2023-2024

INFO-H415 - Advanced Databases

Streaming System Benchmark with Flink and Storm

Group:

Jintao Ma

Linhan Wang

Iyoha Peace Osamuyi

Hieu Nguyen Minh

Professor

Esteban Zimányi

December 2023

Contents

Listings

List of Figures

Abstract

This project encompasses a comprehensive evaluation of streaming databases, focusing on Apache Flink and Apache Storm, and their comparative performance against a traditional relational database, specifically PostgreSQL. The research is structured into seven chapters, each addressing critical aspects of streaming data processing and benchmarking.

Chapter 1 introduces the concept of streaming databases, setting the stage for an in-depth exploration of two prominent tools in this domain: Apache Flink and Apache Storm. This foundational chapter establishes the technological context and relevance of these tools in modern data processing.

Chapter 2 delves into the core of the project - the implementation of five benchmark applications. It also defines four essential performance metrics that form the basis of our evaluation criteria: throughput, latency, CPU utilization, and memory utilization.

Chapter 3 describes the methodology behind our benchmarking process. It provides a detailed account of the experimental setup, execution procedures, and the specific configurations used in testing both Flink and Storm.

Chapter 4 utilizes a variety of charts and data visualizations to present a comparative analysis of Flink and Storm. This chapter focuses on interpreting the differences in performance metrics, offering insights into the strengths and limitations of each tool.

Chapter 5 showcases the implementation of a specific use case called Product dataset, aimed at demonstrating the superior performance of Flink over Storm in a controlled experimental environment.

Chapter 6 extends the comparative study to include PostgreSQL, a traditional relational database.

Chapter 7 culminates the project with a summary of our findings and conclusions.

1 Introduction

1.1 Overview

Although PostgreSQL has been widely used, it still lacks the ability to deal with continuous data. What PostgreSQL can do is to have batch processing. Batch processing involves batches of data that have already been stored over a period of time, and is run on regularly scheduled times or on an as-needed basis. However, batch processing doesn't allow end user interaction. As a result, many use cases are not possible only with traditional relational database.

- **Real-time fraud and anomaly detection**[RB16]. With batch processing, credit card providers performed their time-consuming fraud detection processes in post-transaction. However, with stream processing, credit card providers are able to run thorough algorithms to recognize and block fraudulent charges and trigger alerts for anomalous charges.
- **Internet of Things(IoT) edge analytics**[Yan17]. With batch processing, companies in manufacturing such as transportation, oil and gas detect anomaly and fix problems after a period of time. However, with streaming processing, manufacturer may recognize that a production line is turning out too many anomalies as it is occurring. By stopping the production line and fixing the problems immediately, companies can avoid hugh waste.

- **Real-time personalization, marketing, and advertising**[Muk+10]. With batch processing, the recommendation for users are not real-time, which means possible loss in revenue. However, with streaming processing, companies can have discount for products in cart, make a recommendation to movies just seen, or an advertisement for a product similar to the one you just viewed.

- **Fault Tolerance and Scalability.** Applications that cannot afford to lose data or suffer downtimes due to system failures require a high degree of fault tolerance. Additionally, the ability to scale the system as the data volume or processing needs grow is crucial for many large-scale applications.

1.2 Streaming Process

Stream processing is a computing paradigm that involves the continuous processing of data as it is generated or ingested, rather than processing it in batch mode [KDA19]. In stream processing, data is treated as a continuous flow or stream, and computations are performed on the data incrementally as it arrives. This paradigm is particularly well-suited for scenarios where low-latency, real-time insights, and immediate response to changing data are essential.

There are many stream processing systems such as Apache Apex, Aurora, S4, Storm, Samza, Flink, Spark Streaming, IBM InfoSphere Streams, and Amazon Kinesis.[Che+03] However, some tools are not open-source. In this project, we choose 2 different open-source and public system Apache Storm and Apache Flink to compare their performances on different benchmarks.

1.3 Apache Storm

Apache Storm is an open-source, distributed real-time stream processing system designed for handling large volumes of data swiftly and efficiently[Sto23]. It provides a robust platform for developing applications that demand low latency and high throughput in the processing of continuous data streams. Originally developed by Nathan Marz, Storm was later contributed to the Apache Software Foundation, where it has been maintained and enhanced.

1.3.1 Architecture of Apache Storm

The architecture of Apache Storm is characterized by its distributed and fault-tolerant design, leveraging a master-slave configuration for efficient cluster and application management[Clo23].

Nimbus: The Master Node

- Nimbus serves as the master node, or the controller node, in a Storm cluster.
- Its primary responsibilities include distributing code across worker nodes, assigning tasks to Supervisor nodes, and monitoring the cluster's overall health and status.
- Nimbus plays a pivotal role in managing the scheduling of new topologies, ensuring fault tolerance, and facilitating communication with Zookeeper.
- Upon submission of a new application to the cluster, Nimbus executes a scheduling algorithm to allocate the application across the available slave nodes.

Zookeeper: Cluster Coordination

- Zookeeper is integral for coordination and distributed synchronization within Storm clusters.
- It functions as a distributed configuration store and is crucial for maintaining the state and stability of the cluster.

Supervisor Nodes: The Executors

- Supervisor nodes are tasked with executing the assignments delegated to them by Nimbus.
- Each worker node in a Storm cluster operates a Supervisor daemon, also referred to as the slave node.
- These nodes are the operational backbone of the Storm cluster, directly handling the processing tasks.

The architecture of Storm is shown in Figure 1.1.

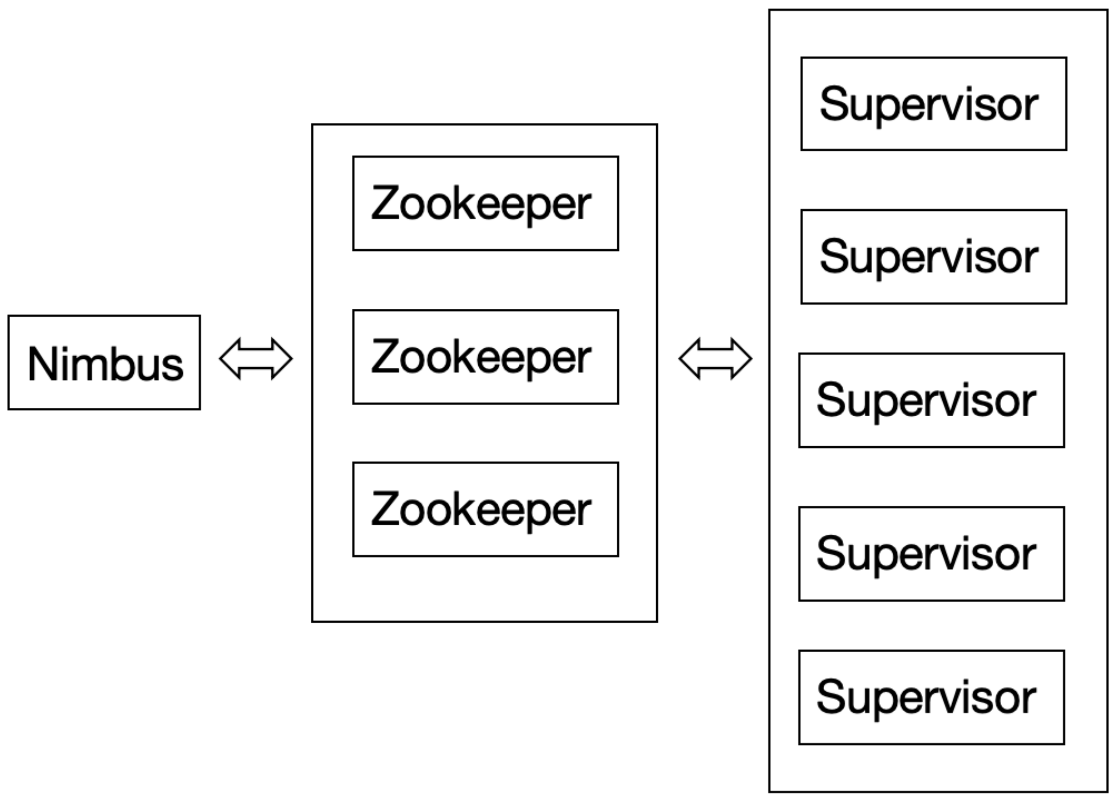


Figure 1.1: The Architecture of Storm

1.3.2 Advantage of using Apache Storm

Among the benefits of using Apache Storm for managing streaming data are:

- Scalability: Easily scales with the amount of data and the complexity of the processing logic.
- Provides robust fault tolerance with at-least-once processing guarantees.
- Capable of processing thousands of messages per second per node.
- Provides a simple programming model that is easy to understand and develop for.
- Supports multiple programming languages.
- True streaming framework with low latency and high throughput
- Support complex event processing and pattern matching over data streams

1.4 Apache Flink

Apache Flink is a dynamic, open-source stream processing framework designed for high-performance, scalable, and accurate real-time data processing applications[Fli23c]. Unlike many other streaming frameworks, Apache Flink is distinguished by its true streaming model, which processes data in a continuous flow rather than in batches or micro-batches. This approach allows for more efficient and timely data processing, making it particularly suitable for applications where low latency and high throughput are critical.

Developed originally by the company Data Artisans, Apache Flink is now managed and improved under the Apache License by the Apache Flink Community. Its design and architecture have made it a popular choice for a variety of real-time data processing tasks in numerous industries.

1.4.1 Architecture of Apache Flink

The architecture of Apache Flink is a key factor in its performance and scalability. Central to its architecture are the Task Managers and Job Managers, which play crucial roles in executing and managing Flink applications (jobs) [Kra20]. The architecture is shown in Figure 1.2.

JobManager: The Coordinator

- Each Flink cluster contains a JobManager, which functions as the central coordinating node.
- The JobManager is responsible for assigning tasks to the TaskManagers, managing the overall workload, and coordinating the execution of tasks across the cluster.
- It also handles job scheduling, recovery from failures, and other administrative tasks.

TaskManager: The Executor

- TaskManagers are responsible for executing the tasks of a Flink job.
- A unique feature of the TaskManager is its division into multiple task slots. This design allows a TaskManager to execute different tasks concurrently, thereby enhancing the efficiency and throughput of data processing.
- TaskManagers communicate with the JobManager to receive tasks and report on their execution status.

Flink's Streaming Model

- Flink's true streaming model is one of its most defining features. This model processes data as a continuous stream, enabling more responsive and real-time data processing solutions.
- It is particularly adept at handling complex event processing, stateful computations, and windowing operations, making it a versatile tool for a wide range of streaming applications.

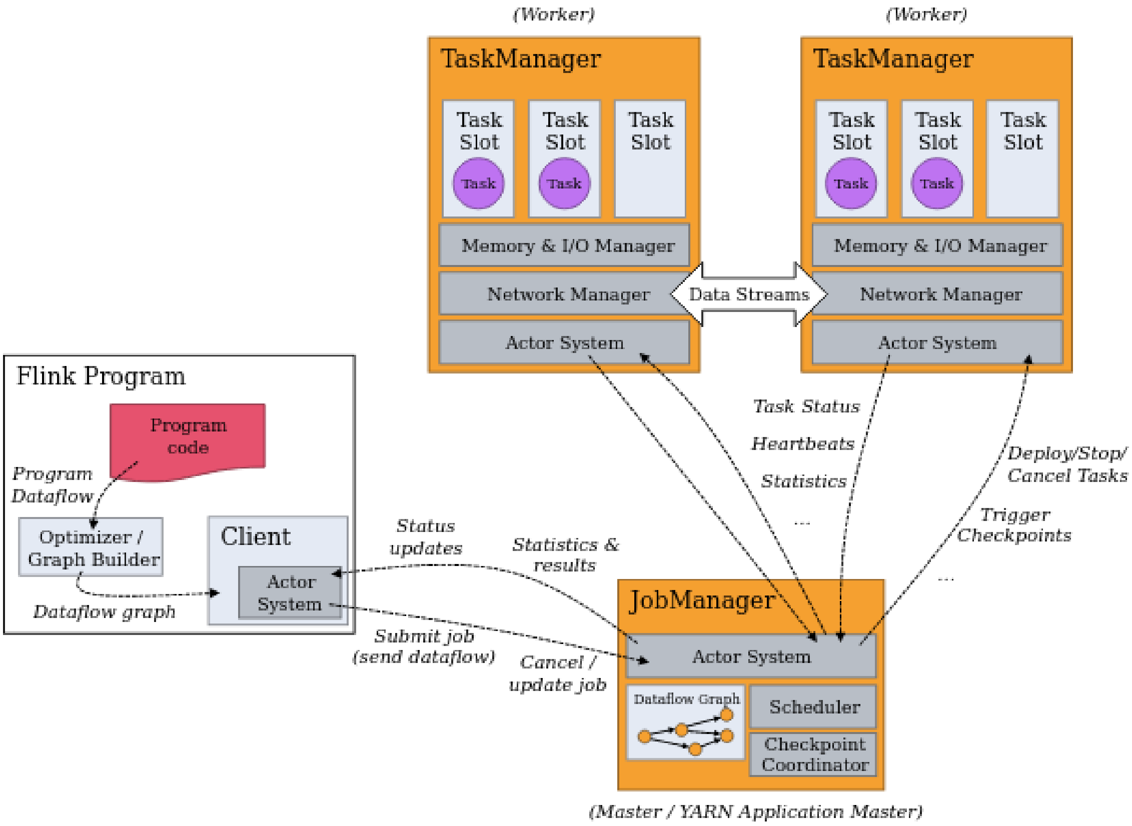


Figure 1.2: The Architecture of Flink

1.4.2 Flink's API

Flink offers two types of APIS, depending on the data source and whether batch or stream processing is being used. While DataStream API is used for streaming, DataSet API is utilized for batch processing[Ali21].

DataSet API: with the help of this API, we may receive data sets from data sources, publish via sinks to the required location [Fli23b]. Examples of transformation functions are Union, Distinct, Rebalance, Join, Filter, and Map.

DataStream API: Real-time data streaming applications such as filtering, updating, windowing, aggregating, and more employ here[Fli23d] . Data streams first originate from files, socket streams, and message queues. The results are retrieved using third-party programs or sunk to data files.

For batch and unified stream processing, table API and SQL are utilized. The user can easily write sophisticated SQL queries with the aid of this api. Table environments can be utilized to construct tables with dataset or datastream APIs. A user can easily select from a table once it has been created [Fli23e].

A library used for complex event processing is called FlinkCEP. This enables the identification of event patterns in an event stream [Fli23a].

Gelly: A graph analysis tool, it may be used to construct, alter, and convert graphs, among other things. For vertex or edge values, it offers a map transformation[Ali21].

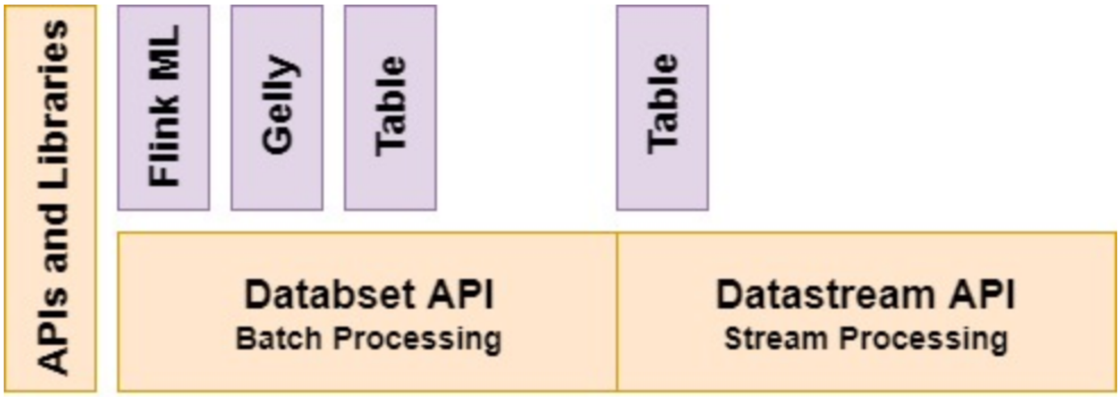


Figure 1.3: Flink API

1.4.3 Advantage of Using Apache Flink

- Native streaming with low latency and high throughput
- Rich set of operators and APIs for complex event processing
- Support for event time and out-of-order events
- Scalable and fault-tolerant state management
- Handles both batch and stream processing with a single framework and API
- Flink's custom memory management system reduces the overhead of garbage collection, which can be a bottleneck in JVM-based systems.
- Flink can handle iterative processing natively, which is useful for algorithms that require multiple passes over the same data, such as machine learning algorithms.

- Flink's management of application state is more advanced compared to many other streaming platforms. It supports complex stateful computations and provides various state backends.

