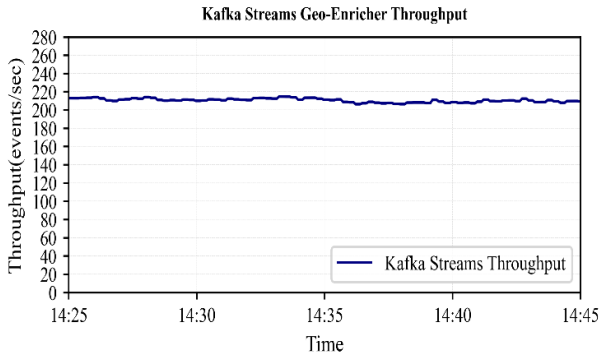
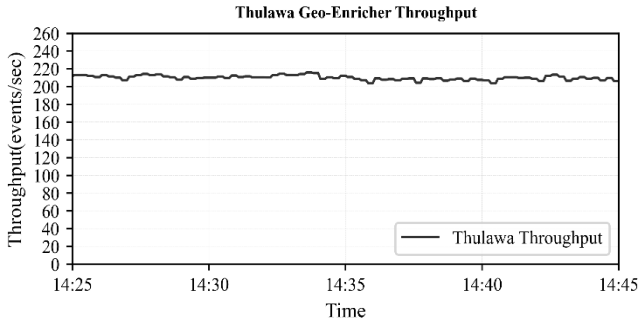


Figure 4.2: Test results of Traditional Kafka Streams application, when the enrichment service was hosted in the same network on a different device.

As Visualized in Figure 4.2 the traditional Kafka Streams application instance was able to maintain a steady



processing throughput of 210 events per second on average.

Figure4.3 visualizes the throughput results from the Thulawa Kafka Streams application instance, and it was able to maintain a steady processing throughput of 210 events per second on average. Therefore, as evident from the results of test scenario 1, when the processing latency is minimal, the throughput when the two application instances are compared is nearly identical.

4.2.2. Test Scenario 2

In this setup, the GeoIP enrichment service was hosted on a different device within the same local network. This introduced moderate network latency, simulating microservices like deployment in an on-premises or LAN

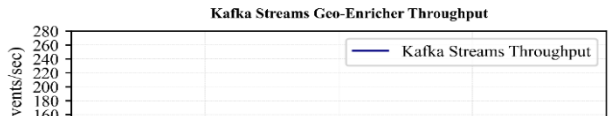


Figure 4.3: Test results of Thulawa Kafka Streams application, when the enrichment service was locally hosted.

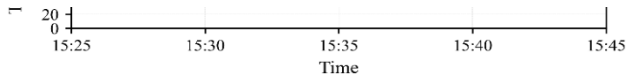


Figure 4.4: Test results of Traditional Kafka Streams application, when the enrichment service was locally hosted.

environment.

Figure4.4 visualizes the throughput results from the results of the traditional Kafka Streams application instance, when the request response latency was roughly 20ms. It was able to maintain a steady processing throughput of 39 events per second on average, and it is evident that the processing throughput has decreased considerably.

Figure4.5 visualizes the throughput results from the Thulawa Kafka Streams application instance, when the request response latency was roughly 20ms. During this test scenario, it was able to maintain a steady processing throughput of 211 events per second on average. It is evident from the above results that when the processing latency increases, the application instance with Thulawa Kafka Streams application produces better throughput than the traditional Kafka Streams application.

4.2.3. Test Scenario 3

In this setup, the GeoIP enrichment service was hosted on an external device in a different network. This introduced high network latency, simulating real-world scenarios involving third-party APIs or cross-region

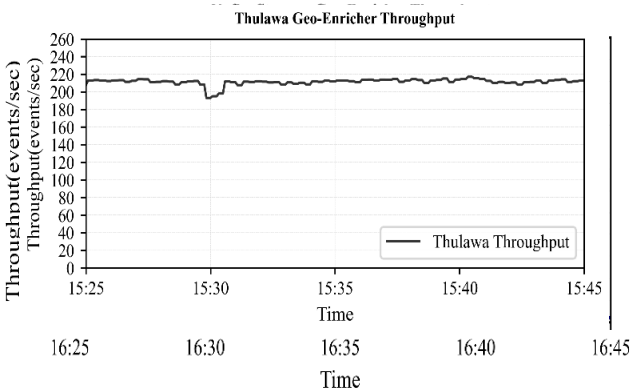


Figure 4 hen the enrichment service was hosted in the same network on a different device.

Figure 4.6: Test results of Traditional Kafka Streams application, when the enrichment service was hosted on an external server.

service calls.

Figure4.6 visualizes the throughput results from the traditional Kafka Streams application instance, when the request response latency was roughly 250ms, which reflects a real-world scenario. During this test scenario, it was able to maintain a steady processing throughput of only 4 events per second on average. Therefore, it is now evident that as thelatency increases, the traditional Kafka Streams application's processing throughput decreases drastically.