1.5 Apache Flink and Apache Storm Comparison

The table[Glu23] shows a comparison between apache flink and apache storm:

Aspect	**Flink**	**Storm**
Type	Hybrid (batch and stream)	Stream-only
Distributed	Full (cluster deployment, HA, Fault Tolerant)	Yes
Stateful	Yes (RocksDB)	Yes (with Trident)
Table API	Yes	No
Supports handling late arrival	Yes	No
Learning curve	Moderate	Hard
Support for 3rd party systems	Multiple source and sink	Yes (Kafka, HDFS, Cassandra, etc.)
Complex event processing	Yes (native support)	No
Streaming window	Tumbling, Sliding, Session, Count	Time-based and count-based
Iterations	Supports iterative algorithms natively	No
SQL	Table, SQL API	No
Optimization	Auto (data flow graph and the available resources)	No native support
State Backend	Memory, file system, RocksDB or custom backends	Memory, file system, HBase or custom backends
Backpressure	Auto (adjusting the processing speed)	Manual (tuning the spout configuration parameters)

Aspect	**Flink**	**Storm**
Latency	Streaming: very low latency (milliseconds)	Tuple-by-tuple: very low latency (milliseconds)
Data model	True streaming with bounded and unbounded data sets	Tuple-based streaming
Processing engine	One unified engine for batch and stream processing. Uses a streaming dataflow model that allows for more optimization than Spark's DAG model.	Stream engine that processes each record individually as it arrives. Uses a topology model that consists of spouts (sources) and bolts (processors).
Delivery Guarantees	Supports exactly-once processing semantics by using checkpoints and state snapshots. Also supports at-least-once and at-most-once semantics.	Supports at-least-once processing semantics by using acknowledgments and retries. Can achieve exactly-once semantics by using Trident API, which provides transactions and state management.
Fault Tolerance	Provides high availability and fast recovery from failures by using checkpoints and state snapshots stored in external storage systems. Supports local recovery for partial failures.	Provides fault tolerance by using acknowledgments and retries to ensure reliable message delivery. Also uses ZooKeeper to store the state of the topology and the spouts' offsets.
Performance	Achieves high performance and low latency by using in-memory processing, pipelined execution, incremental checkpoints, network buffers, and operator chaining. Also supports batch and iterative processing modes for higher throughput.	Achieves high performance and low latency by using in-memory processing, parallel execution, local state management, and backpressure control. However, Storm does not support batch or iterative processing modes natively.

In this project, we will adopt different benchmarks to compare the performances of the two streaming processing systems.

2 Benchmark

We conducted our benchmarking analysis using the StreamBenchmarks framework[1], a comprehensive suite of benchmark applications specifically designed for streaming processing systems.

This repository contains a set of stream processing applications taken from the literature[Bor+20], and from existing repositories[2], which have been cleaned up properly to ensure consistency and reliability. The applications can be run in a homogeneous manner and during their execution, the framework efficiently collects and records key performance statistics, such as throughput and latency, under various conditions.

The Streambenchmark's robust framework and its comprehensive collection of applications provide a solid foundation for our benchmarking endeavors, enabling us to conduct an in-depth and systematic evaluation of streaming processing systems.

Below we list the applications with the availability in different Stream Processing Engines and Libraries. We consider Apache Storm and Apache Flink to make comparison in this table2.1:

[1]https://github.com/ParaGroup/StreamBenchmarks
[2]https://github.com/GMAP/DSPBench

Application	Acronym	Apache Storm	Apache Flink
FraudDetection	FD	Yes	Yes
SpikeDetection	SD	Yes	Yes
VoipStream	VS	Yes	Yes
WordCount	WC	Yes	Yes
Yahoo! Streaming Benchmark	YSB	Yes	Yes

Table 2.1: Applications.

This repository also contains small datasets[3] used to run the applications except for the VoipStream. For this application, datasets can be generated as described here [4]. After generated, We copied the dataset files in the Datasets/VS folder respectively. The datasets are used by all versions of the same application in all the supported frameworks. For the Yahoo! Streaming Benchmark (YSB) no dataset is actually required by the present implementation (synthetic data are continously generated by Sources).

2.1 Benchmark's Application

To test the performance of the chosen stream processing benchmarks, we widely select benchmarks from different areas such as finance, network monitoring, traffic monitoring, advertising, social network, telecommunication and gaming. Finally, we choose 5 benchmarks: FraudDetection, SpikeDetection, VoipStream, WordCount and Yahoo! Streaming Benchmark.

[3]https://github.com/ParaGroup/StreamBenchmarks/tree/master/Datasets
[4]https://github.com/ParaGroup/StreamBenchmarks/blob/master/Storm/VoipStream/README.md

2.1.1 FraudDetection

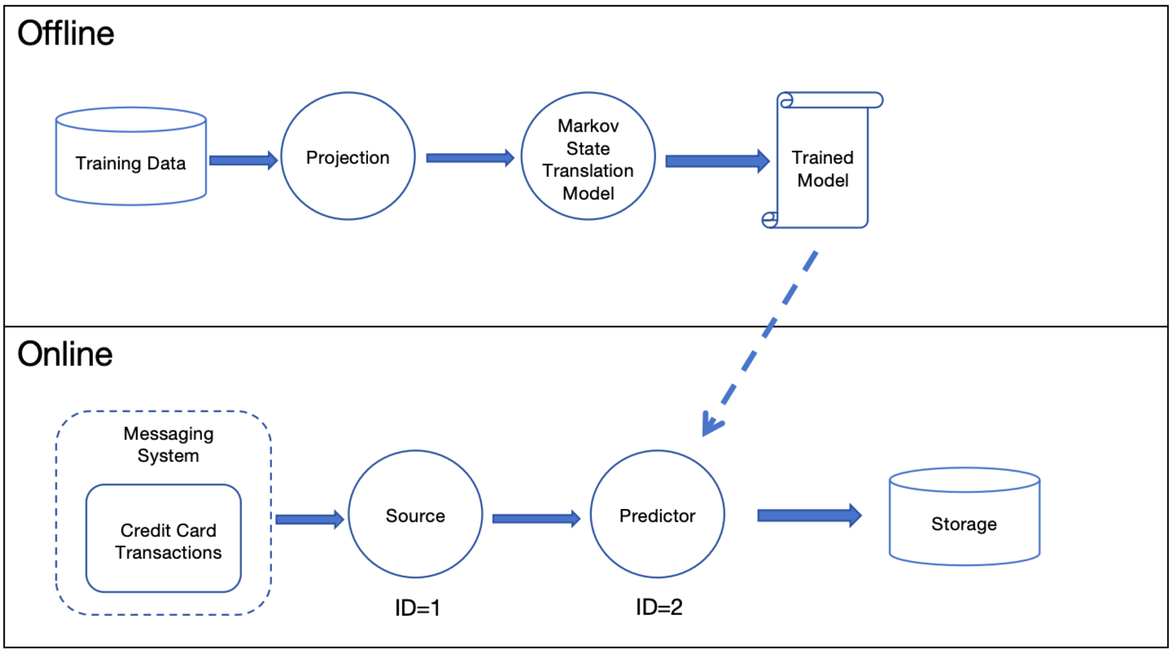


Figure 2.1: Fraud Detection

FraudDetection is a benchmark to identify credit card transactions as normal or not. The classification model is trained offline using Markov model[ZZM18]. The source operator is reponsible for cleaning the raw data sent by Messaging System. The predictor uses the trained model to determine whether the transaction is fraud or not. Finally the classification result is stored in database. The process of applying FraudDetection is shown in Figure 2.1.

2.1.2 SpikeDetection

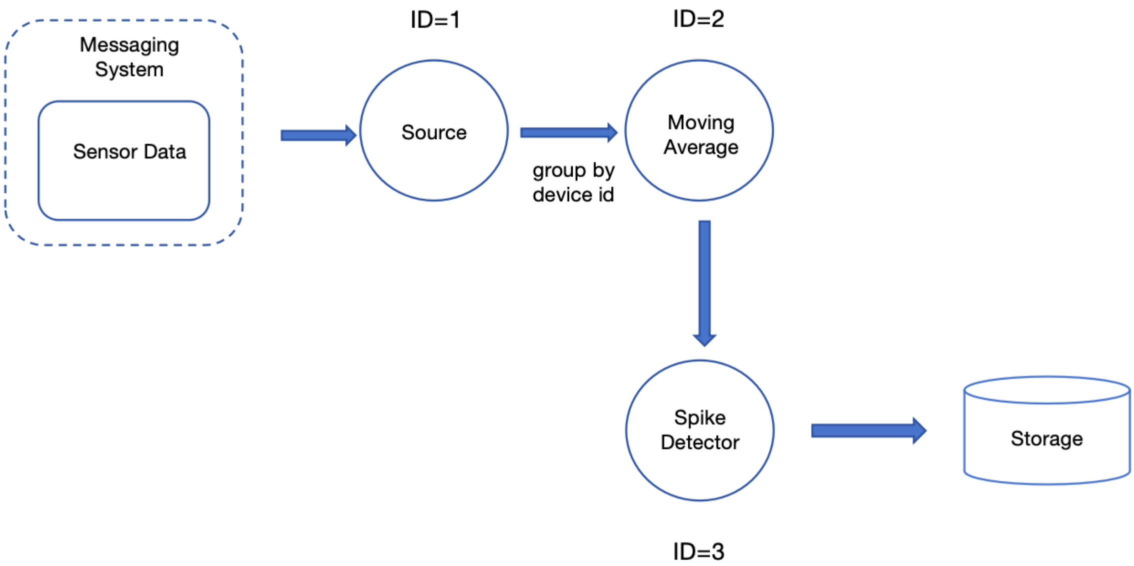


Figure 2.2: Spike Detection

The sensors are sending continuous data to the system and then the system can monitor spikes[Phu+07]. The source operator is responsible for cleaning the data. The moving average operator receives data grouped by device id and maintain a moving window. When the moving average operator receives new event, it adds the new value to the moving window, and send device id, current value and moving average to the spike detector operator. The spike detection operator receives these event and compute the relative difference between current value and moving average, determining whether this event is a spike. If it is, then the details are stored in the database. The process of applying SpikeDetection is shown in Figure 2.2.

2.1.3 VoipStream

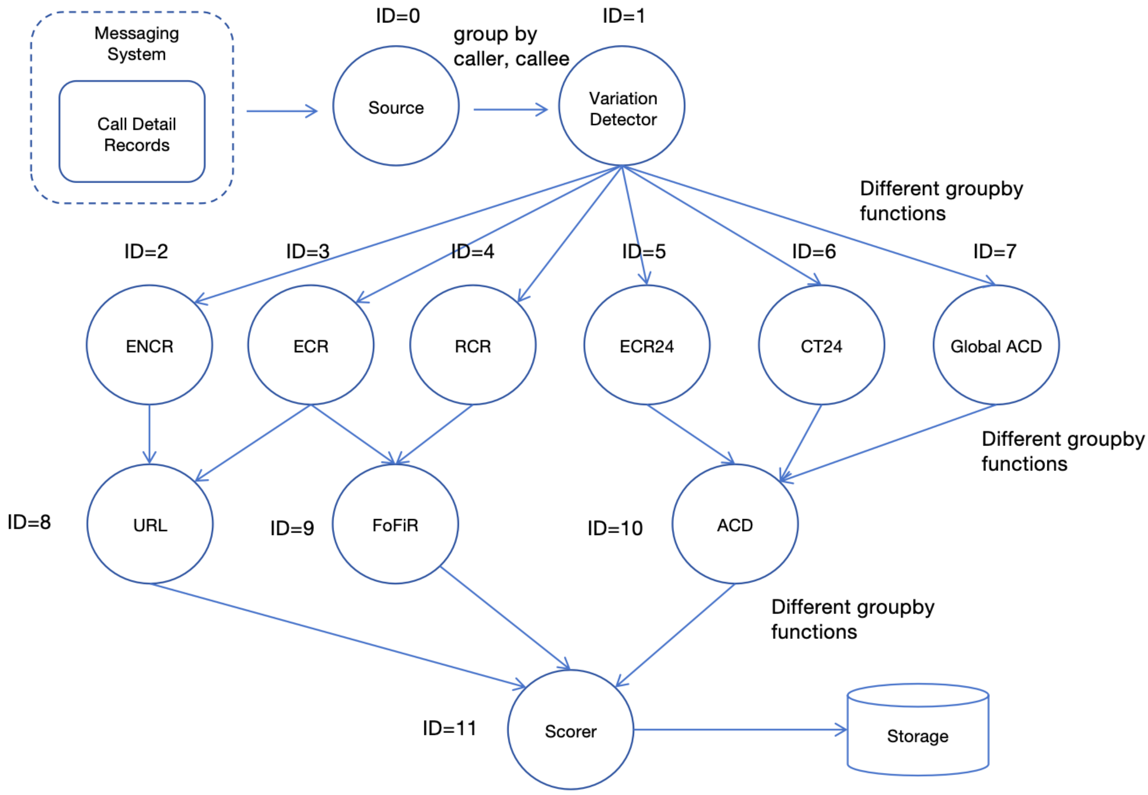


Figure 2.3: VoipStream

VoipStream is a benchmark to identify telecom spam call[Cha10]. The messaging system sends Call Records Detail to the operators. The filters use a set of filters based on time-decaying bloom filters. The operators widely use groupby distributions to manage data(group by caller or group by callee). The scorer operator receives events from filters and calculates the possibility of spam call.

The process of applying VoipStream is shown in Figure 2.3.

2.1.4 WordCount

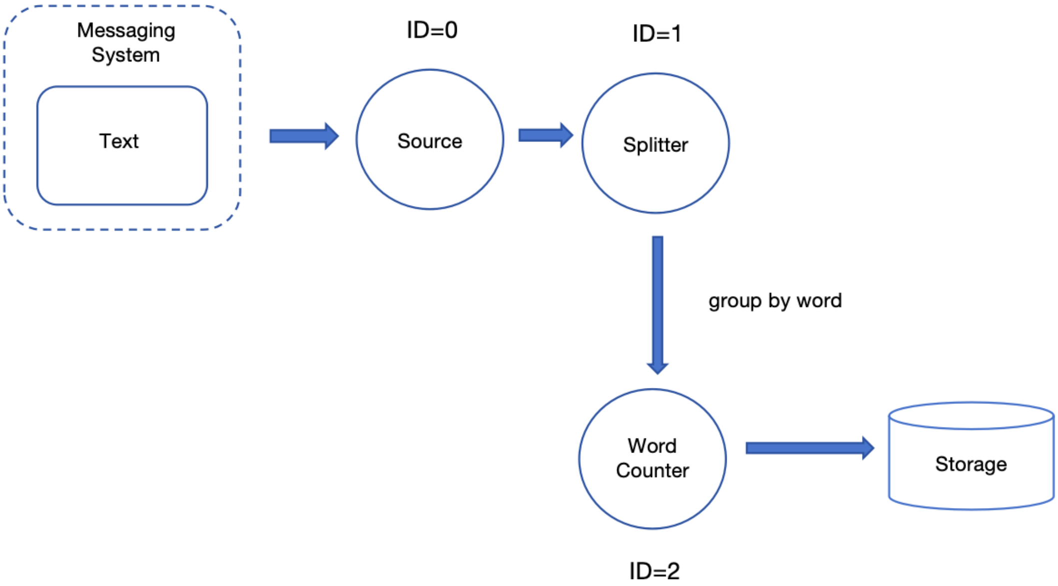


Figure 2.4: WordCount

WordCount is a synthetic benchmark to count the frequency of every word in the whole corpus[Lu+14]. The splitter operator splits sentences to words and the word counter operator counts the frequency. The process of applying WordCount is shown in Figure 2.4.

2.1.5 Yahoo! Streaming Benchmark

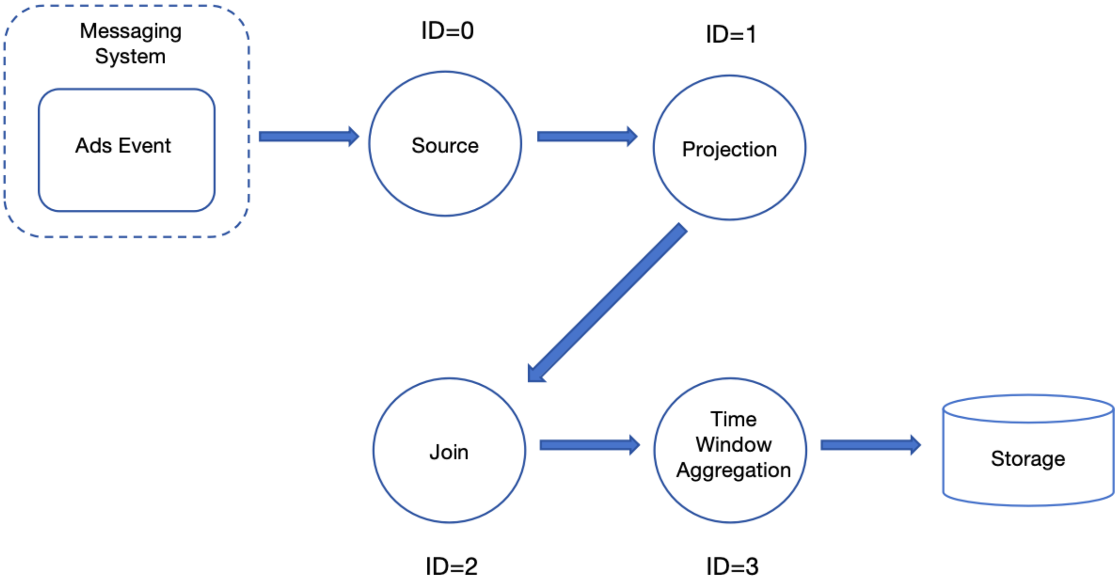


Figure 2.5: Yahoo! Streaming Benchmark

The Yahoo! Streaming Benchmark is a simple advertisement application[CCM12]. There are a number of advertising campaigns and a number of advertisements for each compaign. The goal of this benchmark is to calculate the windowed count of events per campaign.

The source operator reads the input stream data. The projection operator filters out irrelevant information. The join operator join the id of advertisement and the id of campaign. The time window aggregation operator calculates the windowed count.

The process of applying Yahoo! Streaming Benchmark is shown in Figure 2.5.

2.2 Metrics

To measure the performances of different stream processing systems, we need to clearly point out the metrics we want to adopt.

Throughput[Kar+18a] in stream processing systems refers to the rate at which the system can process and handle a certain volume of data within a given time frame. It is a crucial performance metric that quantifies the system's capacity to ingest, process, and produce results for streaming data. Throughput is typically measured in terms of events or records processed per unit of time. The higher the throughput, the better the system.

Latency[Kar+18b] in stream processing systems refers to the time it takes for an event or a piece of data to traverse the entire processing pipeline from its point of entry (ingestion) to its final output (egress). The lower the latency, the better the system.

CPU and MEM are also important metrics because they indicate how much computational and storage resources the streaming system is using. If the system uses less CPU and MEM, it means the system is more efficient.

- **Throughput**. The throughput is calculated by a counter and the timestamp. So the throughput is the ratio between the number of outputs generated and the passed running time. We use the average throughput as a metric.
- **Latency**. When entering the pipeline, a start timestamp is created. When finishing the whole pipeline, another end timestamp is created. The latency is the time difference between the 2 timestamps. We adopt the 95-th percentile as the latency of the pipeline.
- **Resource consumption**. We collect the CPU and MEM usage of the system at every time unit.

2.3 Parameters

The parameters we need to config are the number of nodes used and the number of parallelism of each operator. For example, in WordCount, we need to config 5 parameters(the number of nodes, parallelism of source operator, parallelism of splitter operator, parallelism of counter operator and parallelism of sink operator). It's usually noted as *nNodes_xSources_xSplitter_xCou*

3

Benchmark Implementation Process

3.1 Setting up the Project

3.1.1 Hardware Specification

In order to carry out the benchmark experimentation, Apple Macbook and HP EliteBook 840 G5 were used. Table 3.1 shows the hardware specification of this computer:

Table 3.1: System specifications.

System	**HP EliteBook 840G5**	**Apple Macbook**
Operating system	Ubuntu 22.04.3 LTS	MacOS 13.0
System type	64-Bit	64-Bit
CPU	Intel® Core™ i7-8650U	Apple M1 Pro
CPU frequency	1.90GHz × 8	3.2 GHz
RAM capacity	24 GB	16 GB
RAM type	DDR4	SDRAM
Hard disk capacity	512 GB	512 GB

3.2 Implementation

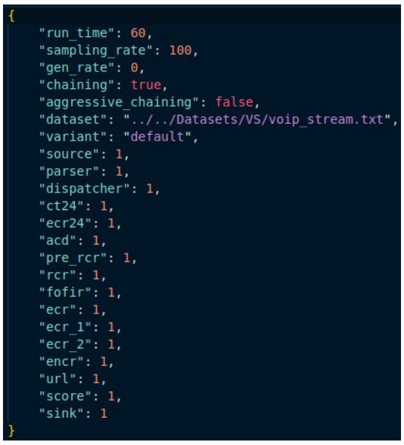
We use the suite of Benchmark Applications for Streaming Processing Systems[1]. In this project, we perform the benchmark for five applications: FraudDetection (FD), SpikeDetec-

[1]https://github.com/ParaGroup/StreamBenchmarks

tion (SD), VoipStream (VS), WordCount (WC), and Yahoo! Streaming Benchmark (YSB). The list of applications along with their parameters and configurations are shown in Table 3.2. Each configuration is identified with a label consisting of its acronym and parameters. For example, FD_1_1_1 is the label for FraudDetection with 1 Source, 1 Predictor, and 1 Sink. An exception is VoipStream, which includes 23 parameters and cannot be fully shown in label. Figure 3.1 provides the settings for all parameter in 3 configurations.

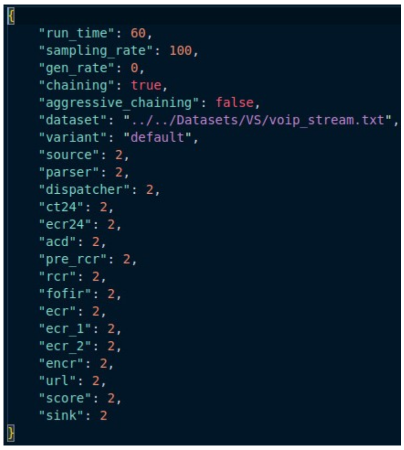
Table 3.2: Applications in the benchmark.

App	**Parameters**	**Configurations**
FD	Source, Predictor, Sink	FD_1_1_1, FD_4_4_4, FD_7_7_7
SD	Source, Moving-Average, Spike-Calculator, Sink	SD_1_1_1_1, SD_4_4_4_4, SD_7_7_7_7
VS	23 parameters defined in config.json	VS_config_1, VS_config_2, VS_config_3
WC	Source, Splitter, Counter, Sink	WC_1_1_1_1, WC_4_4_4_4, WC_7_7_7_7
YSB	Source, Filter, Joiner, Aggregate, Sink	YSB_1_1_1_1_1, YSB_4_4_4_4_4, YSB_7_7_7_7_7



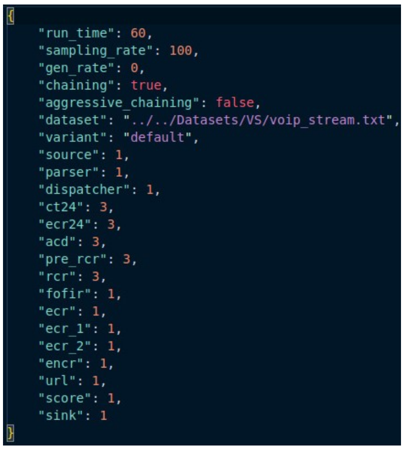

```
{
    "run_time": 60,
    "sampling_rate": 100,
    "gen_rate": 0,
    "chaining": true,
    "aggressive_chaining": false,
    "dataset": "../../Datasets/VS/voip_stream.txt",
    "variant": "default",
    "source": 1,
    "parser": 1,
    "dispatcher": 1,
    "ct24": 1,
    "ecr24": 1,
    "acd": 1,
    "pre_rcr": 1,
    "rcr": 1,
    "fofir": 1,
    "ecr": 1,
    "ecr_1": 1,
    "ecr_2": 1,
    "encr": 1,
    "url": 1,
    "score": 1,
    "sink": 1
}
```


(a) VS_config_1.json




```
{
    "run_time": 60,
    "sampling_rate": 100,
    "gen_rate": 0,
    "chaining": true,
    "aggressive_chaining": false,
    "dataset": "../../Datasets/VS/voip_stream.txt",
    "variant": "default",
    "source": 2,
    "parser": 2,
    "dispatcher": 2,
    "ct24": 2,
    "ecr24": 2,
    "acd": 2,
    "pre_rcr": 2,
    "rcr": 2,
    "fofir": 2,
    "ecr": 2,
    "ecr_1": 2,
    "ecr_2": 2,
    "encr": 2,
    "url": 2,
    "score": 2,
    "sink": 2
}
```


(b) VS_config_2.json




```
{
    "run_time": 60,
    "sampling_rate": 100,
    "gen_rate": 0,
    "chaining": true,
    "aggressive_chaining": false,
    "dataset": "../../Datasets/VS/voip_stream.txt",
    "variant": "default",
    "source": 1,
    "parser": 1,
    "dispatcher": 1,
    "ct24": 3,
    "ecr24": 3,
    "acd": 3,
    "pre_rcr": 3,
    "rcr": 3,
    "fofir": 1,
    "ecr": 1,
    "ecr_1": 1,
    "ecr_2": 1,
    "encr": 1,
    "url": 1,
    "score": 1,
    "sink": 1
}
```


(c) VS_config_3.json

Figure 3.1: Parameter configurations for VoipStream.

The bash script for computing metrics is provided below.

```bash
#!/bin/bash

configs=("1 1 1" "4 4 4" "7 7 7")

for config in "${configs[@]}"
do
    echo "Config: $config"
    # Set the command for running the applicaiton
    command="java -cp target/FraudDetection-1.0.jar FraudDetection.
    FraudDetection --rate 0 --sampling 100 --parallelism $config --
    chaining"

    # Run the command in the background
    $command > "command_output.txt" 2>&1 &

    # Get the PID of the last background process
    pid=$!

    num_cores=$(nproc)
    avg_cpu=0
    avg_mem=0
    count=0
    # While application is running...
    while kill -0 $pid 2> /dev/null; do
        # Get CPU and MEM
        cpu=$(ps aux | grep $pid | grep -v grep | awk '{print $3}')
        avg_cpu=$(echo "$avg_cpu + $cpu" | bc)
        mem=$(ps aux | grep $pid | grep -v grep | awk '{print $4}')
        avg_mem=$(echo "$avg_mem + $mem" | bc)
        count=$((count + 1))

        sleep 1
    done

    output=$(cat "command_output.txt")
    echo "$output$"
    throughput=$(echo "$output" | grep -oE "Measured throughput:
    [0-9.]+" | awk '{print $3}')
    echo "$throughput"
```

```
37     latency=$(jq --arg key "95" '.[$key]' "metric_latency.json")
38     avg_cpu=$(echo "$avg_cpu / $count / $num_cores" | bc)
39     avg_mem=$(echo "$avg_mem / $count" | bc)
40 
41     echo "Flink_FD_$config, $throughput, $latency, $avg_cpu%, $avg_mem%
    " >> "../../results.csv"
42 
43 done
```

Listing 3.1: Bash script for benchmarking streaming systems.

We set different configurations for the test in **line 3**, depending on the specific applications. Take FraudDetection as an example, this application has three component in its data flow, namely Source, Predictor and Sink, so we have to set the parallelism for these three operators. **Line 9** starts the application with the chosen configuration for each operator (Source, Predictor and Sink). The parameter `sampling` indicates that latency values are gathered every 100 received tuples in the Sink, The parameter `rate` specifies the data generation at full speed. **Line 12** runs this application in background, and while is is still, we get and accumulate the CPU and MEM usage every 1 second **from line 22 to line 31**. Finally, we average the CPU and MEM usage after the application has finished, as well as collect the latency and throughput. We save all results to a file `results.csv`.

We run this script for different types of application in streaming system by modifying the configurations and the running command. Each application has a separate set of parameters, whose settings and results are provided in details in Chapter 4.

4 Results and Discussion

As mentioned in the previous chapter, the suite of Benchmarks was performed on Ubuntu, with tests of 4 different metrics: Throughput,Latency,CPU utilization and Memory Utilization. In this chapter, we will be focusing on discussing the test result as the configurations increases and making comparison between Flink and Storm performances.

4.1 Throughput Test

The bar chart in figure 4.1 presents a benchmark comparison of throughput between two stream processing frameworks: Flink (in red) and Storm (in blue), across various application configurations. Throughput is measured in tuples per second, and the y-axis is scaled logarithmically to 1e6 (one million tuples per second).

In most configurations, Flink demonstrates a higher throughput compared to Storm, indicating that Flink can process more data at a faster rate. Notably, in configurations FD_4_4_4, SD_4_4_4_4, YSB_4_4_4_4 and YSB_7_7_7_7, Flink's throughput significantly outperforms Storm's. This trend is consistent across configurations with different numbers and combinations, suggesting Flink's superior performance in these scenarios.

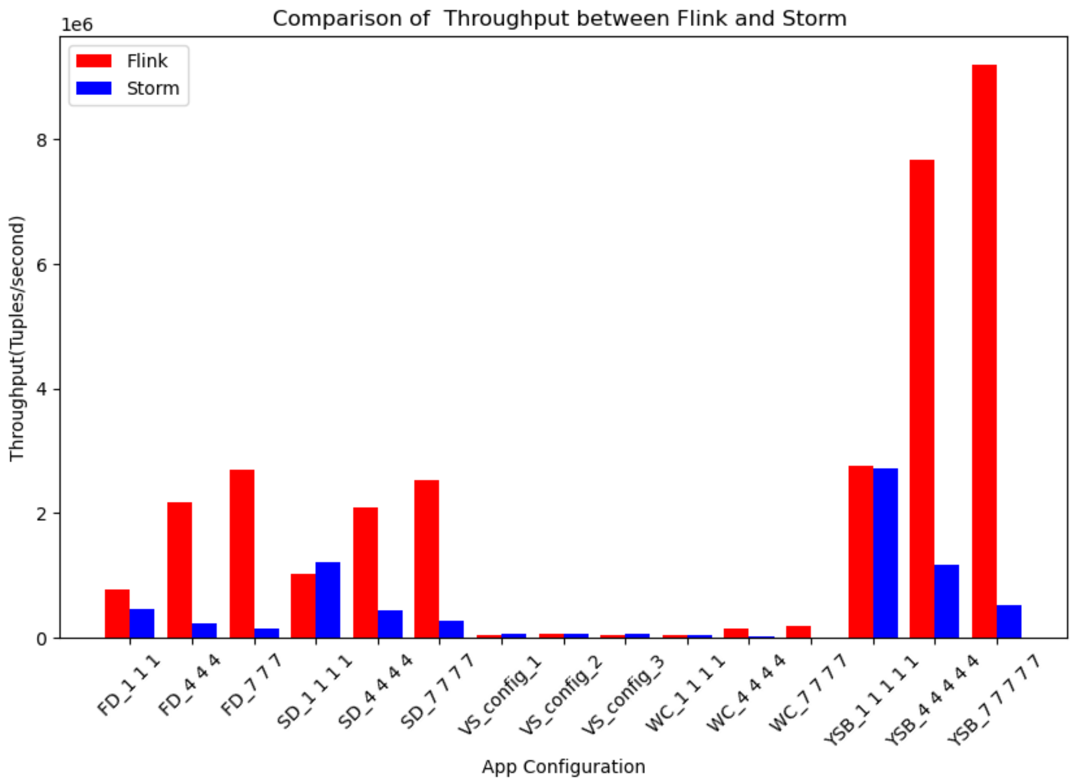


Figure 4.1: Bar Chart of Comparison of Throughput between Flink and Storm

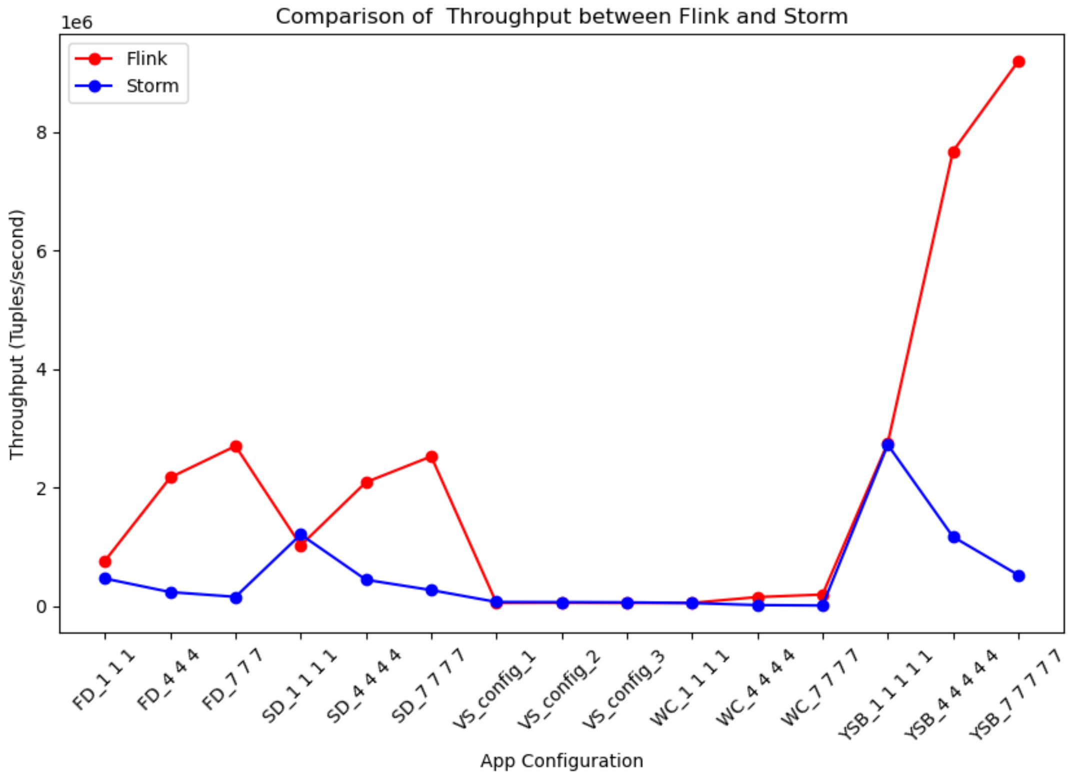


Figure 4.2: Line Chart of Comparison of Throughput between Flink and Storm

However, in Application VoipStream 4.3, Storm narrows the gap, event Storm's throughput is greater than Flink's. Yet, it does not surpass Flink's throughput in other given configuration.