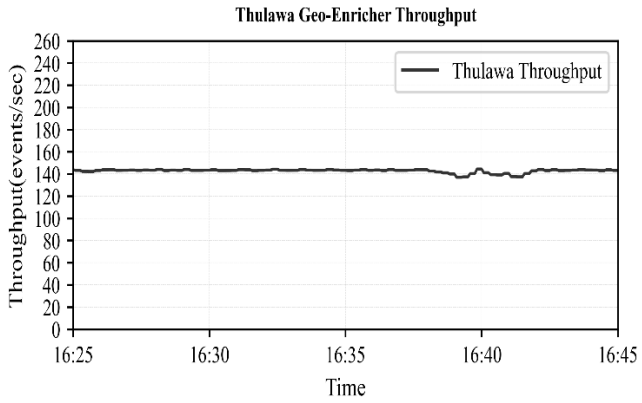


Figure 4.7: Test results of Thulawa Kafka Streams application, when the enrichment service was hosted on an external server.

Figure4.7 visualizes the throughput results from the Thulawa Kafka Streams application instance, when the request response latency was roughly 250ms. During this test scenario, it was able to maintain a steady processing throughput of 143 events per second on average. Even though the Thulawa Kafka Streams application instance also has decreased, it has been able to maintain a higher processing throughput compared to the traditional Kafka Streams application instance.

The following table summarizes the measured throughput of both the Traditional Kafka Streams and Thulawa Kafka Streams applications across all three test scenarios of the Confluent Cloud based Geo-IP enrichment simulation.

Table 2: Testing results summarized

Test Scenario	Observed Latency (ms)	Traditional Kafka Streams	Thulawa Kafka Streams
1	~5	210	210
2	~20	39	211
3	~250	4	143

This confirms that Thulawa's asynchronous, key-based parallelism is particularly effective in high-latency enrichment environments, making it a more robust solution for real-world systems where external API response times are not guaranteed to be minimal.

4.3. Analysis and Discussion

The performance-wise improvements displayed in the test scenarios indicate the efficiency and the effectiveness of Thulawa's architectural approach.

Some of the key factors have contributed to these significant improvements:

- Enhanced Parallelism - With the Thulawa Library, the parallel processing was enhanced to a point beyond the partition limitations of other event processing frameworks and was able to process a number of events concurrently within one single partition.
- Non-Blocking Execution - With the approach of asynchronous processing of the Thulawa library, the system was able to process new events without being blocked by the external API response delay in each event's processing. For context, the standard Kafka streams implementation was processing events within a partition sequentially which led to a block in execution of the events.
- Fair Scheduling - The priority-based scheduling mechanism ensures that all events are processed according to their arrival time, which prevented the starvation of any particular key while improving the processing speed.

Thulawa library's enhancements compared to the standard Kafka Streams' performance in the simulation scenarios were significant observations which can be useful in the industries of financial services, telecommunications, and IoT applications which handles time-sensitive data and requires real-time event processing abilities.

These observed results indicate that this followed approach successfully addresses the key limitations identified in traditional partition-based stream processing frameworks, mainly the limitations on parallel processing and use cases requiring external enrichment and high throughput. By extending the execution model beyond partition limitations while maintaining essential key ordering guarantees, it has been shown that it is possible to enhance processing throughput significantly. These enhancements position Kafka Thulawa as a compelling extension to the standard Kafka Streams model, especially for high-throughput, low-latency-sensitive, and externally enriched data pipelines.

4.4. Limitations

The proposed Thulawa architecture addresses several fundamental limitations of Apache Kafka Streams, including its single-threaded execution model, intra-partition processing bottlenecks, and blocking behavior. But the proposed Thulawa architecture also do possess certain limitations.

- Event Ingestion Pressure - The Kafka StreamThread in our implementation is designed to continuously poll and queue incoming events into Key-Based Queues. Although this ensures the processing threads are well-fed and maximizes throughput under normal conditions, it can lead to excessive queuing and memory pressure when processing throughput falls behind ingestion rate.
- Micro-Batching Strategy - Thulawa introduces a place for micro-batching at the key level with the broader idea to reduce scheduling overhead, improve CPU utilization, and enable better coordination of event processing tasks. However, the current batching mechanism operates using fixed-size or static batch thresholds, which do not adapt to real-time workload characteristics.

4.5. Future Work

To address the limitations identified above and to further enhance the practicality and robustness of Thulawa and similar streaming architectures, we outline several promising directions for future work below.

- Adaptive fetch rate control - Adjusts polling frequency or fetching batch size based on current Key-Based Queue occupancy or processing lag of the application. This would enhance the stability of the application while preserving the benefits of the Thulawa architecture.
- Adaptive Micro-Batching – The use of Adaptive Micro-Batching algorithms that dynamically adapt based on the workload characteristics can be experimented further with Thulawa architecture. By evolving beyond fixed size batching, the improvements gained can be tested and verified if significant.
- Decouple State Store from Compute - Another direction for future enhancement is the decoupling of the state store from local compute, particularly by adopting a tiered or externalized state model. This

can be explored by implementing a key-aware state offloading mechanism, where the state associated with infrequently accessed keys is dynamically offloaded to an external database or distributed store.