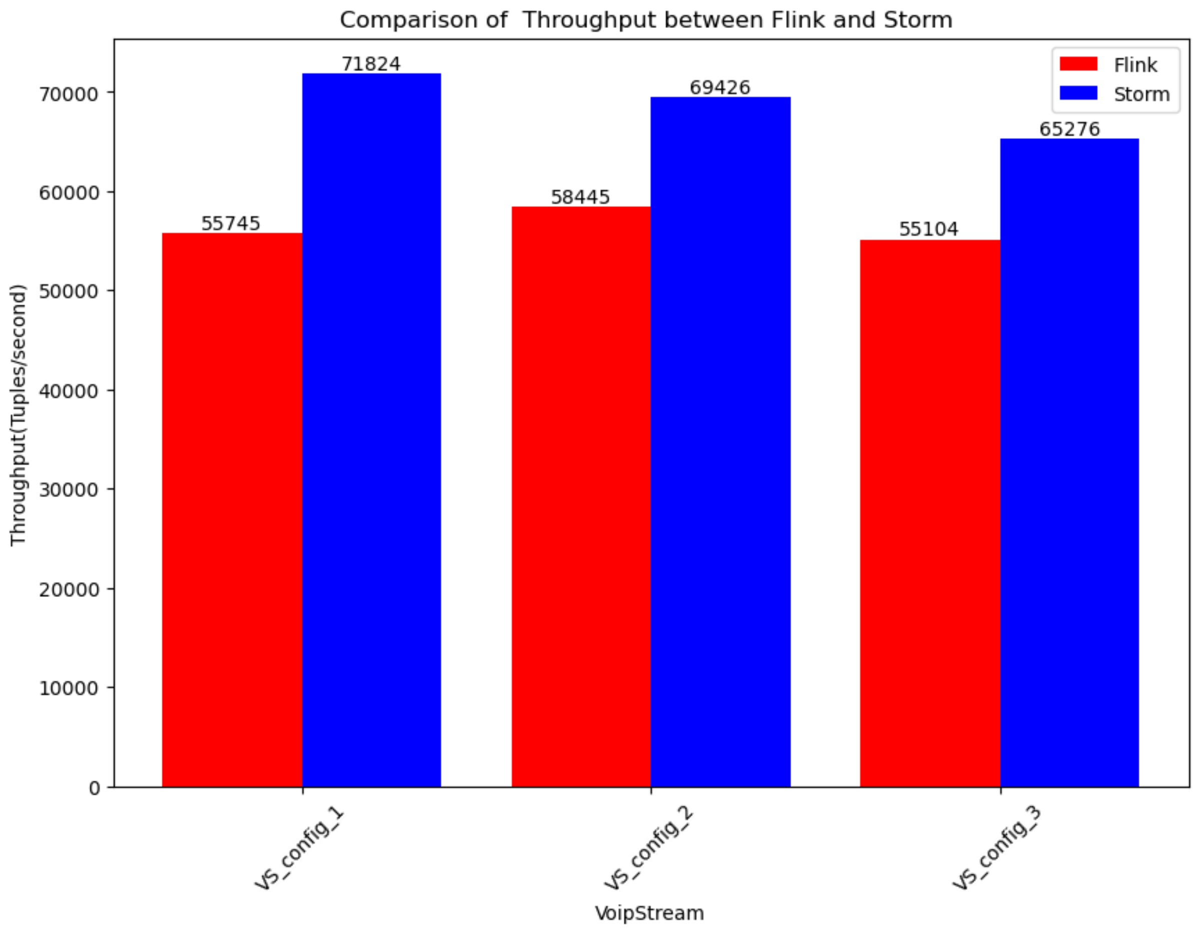


Figure 4.3: Comparison of Throughput of VoipStream between Flink and Storm

As figure 4.4 shows, the most striking difference is observed in configuration YSB_7_7_7_7_7, where Flink's throughput peaks at just over 9 million tuples per second, vastly exceeding Storm's throughput. This suggests that for this particular workload configuration, Flink is highly optimized.

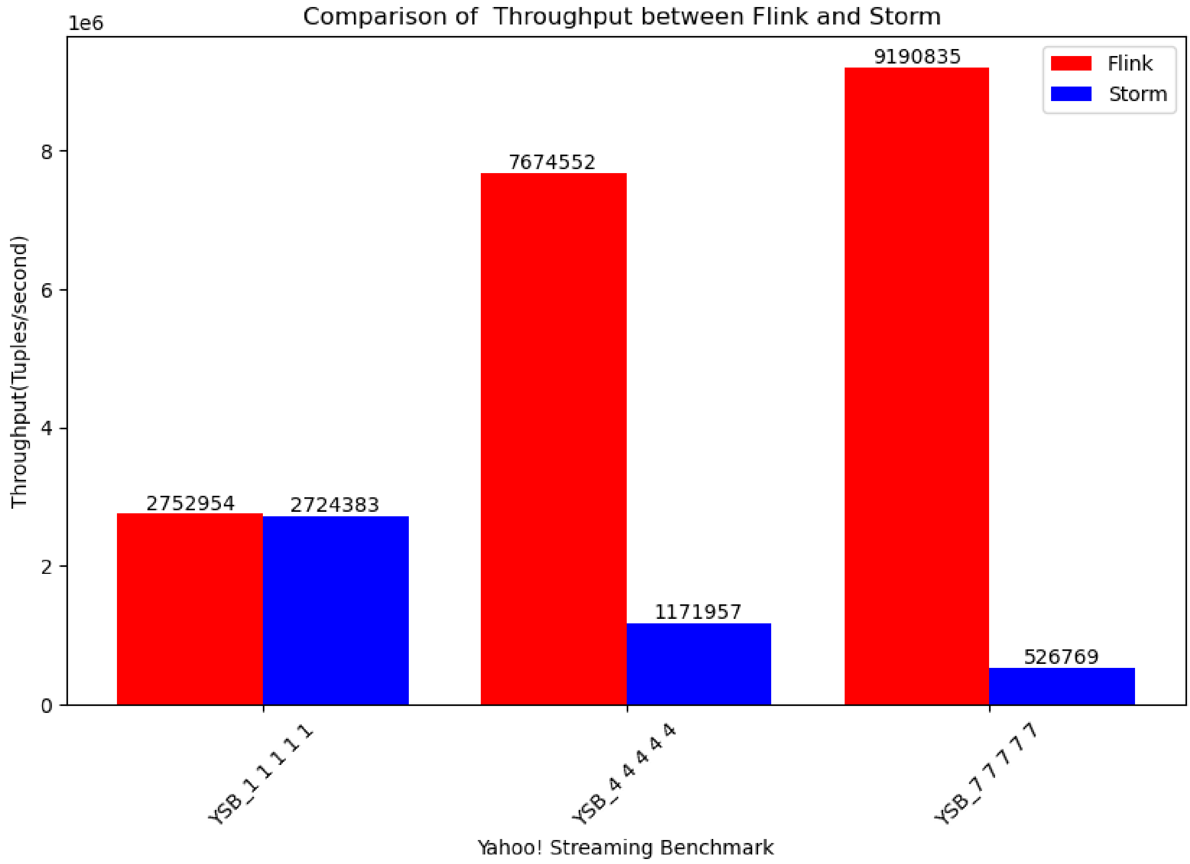


Figure 4.4: Comparison of Throughput of YSB between Flink and Storm

Overall, the benchmark results show that Flink's throughput performance tends to be better in most of the tested configurations. This may indicate that Flink is robust and efficient in handling large data streams. Users choosing between these two frameworks may want to consider using Flink for high throughput needs.

4.2 Lantency Test

The bar chart in figure 4.5 illustrates a benchmark comparison of latency between the Flink and Storm streaming frameworks across various application configurations, with latency measured in milliseconds. The vertical axis is logarithmically scaled to 1e6 milliseconds for visual clarity.

Observing the chart, both Flink (red bars) and Storm (blue bars) exhibit increased latency as the complexity of the application configurations grows. In simpler configurations like FD_1_1_1, both frameworks maintain relatively low latency, but as the configurations evolve to FD_4_4_4 and beyond, a noticeable increase in latency is evident for both frameworks.

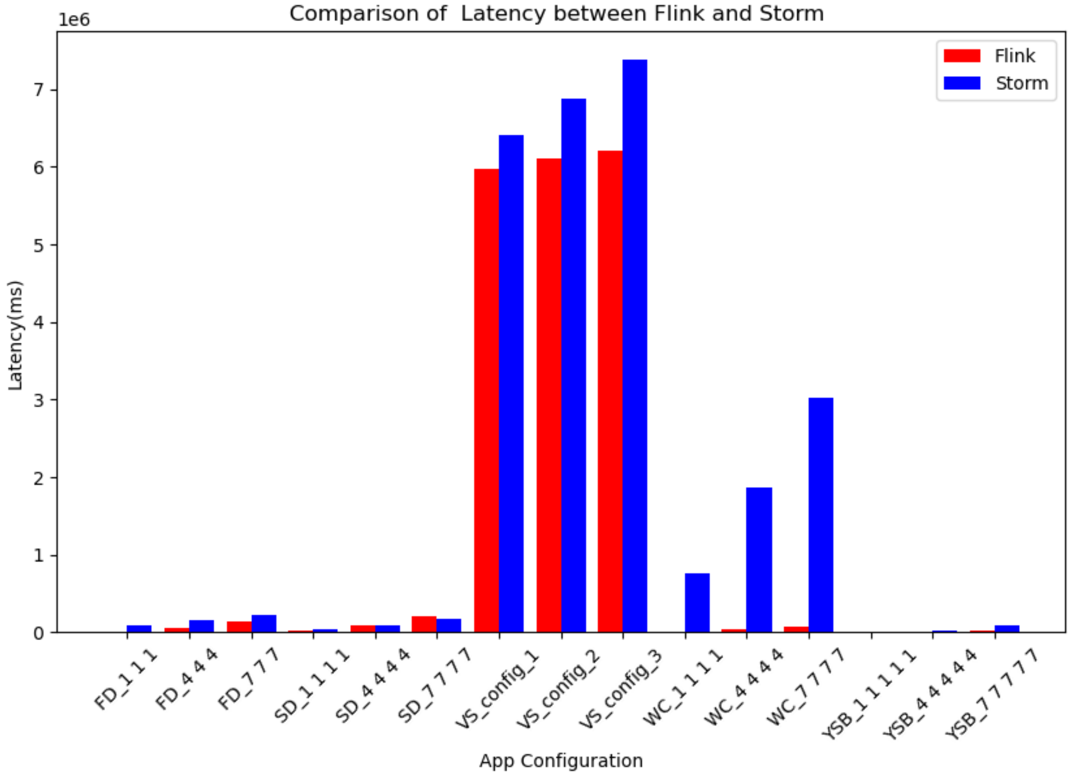


Figure 4.5: Bar chart of Comparison of Latency between Flink and Storm

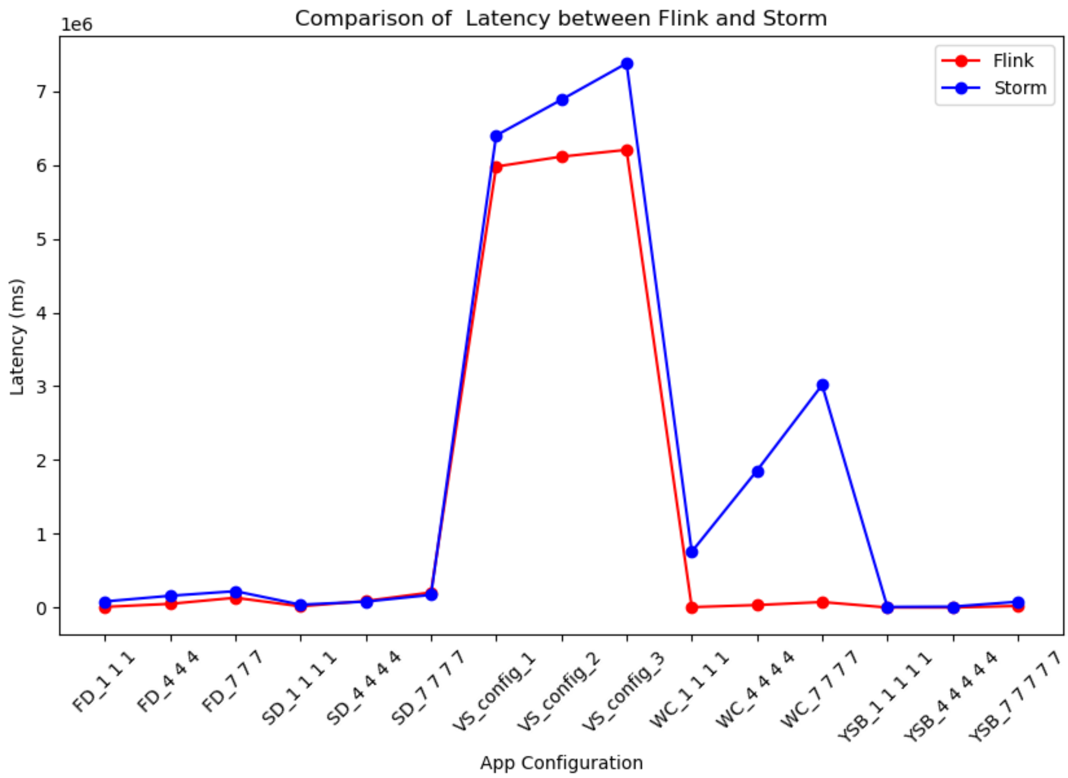


Figure 4.6: Line chart of Comparison of Latency between Flink and Storm

For configuration SD_4_4_4_4 and SD_7_7_7_7, Flink shows a slightly higher latency compared to Storm, suggesting that in this scenario, Storm may handle latency more efficiently. In contrast, for other configurations, Flink significantly outperforms Storm, indicating better latency handling in that particular configuration.

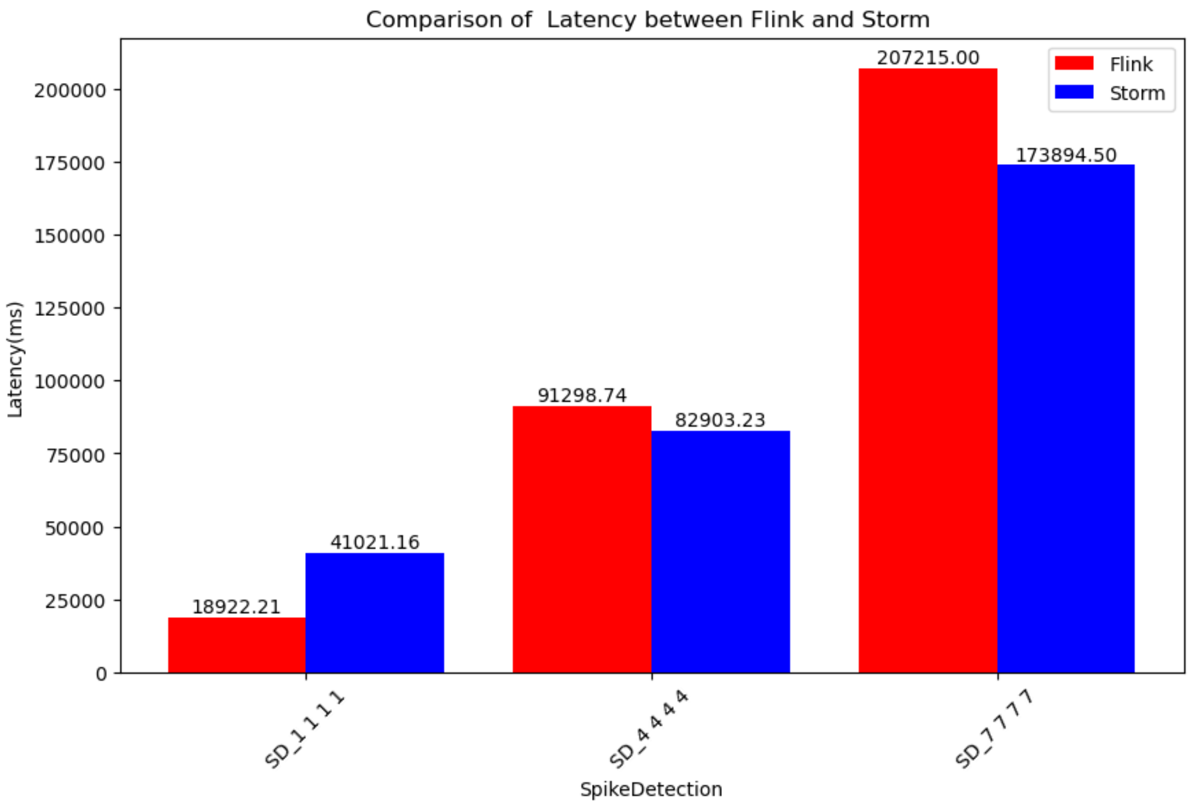


Figure 4.7: Comparison of Latency of SD between Flink and Storm

As figure 4.8 shows,the most substantial disparity appears in the WordCount Application, where Flink's latency is drastically lower than Storm's, implying Flink's superior performance under this specific workload.

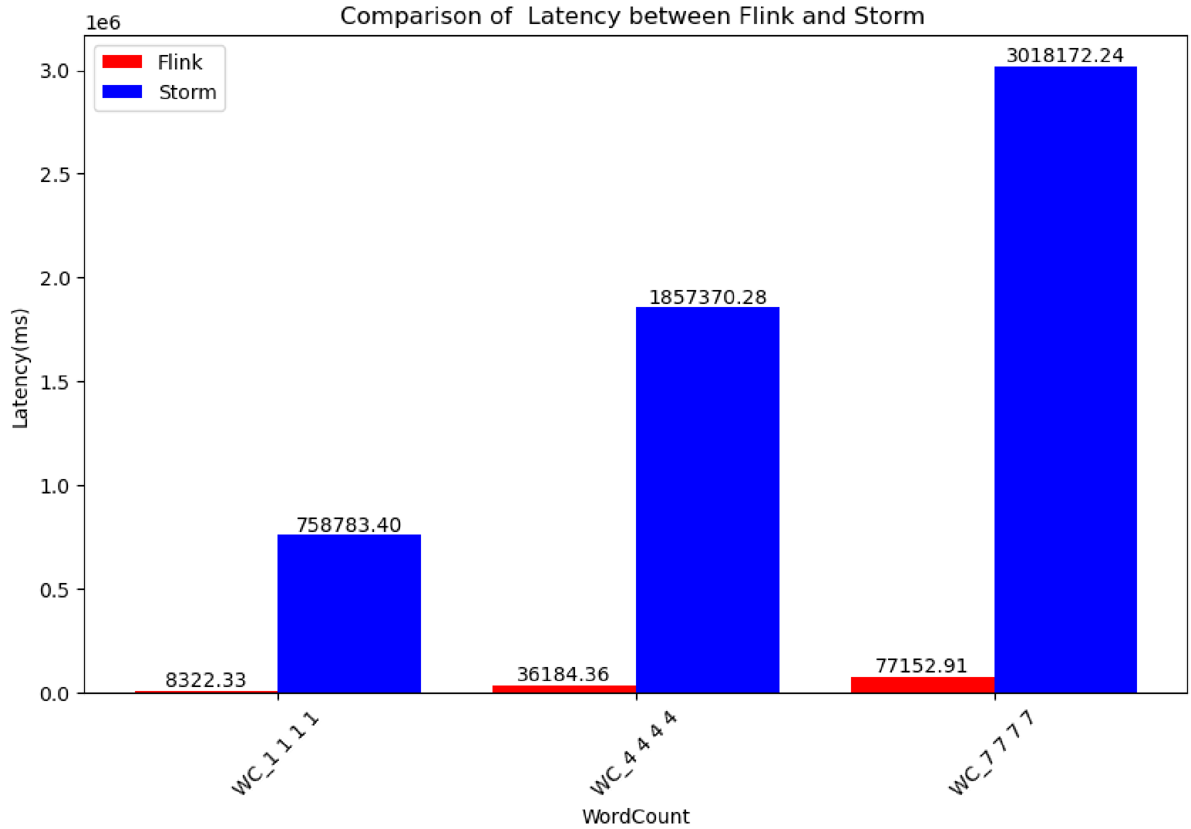


Figure 4.8: Comparison of Latency of WC between Flink and Storm

In conclusion, the choice between Flink and Storm for minimizing latency may depend heavily on the specific application configuration and workload pattern. This benchmark indicates that while Flink may offer lower latency in certain configurations, Storm may be preferable in others, especially those resembling the SpikeDetection workload. Users should consider these performance characteristics in relation to their specific use cases when selecting a stream processing framework.

4.3 Resource Consumption

4.3.1 CPU Utilization Test

The provided bar chart in fig 4.9 offers a benchmark comparison of CPU usage between the Flink and Storm streaming frameworks across a series of application configurations. CPU usage is indicated as a percentage, showing how much of the CPU resources each framework utilizes during operation.

Observing the chart, both Flink (red bars) and Storm (blue bars) exhibit increased CPU usage as the complexity of the application configurations grows.

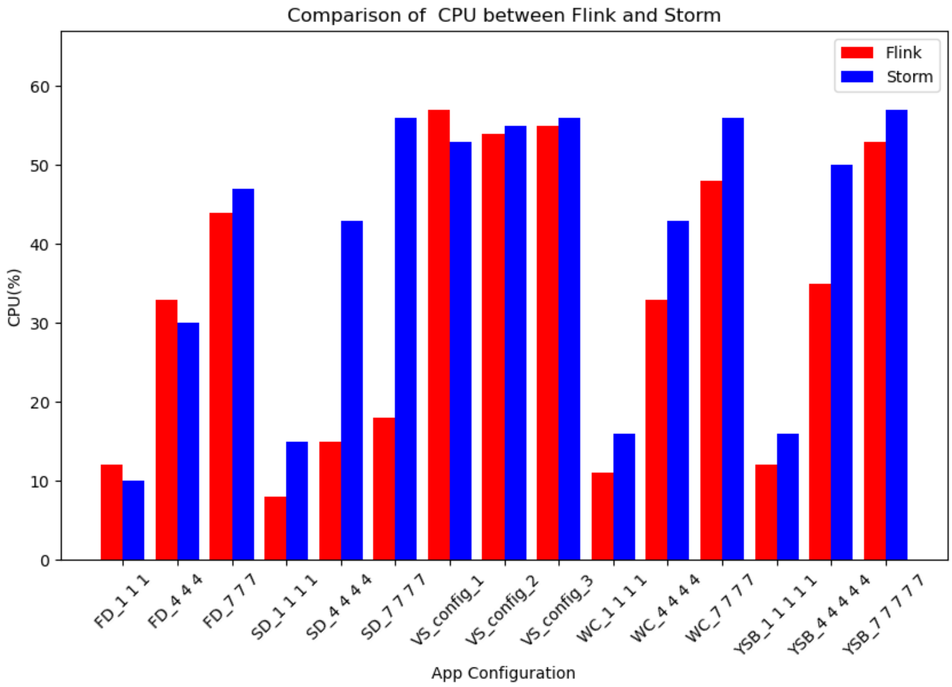


Figure 4.9: Bar Chart of Comparison of CPU between Flink and Storm

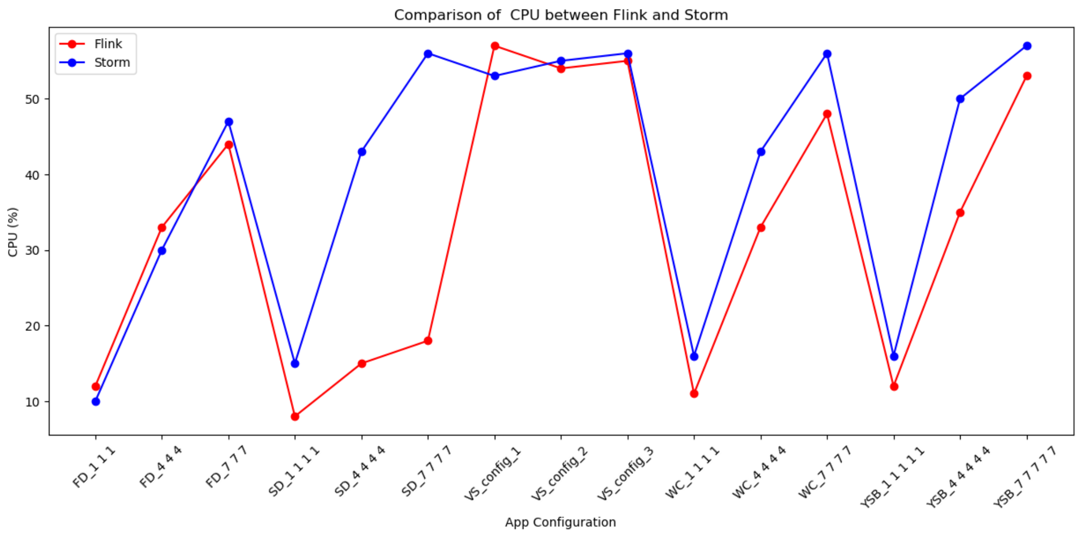


Figure 4.10: Line Chart of Comparison of CPU between Flink and Storm

From the chart4.11, it is observable that CPU utilization varies significantly depending on the application configuration. For the initial configurations (FD_1_1_1 and FD_4_4_4), Flink tends to consume more CPU than Storm, suggesting that Storm is more CPU-efficient under these conditions. As the configurations progress, Flink shows a marked increase in CPU usage, surpassing Storm in the FD_7_7_7_7 configuration.

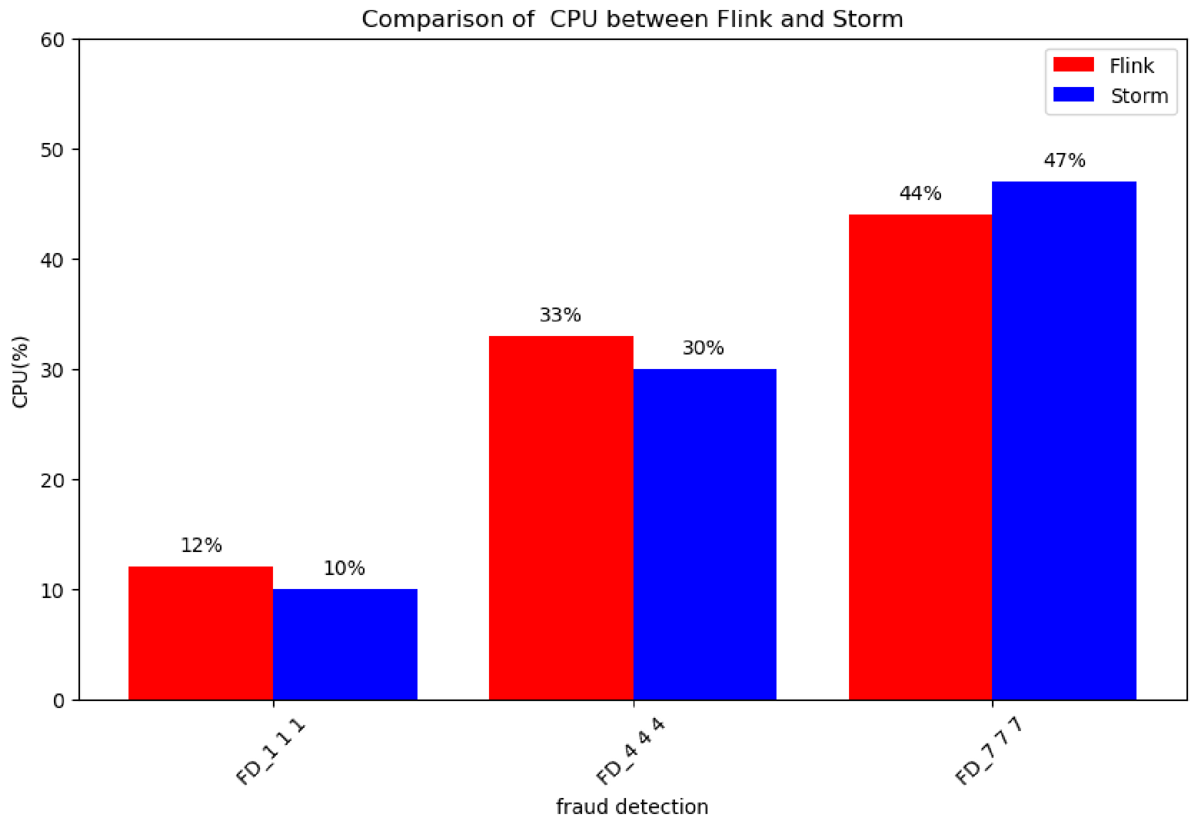


Figure 4.11: Comparison of CPU of FD between Flink and Storm

Finally, in the WC and YSB configurations, both frameworks exhibit a progressive increase in CPU utilization as the configuration intensifies from WC_1_1_1 to WC_7_7_7 and from YSB_1_1_1 to YSB_7_7_7. However, Flink consistently uses more CPU than Storm across these configurations, suggesting that Storm might be more CPU-efficient for these specific benchmark settings.

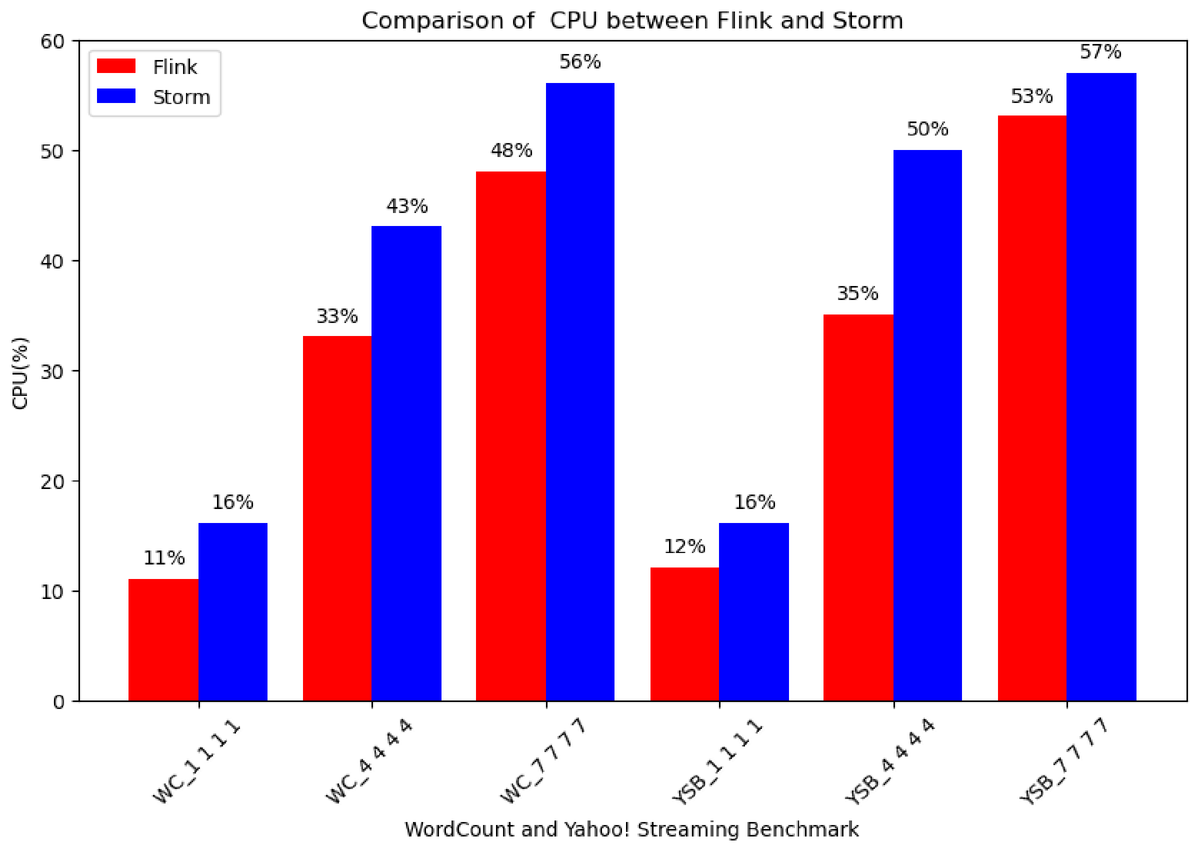


Figure 4.12: Comparison of CPU of WC and YSB between Flink and Storm

In summary, this benchmark indicates that the choice between Flink and Storm may depend on the specific requirements of the application configuration, as each framework exhibits strengths in different scenarios. For applications where CPU efficiency is paramount, Storm may be the preferable choice in most of the provided configurations, although Flink demonstrates better efficiency in most instances. Users should consider these findings in conjunction with other performance metrics and framework features when selecting a streaming solution for their needs.

4.3.2 MEM Utilization Test

The bar chart 4.13presents a comparison of memory (MEM) utilization, expressed as a percentage, between the Flink (red bars) and Storm (blue bars) streaming frameworks across various application configurations.