5. Conclusion

The findings of this research provide valuable and highly tested insights into Event Stream Processing frameworks which are based primarily on partition-based parallelism, particularly Apache Kafka Streams. The key problems that were identified and addressed though this research were;

- Under-utilized parallelism – Tasks assigned to a single StreamThread cannot leverage fine-grained parallel execution
- Blocking Execution Model – Scheduling, Submission, and execution is handled sequentially.

By introducing Kafka Thulawa Library as an extension to Kafka Streams, a new approach was demonstrated that enables parallel processing beyond the partition limitations through a key-based execution process.

Kafka Thulawa's architecture addresses the key problems that were identified throughout this research. It consists of multi-layered processing model which includes key-based queues, priority mapping, micro-batching, and an efficient processing mechanism. This specific approach ensured that event records are processed with high throughput.

The key contributions of this research include:

1. Enhanced Parallelism – The ability to improve the ability of parallel processing beyond the partition limitations resulted in the significant improvement of the processing efficiency.
2. Optimized Scheduling – All events will receive processing attention according to the time of arrival of the event due to the scheduling mechanism which ensures optimized scheduling.
3. Non-blocking Execution – The system can handle events without being blocked by long-running operations as a result of asynchronous processing.
4. Practical Implementation – Working with a foundation of an existing framework rather than building from the start, allows systems that already use Kafka streams for event processing, seamlessly integrate Kafka Thulawa.

Kafka Thulawa was able to demonstrate notable throughput improvements as discussed in the Results and Discussion section, especially in high-throughput environments with different event keys. This approach will be helpful for applications that process real-time event streams that require both high throughput and guaranteed Key-Based event ordering.

This research advances the field of real-time data processing by addressing key performance bottlenecks in current stream processing architectures by providing a foundation for more efficient and scalable ESP systems.

Refrences

[1] P. M. El-Kafrawy and M. Bennawy, "Walk Through Event Stream Processing Architecture, Use Cases and Frameworks Survey," in Proc. 2022, pp. 45-60, 2022.

[2] Sabrine Khriji, Yassine Benbelgacem, Rym Chéour, D. El Houssaini, and OlfaKanoun, "Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks," The Journal of Supercomputing, vol. 78, no. 4, p. 3374–3401, 2022.

[3] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos, "A survey on the evolution of stream processing systems," VLDB Journal, vol. 33, p. 1–35, 2023.

[4] Kia Teymourian, Faisal Bukhari Hasan, and Adrian Paschke, "Fusion of Event Stream and Background Knowledge for Semantic-Enabled Complex Event Processing," in in Proceedings of the 2011 Workshop on Data-Driven Research in Event Processing, 2011.

[5] H. Jiang, Y. Li, Z. Wang, and X. Zhang, "Event stream denoising method based on spatio-temporal density and time sequence analysis," Sensors, vol. 24, no. 20, p. 6527, 2024.

[6] Theofanis P. Raptis and Andrea Passarella, "On Efficiently Partitioning a Topic in Apache Kafka," 2022
[7] Akbar Sharief Shaik, "Advancements in Real-Time Stream Processing: A Comparative Study of Apache Flink, Spark Streaming, and Kafka Streams," International Journal of Computer Engineering and Technology, vol. 15, no. 6, pp. 631-639, 2024.
[8] Srijan Saket, Vivek Chandela, Md. Danish Kalim, "Real-time Event Joining in Practice With Kafka and Flink,", 2024.
[9] Yuriy Drohobytskiy, Vitaly Brevus and Yuriy Skorenkyy, "Spark Structured Streaming: Customizing Kafka Stream Processing," in 2020 IEEE Third International Conference on Data Stream Mining & Processing, 2020.
[10] Zhenghe Wang, Wei Dai, Feng Wang and Hui Deng, "Kafka and Its Using in High-throughput and Reliable Message Distribution," in 2015 8th International Conference on Intelligent Networks and Intelligent Systems, 2015.
[11] Tyler Akidau, Edmon Begoli, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth Knowles, Daniel Mills and Dan Sotolongo, "Watermarks in stream processing systems : Semantics and comparative analysis of apache flink and google cloud dataflow," Proceedings of the VLDB Endowment, vol. 14, no. 12, pp. 3135-3147, 2021.
[12] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, Marco Serafini, "Partial Key Grouping: Load-Balanced Partitioning of Distributed Streams,", 2015.
[13] Gang Liu, Zeting Wang, Amelie Chi Zhou and Rui Mao, "Adaptive key partitioning in distributed stream processing," CCF Transactions on High Performance Computing, p. 14, 2024.
[14] Junhua Fang, Rong Zhang, Tom Z.J.Fu, Zhenjie Zhang, Aoying Zhou, Junhua Zhu, "Parallel Stream Processing Against Workload Skewness and Variance,", 2016.