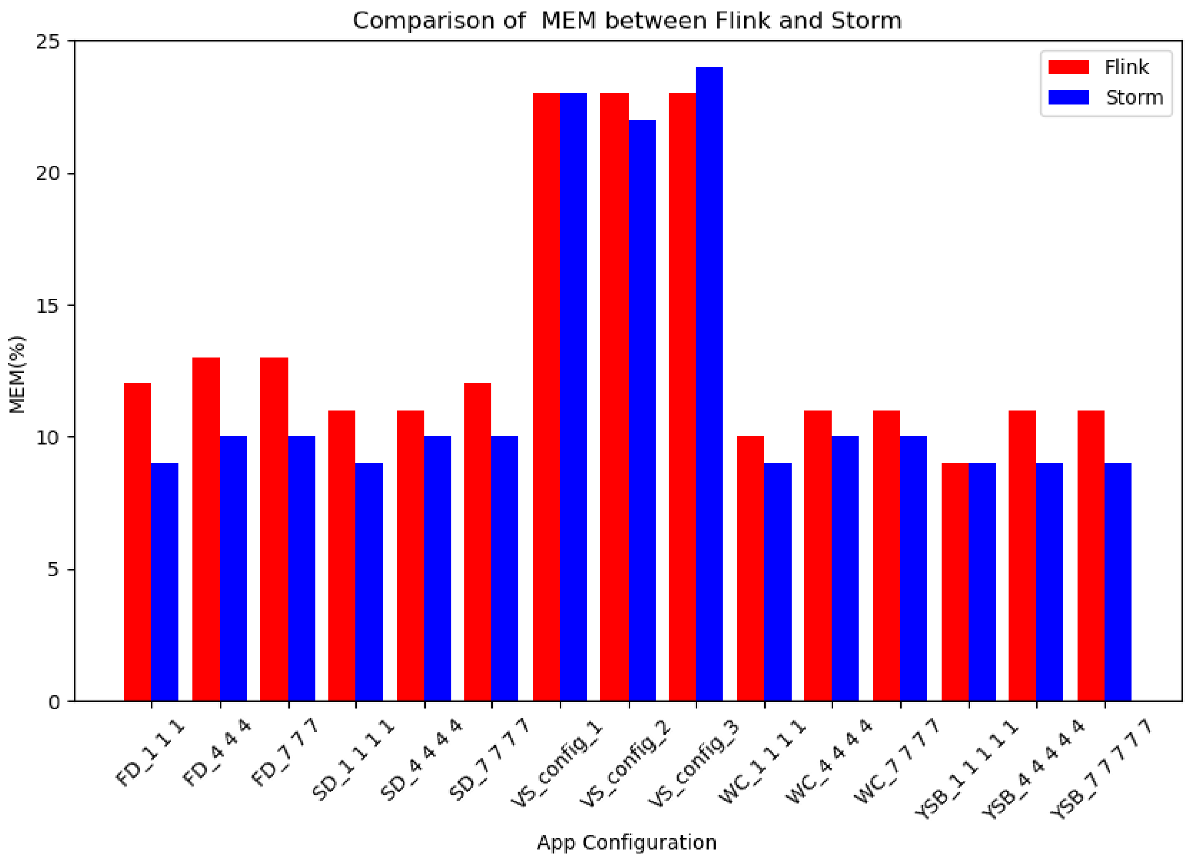


Figure 4.13: Bar Chart of Comparison of MEM between Flink and Storm

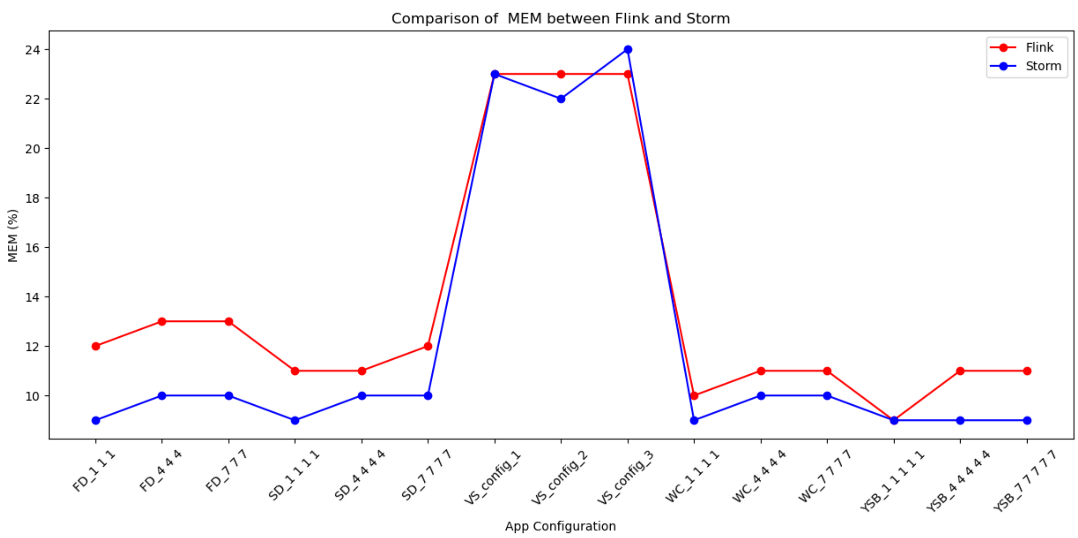


Figure 4.14: Line Chart of Comparison of MEM between Flink and Storm

In the VS application configurations (VS_config_3), Flink exhibits lower memory utilization compared to Storm.This is the only place where Flink's memory utilization is lower than Storm's.

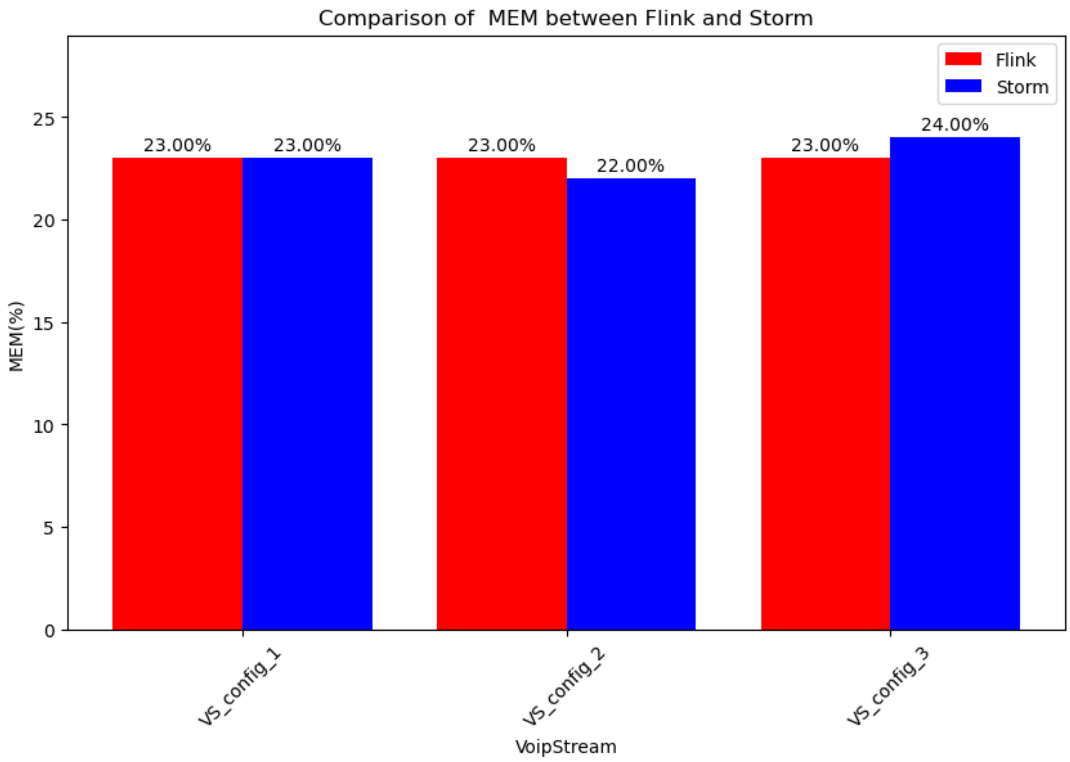


Figure 4.15: MEM Utilization of VoipStream of Flink

For the other application configurations , we see that Storm's memory usage is consistently lower than Flink's, suggesting that Storm is more memory-efficient for the almost all of applications. In addition, we find that for the same applications with different configurations, their memory occupancy is almost equal, with very little variation.

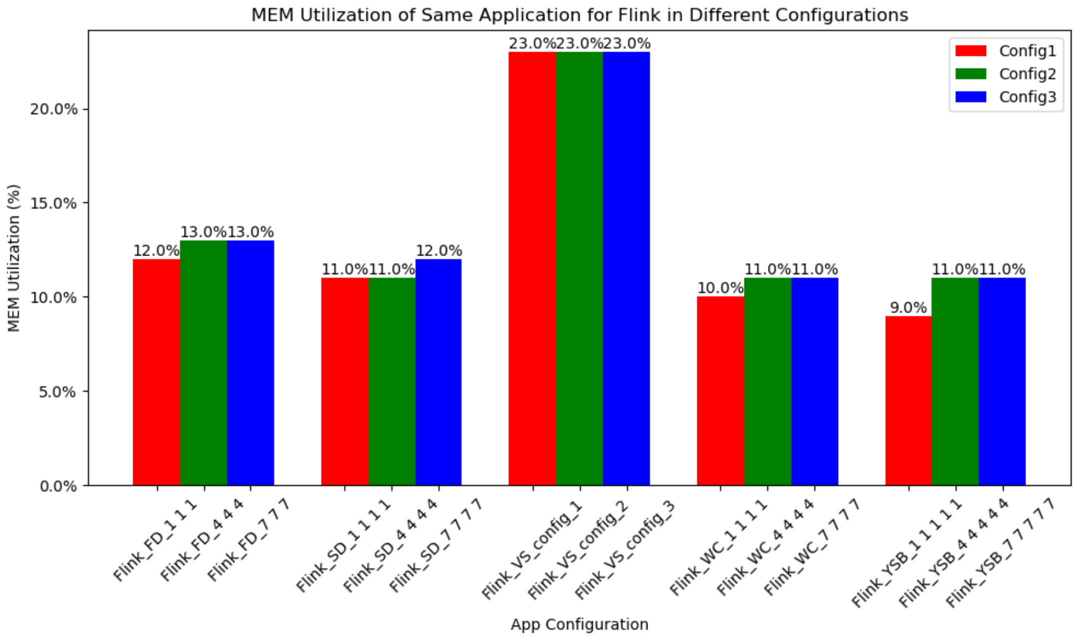


Figure 4.16: MEM Utilization of Flink

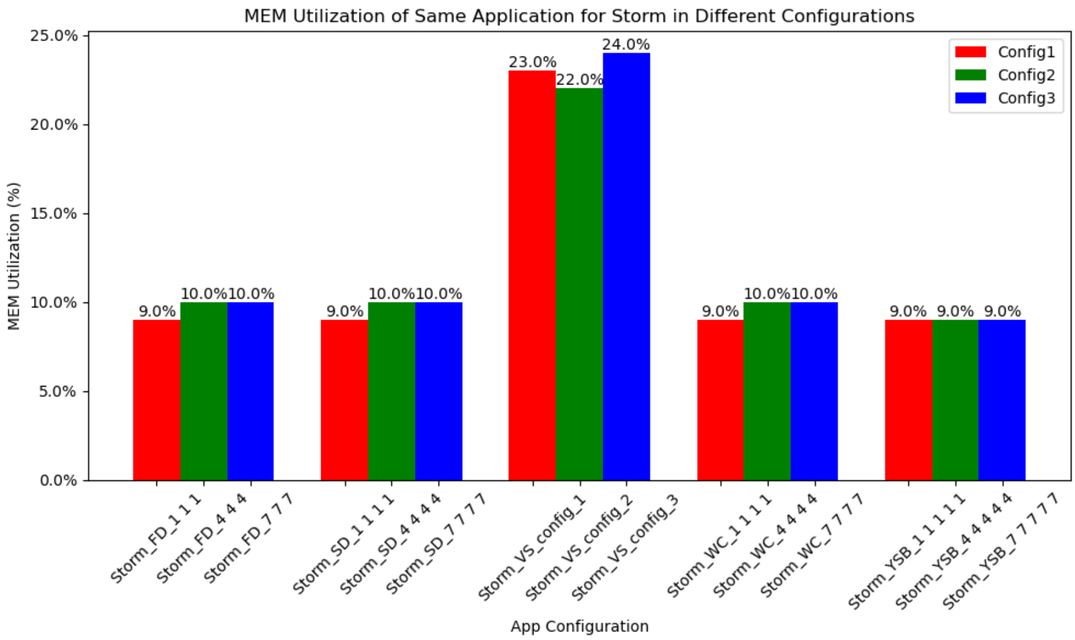


Figure 4.17: MEM Utilization of Storm

Overall, the chart suggests that Storm tends to be more memory-efficient across the range of applications and configurations tested, with some exceptions where Storm's memory usage is competitive. Users selecting a streaming framework might consider Flink for scenarios where memory efficiency is critical, especially within the FD and YSB applications as indicated by this benchmark. However, the specific needs of the application and other performance factors should also be taken into account when making a decision.

4.4 Discussion

The results show that Flink performs better than Storm for the three chosen applications. It generally provides higher throughput and lower latency. Flink not only handles larger volumes of data effectively (as evidenced by higher throughput) but also processes individual data units faster (indicated by lower latency). This combination of high throughput and low latency suggests that Flink is more efficient and suitable for scenarios that require both fast processing of large data streams and quick response times. But the choice between Flink and Storm would depend on the specific requirements of a given application, particularly in terms of data volume and processing speed.

In terms of resource utilization, CPU usage of Flink is lower than Storm but the MEM utilization of Flink is higher than Storm. It suggests a trade-off between CPU and memory resources when choosing between Flink and Storm for stream processing tasks. Flink appears to be more CPU-efficient but at the cost of higher memory consumption, whereas Storm, while being more demanding on CPU resources, tends to be more memory-efficient. This kind of information is crucial when architecting systems, as it helps in making informed decisions based on the resource availability and the specific requirements of the use case.

5 Streaming DataBase Use Case

Our Streaming Database Use Case is a Product Dataset gotten from Kaggle [1]. The Product dataset contains list of product with their prices and other details you might see in supermarket. Below is the attribute contained in the dataset:

- S.No: Serial number
- BrandName: The brand of the product.
- Product ID: A unique identifier for each product.
- Product Name: The name of the product.
- Brand Desc: A description of the brand.
- Product Size: Size or dimensions of the product.
- Currency: The currency used in pricing
- MRP (Maximum Retail Price): The maximum price at which the product can be sold.
- SellPrice: The selling price of the product.
- Discount: Any discounts applied to the product.
- Category :The category to which the product belongs

[1] https://www.kaggle.com/datasets/sujaykapadnis/products-datasets

The purpose of this use case is to test the two databases and perform an analytical operation on a product sales dataset.The Objectives include:

- To aggregate product sales data by category, calculate total sales and sales count,
- To sort the total sales in descending order,
- To calculate the run-time of each database.

This is a batch data analysis. This analysis is crucial for understanding sales trends, identifying top-performing categories, and making informed decisions regarding inventory management, marketing strategies, and product development. The use case was done with Java Programming Language and the IDE used was IntelliJ IDEA Community Edition.

5.1 Apache Flink

5.1.1 Processing Workflow in Apache Flink

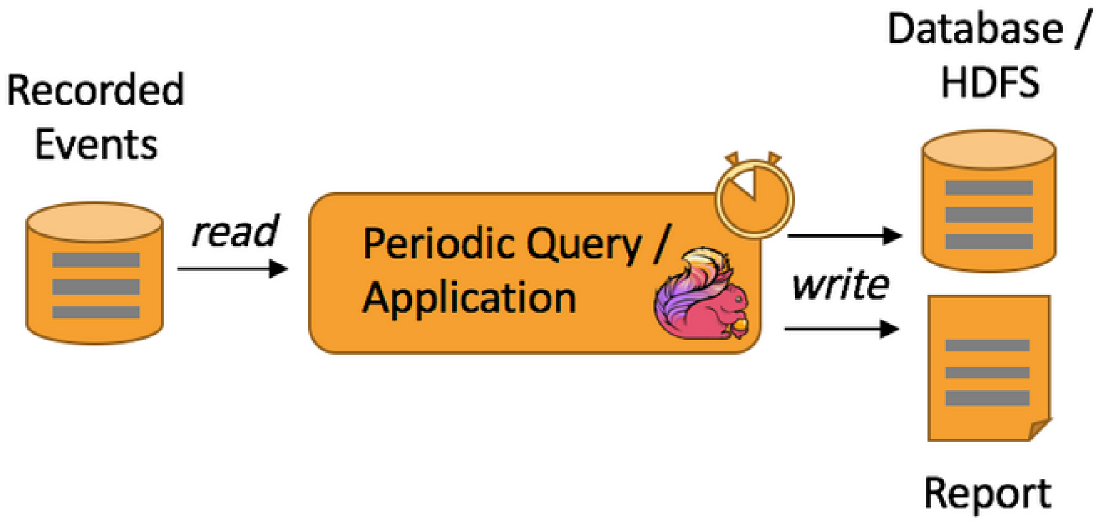


Figure 5.1: Apache Flink Analysis Architecture

Setup Execution Environment

This is use to initialize the Flink execution environment, which is the context in which Flink jobs are executed. Below is the code to initialize the environment:

```
final ExecutionEnvironment env = ExecutionEnvironment.
    getExecutionEnvironment();
```

Listing 5.1: Java script environment setup.

Read and Parse the Input Data

```
DataSource<Products> Product = env
    .readCsvFile("/home/pce/Music/StockAnalysis/DataSets/Product.csv")
    .ignoreFirstLine()
    .pojoType(Products.class, "S_No", "BrandName", "Product_ID", "
    Product_Name", "Brand_Desc", "Product_Size", "Currency", "MRP", "
    SellPrice", "Discount", "Category");
```

Listing 5.2: Reading and Parsing the Data

This block reads data from a CSV file containing product information.

Data Transformation

Each Products instance is transformed into a ProductSalesDTO instance. This transformation extracts the category, sellPrice, and product Name from each product, and initializes the sales count to 1. The ProductSalesDTO class is a class designed to store sales information.

```
DataSet<ProductSalesDTO> ProductSales = Product
    .map(new MapFunction<Products, ProductSalesDTO>() {
        @Override
        public ProductSalesDTO map(Products product) throws Exception {
            return new ProductSalesDTO(
                product.getCategory(),
                product.getSellPrice(),
```

```
                1, // Count each product as one sale
                product.getProduct_Name()
            );
        }
    })
    .returns(ProductSalesDTO.class);
```

Listing 5.3: Data Transformation

Aggregating Sales Data by Category

```
DataSet<ProductSalesDTO> aggregatedSales = ProductSales
    .groupBy("category")
    .reduce(new ReduceFunction<ProductSalesDTO>() {
        @Override
        public ProductSalesDTO reduce(ProductSalesDTO value1,
   ProductSalesDTO value2) throws Exception {
            return new ProductSalesDTO(
                value1.getCategory(),
                value1.getTotalSales() + value2.getTotalSales(),
                value1.getCount() + value2.getCount(),
                value1.getProduct_name()
            );
        }
    });
```

Listing 5.4: Aggregating Sales Data

From the code above, Sales data is aggregated by category using a ReduceFunction. For each category, it sums the total sales and counts the number of sales.

Sorting and Outputting the Results

The aggregated results are sorted in descending order based on the total sales. The sorted results are printed to the console. This is shown below:

```
aggregatedSales.sortPartition("totalSales", Order.DESCENDING).print();
```

Listing 5.5: Sorting Sales Data

Writing Output to a CSV File

The OutputFormat interface is implemented to write the aggregated sales data to an output CSV file and was then sent to Grafana for reporting. This is show below:

```
aggregatedSales.output(new OutputFormat<ProductSalesDTO>() {
      private transient BufferedWriter writer;

      @Override
      public void configure(Configuration configuration) {
        // Configuration steps (if needed) can be here
      }

      @Override
      public void open(int taskNumber, int numTasks) throws IOException {
        File outputFile = new File("/home/pce/Music/StockAnalysis/Output/new_output.csv");

        // 'true' in FileWriter constructor for appending to the file
        // If you want to overwrite, set this to 'false'
        this.writer = new BufferedWriter(new FileWriter(outputFile, false));

        // Check if the file is newly created or already exists
        if (outputFile.length() == 0) {
          // Write the header line
          writer.write("Category,Total_Sales,Sales_Count,Product_Name");
          writer.newLine();
    }
}
```

Listing 5.6: Output to CSV File

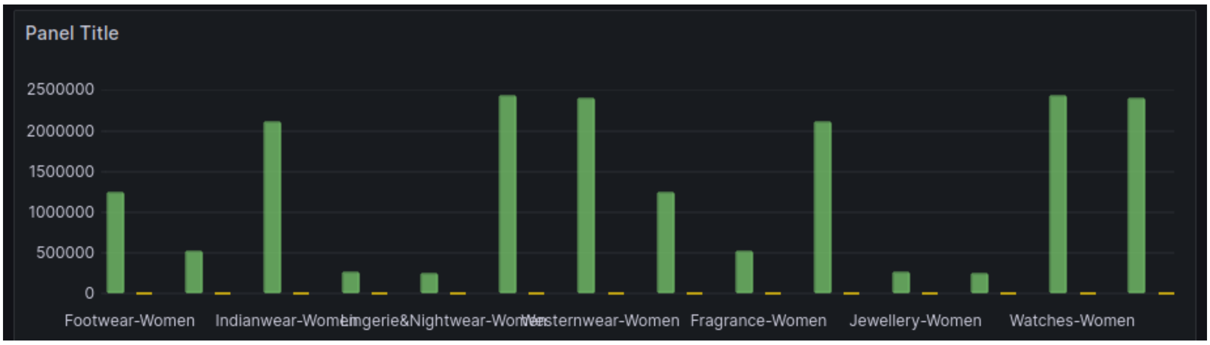


Figure 5.2: Flink Output

Job Execution

This line triggers the execution of the Flink job as shown below:

```
env.execute("Product Sales Analysis");
```

Listing 5.7: Output to CSV File

5.1.2 Processing Time

The custom Throughput class measures the performance of the data processing operation, potentially tracking metrics called processed time or run time, which is vital for benchmarking and optimizing the performance of the Flink application.

The Processed time was 1831 ms.

5.2 Apache Storm

5.2.1 Processing Workflow in Apache Storm

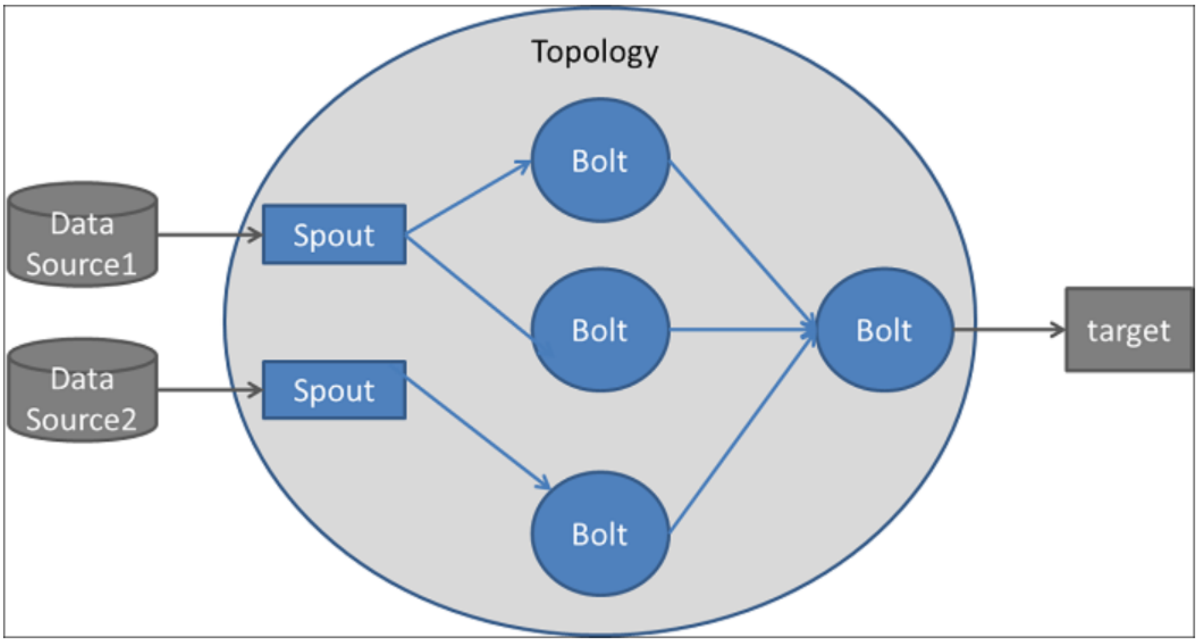


Figure 5.3: Apache Storm Analysis Architecture

5.2.2 Code Functionality Overview

The code is broken down into the following:

1. Spout: This is responsible for reading the input data from a CSV file.
2. Bolts: This is used to handle the processing
3. Topology Setup: This sets up the Storm topology, linking the spout and bolts and configuring how data flows between them.

CSVFileSpout (Data Ingestion)

This Spout reads each line of the product dataset from the CSV file nto fields and emits them as tuples into the Storm topology.It utilizes a BufferedReader to read the file and emits tuples corresponding to the product data. A snippet is show below:

```
public CSVFileSpout(String fileName) {
    this.fileName = fileName;
}

@Override
public void open(Map conf, TopologyContext context,
SpoutOutputCollector collector) {
    this.collector = collector;
    try {
        this.reader = new BufferedReader(new FileReader(fileName));
        reader.readLine();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Listing 5.8: CSV File Spout

AggregationBolt (Data Processing)

The Bolt receives product data tuples from the csvfilespot. It aggregates sales data by category, computing total sales and the number of products sold in each category and then emits aggregated data for further processing or storage. It Uses a HashMap to store and update sales totals and product counts for each category. Below is a snippet:

```
public class AggregationBolt extends BaseRichBolt {
 private OutputCollector collector;
 private Map<String, ProductAggregation> aggregations;
 @Override
 public void prepare(Map stormConf, TopologyContext context,
OutputCollector collector) {
     this.collector = collector;
     this.aggregations = new HashMap<>();
 }
 @Override
```

```
    public void execute(Tuple tuple) {
        System.out.println("Received tuple fields: " + tuple.getFields());
        String category = tuple.getStringByField("Category");
        String productName = tuple.getStringByField("Product_Name");
        double salesAmount = tuple.getIntegerByField("SellPrice");

        ProductAggregation aggregation = aggregations.getOrDefault(category, new ProductAggregation(category));
        aggregation.addSale(salesAmount);
        aggregation.addProduct(productName);
        aggregations.put(category, aggregation);
```

Listing 5.9: Aggregation Bolt

ThroughputBolt (Performance Measurement)

This bolt is used to measure the processed time of the topology. It is useful for monitoring and optimizing the performance of the Storm topology.

CSVWriterBolt (Data Output)

The bolt receives the aggregated data from the aggregation bolt and writes the results into an output CSV file, which was sent to Grafana for reporting. A snippet is below:

```
public class CSVWriterBolt extends BaseRichBolt {
    private transient BufferedWriter writer;

    @Override
    public void prepare(Map<String, Object> topoConf, TopologyContext context, OutputCollector collector) {
        try {
            // Open the file writer
            writer = new BufferedWriter(new FileWriter("/home/pce/Music/streambench/StreamBenchmarks-master/Storm/StormStock/output/output.csv", false)); // Set to 'false' to overwrite
            // Write the CSV header if needed
```

```
            writer.write("Category,TotalSales,ProductCount");
            writer.newLine();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Listing 5.10: CSV Writer Bolt

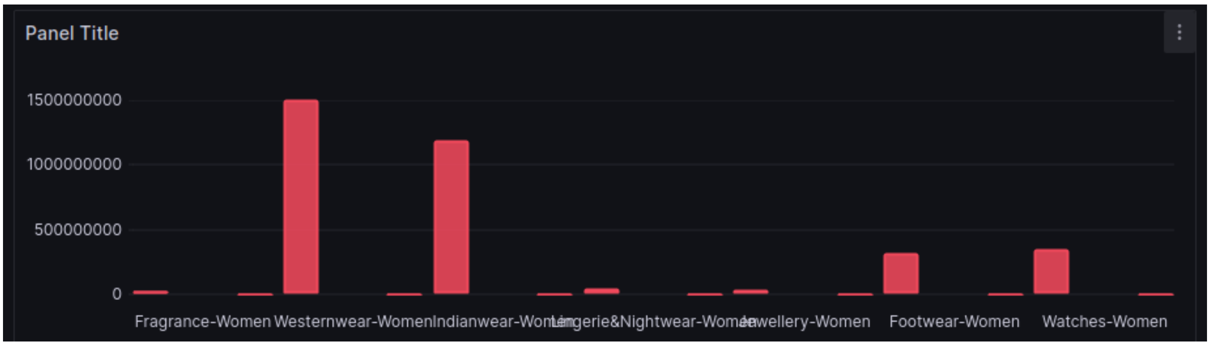


Figure 5.4: Storm Output

MyTopology (Topology Configuration)

MyTopology defines the structure of the Storm topology and set up the spout, bolts and their data flow. This is show below:

```
public class MyTopology {
    public static void main(String[] args) throws Exception {
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("csv-spout", (IRichSpout) new CSVFileSpout("/
home/pce/Music/StockAnalysis/DataSets/Product.csv"));
        builder.setBolt("aggregation-bolt", (IRichBolt) new
AggregationBolt()).shuffleGrouping("csv-spout");
        builder.setBolt("throughput-bolt", new ThroughputBolt()).
shuffleGrouping("aggregation-bolt");
        builder.setBolt("csv-writer-bolt", new CSVWriterBolt()).
globalGrouping("aggregation-bolt");

        Config config = new Config();
        config.setNumWorkers(1); // Number of worker processes
        LocalCluster cluster = new LocalCluster();
```

```
        cluster.submitTopology("csv-processing-topology", config,
    builder.createTopology());
    }
}
```

Listing 5.11: Topology

5.2.3 Processing Time

The Throughput bolt measures the performance of the data processing operation, potentially tracking metrics called processed time or run time, which is vital for benchmarking. **The Processed time was 3000 ms.**

5.3 Disscusion

From the processing time of how both system processed and aggregate the data, the results shows that Apache FLink performed better than Apache Storm. This suggest that apache flink is more efficient for fast stream of data.

6 Streaming vs. Relational Databases

This chapter aims to provide a detailed comparison between streaming databases and relational databases, focusing on their unique features, strengths, and weaknesses. To conduct this analysis, Apache JMeter, a popular testing tool, will be utilized to evaluate key performance metrics and operational characteristics of both types of databases.

Apache Flink, Apache Storm and Apache Kafka emerges as a significant technology, specifically designed for handling real-time data streams. They specializes in processing large volumes of data with low latency, making it ideal for real-time data streaming scenarios.

Contrasting this with PostgreSQL, a traditional relational database management system (RDBMS), we observe fundamental differences in design and functionality. PostgreSQL excels in structured data storage and complex query processing. It is not inherently built for real-time data ingestion or streaming, as it lacks native capabilities or specific extensions for processing data as it is generated in real time.

An interesting development in the realm of PostgreSQL for streaming data was PipelineDB, an extension to PostgreSQL that aimed to enable streaming capabilities. PipelineDB allowed for continuous SQL queries on streaming data, turning real-time streams into continuously updated SQL tables. However, PipelineDB [1] is no longer in active development or in use, which leaves a gap in PostgreSQL's ability to handle streaming data natively.

[1] docs.pipelinedb.com/

Despite these differences, comparing PostgreSQL's data handling capabilities (particularly data insertion and selection operations) with streaming data operations (Producing and Consuming) can be insightful:

PostgreSQL Insertion and Selection Operations: These are crucial for assessing how efficiently PostgreSQL can handle storage and retrieval of data. While PostgreSQL is adept at managing these operations, it typically does so with a focus on transactional integrity and query complexity rather than real-time processing.

Streaming Operations: Streaming database are designed for high-throughput, low-latency processing of data streams. They excels at handling continuous data flows, offering features like event time processing, windowing, and state management, which are essential for real-time analytics and processing.

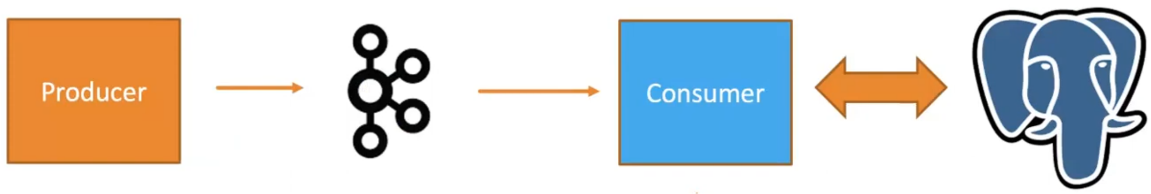


Figure 6.1: The Architecture of Use Case

By comparing these systems, we can understand the distinct roles they play in data architectures.

6.0.1 Using Apache Jmeter

To test the database, you need to

- Create a test plan
- Execute the test plan

The test plan used in this testing consists of the following elements:

- Thread Group
- Pepper-Box PlainText Config
- PepperBoxKafkaSampler in Java Request
- JSR223 Sampler
- BeanShell Sampler
- JDBC Connection Configuration
- JDBC Request
- JDBC Listener

JMeter has a default test plan, which you can use if you don't want to create multiple test plans. However, we need to add the missing elements in the default test plan.

ProstgreSQL is the database for this testing. We have to connect the database with JMeter using the Postgresql JDBC connector. First, download the postgresql42.7.1.jar from the JDBC postgres website[2]. Copy "postgresql42.7.1.jar" file (Version may differ) from the folder and paste it into the "lib" folder of JMeter and Restart the JMeter so that it will work with the library provided by Postgres connector.

Thread Group

The Thread Group is a set of threads that performs a test scenario. In this screen, we can set the number of users and other similar settings to simulate the user requests. We right-click on the TestPlan and then we select the Add->Threads (Users)->Thread Group

Pepper-Box PlainText Config

Pepper-Box PlainText Config is a JMeter configuration element used to generate test messages for Kafka. It allows you to define a template for the messages you want to produce.

[2]https://jdbc.postgresql.org/download/postgresql42.7.1.jar

These templates can include static text, JMeter variable references, and functions to generate dynamic content. To use it in JMeter test plan, right-click on the Test Plan or Thread Group. Go to Add > Config Element and select Pepper-Box PlainText Config.

PepperBoxKafkaSampler in Java Request

The com.gslab.pepper.sampler.PepperBoxKafkaSampler is a class from the Pepper-Box plugin for JMeter, designed to load test Apache Kafka. It is used as a Kafka producer that can send messages to a Kafka topic called university, based on a message template found in Pepper-Box PlainText Config. To add it, right-click on your Thread Group, go to Add > Sampler > Java Request. Then, select the Java Request sampler. In the Java Request sampler, for the Classname field, input com.gslab.pepper.sampler.PepperBoxKafkaSampler.

JSR223 Sampler

The JSR223 Sampler in this test is designed to consume messages from a Kafka topic using the Apache Kafka Consumer API in Groovy. To add this, Navigate to Add > Sampler > JSR223 Sampler.

Beanshell Sampler

The Beanshell Sampler in JMeter is a scripting tool that allows you to write custom scripts to extend the functionality of your JMeter tests. In our test plan, it get values from the Kafka message in JSR223 Sampler and then use the extracted values in other parts of the JMeter test. Add Beanshell Sampler to Your Test Plan by Right-clicking on the Thread Group in your JMeter test plan. Abd then, navigate to Add > Sampler > Beanshell Sampler

JDBC Connection Configuration

This is used to create a valid database connection. You add JDBC Connection Configuration by following the steps : "Add > Config Element > JDBC Connection Configuration"
Then configure your PostgreSQL Database with this:
Database URL: jdbc:postgresql://localhost:5432/name_of_database
Database Class: org.postgresql.Driver
Username: postgres
password: Your Password

JDBC Request

JDBC request element helps to define a SQL query that will be executed by the test user(s). To work with data from the database, You need to Add JDBC Request. So add JDBC Request By Clicking Add > Sampler > JDBC Request we set the Variable Name of Pool declared in JDBC Connection Configuration parameter. This parameter value has to be same as the JDBC connection pool name

JDBC Listener

Before starting our test, we need a Listener that helps to monitor and analyze the result of the test. For this test, we will use the View Results Tree, of listeners. We right-click on the JDBC Request and select Add->Listener->View Results Tree to monitor the detailed result of the test. At the same time, we add a Summary Report that helps to monitor a summarized result of the test. To add a summary report, we right-click on the JDBC Request and select Add->Listener->Summary Report to monitor the detailed result of the test.
Test your database performance by clicking the **Run button on the menu bar**.

6.1 Performance Test

The following table showcases the results derived from Apache JMeter 'View Results in Table' report, providing a detailed overview of the performance metrics observed during our recent testing procedure:

Sample #	Start Time	Thread Name	Label	Sample Time(ms) ↓	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
2	03:40:29.975	Kafka_Posgre...	JSR223 Sampler	84		0	0	0	0
350	03:40:31.172	Kafka_Posgre...	JDBC Request	81		9	0	81	80
722	03:40:42.961	Kafka_Posgre...	JDBC Request	75		9	0	75	73
343	03:40:31.087	Kafka_Posgre...	JSR223 Sampler	72		0	0	0	0
712	03:40:42.088	Kafka_Posgre...	JDBC Request	70		9	0	70	69
679	03:40:31.882	Kafka_Posgre...	JSR223 Sampler	63		0	0	0	0
2144	03:46:48.400	Kafka_Posgre...	JDBC Request	62		9	0	62	60
684	03:40:31.893	Kafka_Posgre...	JSR223 Sampler	61		0	0	0	0
717	03:40:42.421	Kafka_Posgre...	JDBC Request	61		9	0	61	60
2499	03:48:45.566	Kafka_Posgre...	JDBC Request	60		9	0	60	58
727	03:40:43.420	Kafka_Posgre...	JDBC Request	56		9	0	56	55
168	03:40:30.529	Kafka_Posgre...	JSR223 Sampler	55		0	0	0	0
509	03:40:31.551	Kafka_Posgre...	JSR223 Sampler	54		0	0	0	0
514	03:40:31.556	Kafka_Posgre...	JSR223 Sampler	54		0	0	0	0
333	03:40:31.026	Kafka_Posgre...	JSR223 Sampler	53		0	0	0	0
1930	03:45:47.323	Kafka_Posgre...	JDBC Request	53		9	0	53	51
318	03:40:30.946	Kafka_Posgre...	JSR223 Sampler	51		0	0	0	0
328	03:40:30.999	Kafka_Posgre...	JSR223 Sampler	51		0	0	0	0
1201	03:42:36.063	Kafka_Posgre...	JDBC Request	51		9	0	51	50
639	03:40:31.770	Kafka_Posgre...	JSR223 Sampler	50		0	0	0	0
981	03:41:43.570	Kafka_Posgre...	JDBC Request	50		9	0	50	48
2319	03:47:47.464	Kafka_Posgre...	JDBC Request	50		9	0	50	49
4	03:40:30.061	Kafka_Posgre...	JDBC Request	49		9	0	49	48
2129	03:46:45.394	Kafka_Posgre...	JDBC Request	49		9	0	49	48
313	03:40:30.940	Kafka_Posgre...	JSR223 Sampler	48		0	0	0	0
488	03:40:31.493	Kafka_Posgre...	JSR223 Sampler	48		0	0	0	0
634	03:40:31.763	Kafka_Posgre...	JSR223 Sampler	48		0	0	0	0

Figure 6.2: Latency Result

Based on the information presented in the diagram, it is evident that the PostgreSQL operations, specifically data insertion and selection performed via JDBC Requests, exhibited higher latency compared to the Kafka Producer and Consumer operations, which were executed through Java Requests and JSR223 Samplers. Furthermore, it's notable that the Kafka Consumer operation, in particular, demonstrated a longer load time than both the data insertion and selection processes in PostgreSQL. This could be indicative of delays in processing or fetching data from Kafka.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Java Request	500	0	0	3	0.34	0.00%	1.0/sec	0.14	0.00	137.9
JSR223 Sam...	500	7070	0	10027	4358.64	43.40%	1.0/sec	0.00	0.00	.0
BeanShell S...	500	0	0	3	0.50	0.00%	1.0/sec	0.00	0.00	.0
JDBC Request	1000	3	0	81	8.69	0.00%	2.0/sec	20.86	0.00	10585.0
TOTAL	2500	1415	0	10027	3434.27	8.68%	5.0/sec	20.99	0.00	4261.6

Figure 6.3: Throughput Result

Based on the observed results depicted in the diagram, we can draw several insights regarding the performance characteristics of PostgreSQL in relation to Kafka, particularly in the context of handling streaming data. The throughput results indicate that PostgreSQL, both in its data insertion and selection operations, demonstrated a throughput performance comparable to that of Kafka's producer and consumer processes.

This finding suggests that while PostgreSQL may not inherently possess capabilities for real-time data streaming or consuming data directly from a real-time source, it is nonetheless capable of effectively storing streaming data once it has been processed and delivered by a streaming consumer. The simulation's results thus highlight PostgreSQL's robustness and efficiency in managing database, affirming its suitability for scenarios where streaming data, after being consumed and potentially processed by a tool like Kafka, needs to be stored reliably and accessed efficiently.

6.2 Comparative Analysis

The figure below shows a comparative analysis between a streaming database and a relational database:

Table 6.1: Detailed Comparison between Streaming Databases and Relational Databases

Feature	**Streaming Database**	**Relational Database (e.g., PostgreSQL)**
Primary Function	Designed for continuous ingestion, processing, and analyzing of streaming data in real-time. Capable of handling large volumes of data with minimal delay, making them ideal for applications that require immediate responses to data inputs.	Focused on the storage, retrieval, and management of structured data. Provides robust capabilities for complex querying, data consistency, and transactional integrity, suitable for applications where data relationships and integrity are of utmost importance.

Continued on next page

Table 6.1 – *Continued from previous page*

Feature	**Streaming Database**	**Relational Database (e.g., PostgreSQL)**
Data Model	Oriented around unbounded streams of data, typically handling data in a continuous flow. Data is processed in real-time, or near-real-time, often without being persisted.	Utilizes a structured, table-based data model, with well-defined schemas comprising rows and columns. Data is stored persistently, allowing for complex relationships between different entities and tables.
Processing Model	Excels in stateless and stateful stream processing, with a focus on event-driven architectures. Ideal for scenarios requiring real-time decision-making based on live data streams.	Employs a transactional model, ensuring data consistency and integrity using the ACID properties. Supports complex transactions, making it suitable for systems where data accuracy and consistency are critical.
Use Cases	Commonly used in scenarios like real-time analytics, monitoring systems, IoT data processing, fraud detection, and other applications where immediate data processing is essential.	Widely used in enterprise applications, online transaction processing (OLTP), customer relationship management (CRM), enterprise resource planning (ERP), and reporting and data analysis systems.
Query Capabilities	Offers capabilities for time-window based queries, aggregations over streams, and real-time data processing. Query language might be SQL-like or a proprietary DSL.	Provides extensive SQL support for complex queries, including joins, subqueries, aggregations, and window functions. Ideal for applications requiring comprehensive data analysis and reporting.

Continued on next page

Table 6.1 – *Continued from previous page*

Feature	**Streaming Database**	**Relational Database (e.g., PostgreSQL)**
Data Integrity	Often designed with a focus on event time processing and timeliness of data. While some systems ensure exactly-once processing semantics, traditional transactional integrity is less emphasized.	Ensures strict data integrity through ACID compliance, reliably managing transactions and maintaining the consistency and accuracy of data, even across complex operations.
Scalability	Typically highly scalable, especially horizontally, efficiently managing high-throughput scenarios and processing large volumes of data quickly.	Can scale to handle large datasets and high transaction volumes. Traditional relational databases often rely more on vertical scaling, though horizontal scaling solutions exist.
Performance Metrics	Evaluated based on throughput (events processed per second) and latency in processing each event. High throughput and low latency are essential for effective real-time data processing.	Performance metrics include query response time, transaction processing speed, and the ability to handle concurrent operations efficiently.
Strengths	Highly efficient in processing high volumes of data in real-time, providing capabilities for rapid decision-making based on live data.	Robust and reliable for structured data storage, complex data querying, and maintaining data integrity and relationships.

Continued on next page

Table 6.1 – *Continued from previous page*

Feature	**Streaming Database**	**Relational Database (e.g., PostgreSQL)**
Weaknesses	Typically less suited for complex transactional operations, historical data querying, and situations requiring long-term data persistence.	Not designed for real-time data ingestion or streaming, potentially leading to challenges in scenarios requiring immediate data processing.

7 Conclusion

In this project, we have undertaken a comprehensive exploration of the fundamental theories and concepts associated with two prominent streaming databases: Apache Flink and Apache Storm. Our journey included the development and implementation of a suite of benchmark applications tailored for stream processing systems, including Fraud Detection, Spike Detection, Word Count, Yahoo! Streaming Benchmark, and VoipStream.

The core of our research involved conducting a series of rigorous experiments, focusing on four critical metrics: throughput, latency, CPU utilization, and memory utilization. A comparative analysis between Apache Flink and Apache Storm formed a significant part of our study. Through extensive testing in various scenarios—centered on throughput, latency, and CPU utilization—we established that Apache Flink consistently outperformed Apache Storm. However, it's worth noting that Flink exhibited higher memory utilization compared to Storm. This distinction is critical for understanding the trade-offs between these systems in resource-intensive environments.

The experiments were supplemented with data visualization techniques, utilizing figures and charts to present and analyze the results in an accessible and informative manner.

To further our comparative study, we executed a Streaming Database Use Case using a product sales dataset, which provided additional empirical evidence of Apache Flink's superior performance over Apache Storm in our testing scenarios.

Expanding our research scope, we compared streaming databases with a traditional database system, Postgres, using Apache JMeter. Here, we are compared PostgreSQL's data handling capabilities (particularly data insertion and selection operations) with streaming data operations (Producing and Consuming). The test showed that postgreSQL is capable of effectively storing streaming data once it has been processed and delivered by a streaming consumer.

References

[Ali21] Alizada, Mansur (2021). “Real time vs micro-batching in streaming data processing: performance and guidelines”. In.

[Bor+20] Bordin, Maycon Viana et al. (2020). “DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems”. In: *IEEE Access* 8, pp. 222900–222917. DOI: 10.1109/ACCESS.2020.3043948.

[Cha10] Chatterjee, Shreeshankar (2010). “Modeling, debugging, and tuning QoE issues in live stream-based applications-A case study with VoIP”. In: *2010 Seventh International Conference on Information Technology: New Generations*. IEEE, pp. 1044–1050.

[CCM12] Chauhan, Jagmohan, Shaiful Alam Chowdhury, and Dwight Makaroff (2012). “Performance evaluation of Yahoo! S4: A first look”. In: *2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. IEEE, pp. 58–65.

[Che+03] Cherniack, Mitch et al. (2003). “Scalable Distributed Stream Processing.” In: *CIDR*. Vol. 3. Citeseer, pp. 257–268.

[Clo23] CloudDuggu (2023). “Apache Storm Architecture”. In: URL: https://www.cloudduggu.com/storm/architecture/.

[Fli23a] Flink, Apache (2023a). “Apache Flink CEP.” In: URL: https://ci.apache.org/projects/flink/flink-docs-release-1.18/docs/libs/cep/.

[Fli23b] Flink, Apache (2023b). "Apache Flink Dataset." In: URL: https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/dev/dataset/overview/.

[Fli23c] — (2023c). "Apache Flink Documentation." In: URL: https://nightlies.apache.org/flink/flink-docs-release-1.18/.

[Fli23d] — (2023d). "Apache Flink Documentation." In: URL: https://nightlies.apache.org/flink/flink-docs-release-1.18/.

[Fli23e] — (2023e). "Apache Flink Table". In: URL: https://ci.apache.org/projects/flink/flink-docs-release-1.18/docs/dev/table/overview/.

[Glu23] Glushach, Roman (2023). "Flink, Spark, Storm, Kafka: A Comparative Analysis of Big Data Stream Processing Frameworks." In: URL: https://romanglushach.medium.com/flink-spark-storm-kafka-a-comparative-analysis-of-big-data-stream-processing-frameworks-for-dab3dd42fc16/.

[Kar+18a] Karimov, Jeyhun et al. (2018a). "Benchmarking distributed stream data processing systems". In: *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, pp. 1507–1518.

[Kar+18b] — (2018b). "Benchmarking distributed stream data processing systems". In: *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, pp. 1507–1518.

[KDA19] Kolajo, T., O. Daramola, and Adebiyi (2019). "A Big data stream analysis: a systematic literature review." In: *J Big Data 6, 47*. DOI: 10.1186/s40537-019-0210-7.

[Kra20] Krasňan, Be. Peter (2020). "Benchmarking Big Data Streaming Platforms". In.

[Lu+14] Lu, Ruirui et al. (2014). "Stream bench: Towards benchmarking modern distributed stream computing frameworks". In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE, pp. 69–78.

[Muk+10] Mukherjee, Aniruddha et al. (2010). "Capital market surveillance using stream processing". In: *2010 2nd International Conference on Computer Technology and Development*. IEEE, pp. 577–582.

[Phu+07] Phua, Clifton et al. (2007). "Adaptive spike detection for resilient data stream mining". In: *Proceedings of the sixth Australasian conference on Data mining and analytics-Volume 70*, pp. 181–188.

[RB16] Rajeshwari, U and B Sathish Babu (2016). “Real-time credit card fraud detection using Streaming Analytics”. In: *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*. IEEE, pp. 439–444.

[Sto23] Storm, Apache (2023). “Why use Apache Storm?” In: URL: `https://storm.apache.org/`.

[Yan17] Yang, Shusen (2017). “IoT stream processing and analytics in the fog”. In: *IEEE Communications Magazine* 55.8, pp. 21–27.

[ZZM18] Zhang, Ruinan, Fanglan Zheng, and Wei Min (2018). “Sequential behavioral data processing using deep learning and the Markov transition field in online fraud detection”. In: *arXiv preprint arXiv:1808.05329*.