Investigation about stream processing frameworks

ChengLitao

Aug 19th, 2022

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2022

Abstract

Big data processing can extract insight and value from massive data. The characteristics of rapid generation and low-value density of big data pose challenges to data processing methods. An ideal data processing system should have both high throughput and low response time . Streaming and batch processing are two common ways of processing big data. In some systems requiring fast response, stream processing has advantages over batch processing [1]. Stream and micro-batch processing can be implemented for infinitely generated unbounded data streams. Stream processing frameworks can help to quickly develop unbounded data processing applications, such as Spark, Flink, and Strom. Stream processing frameworks can be used for business analytics, IoT and social media analytics [2]. Due to the widespread use of stream processing, we really wanted to know the deployment details and applicable scenarios of different stream processing frameworks. To explore performance under different data pressures, we need to measure performance using stream processing benchmarking tools.

In this project, we summarize the stream processing architecture and deployment methods. Design ideas for four different open source stream processing benchmarks are discussed, and their strengths and weaknesses are evaluated. We propose a comprehensive metric for judging whether a stream processing application is stable. We explore Spark, Flink throughput, latency, and scalability on a local virtual machine cluster using four workloads from HiBench. The results show that, under the experimental conditions, compared with Spark and Flink, Spark's Maximum Stable Throughput(MST) dominates, while Flink's response time is significantly faster. When the data pressure is close to critical, Spark's MST is about twice that of Flink, and Flink's latency is less than 0.1 times that of Spark. Flink is better at stateful operations, especially window operations. Then the CPU-intensive applications are more scalable than IO-intensive applications. Finally, some performance tuning operations are also verified to be effective. Overall, Flink is more convenient to develop and has an absolute advantage in latency performance. Therefore, when throughput is not the main requirement, it is recommended to use Flink to develop stream processing applications. What is more, stream processing applications on HPC clusters are also a possible future exploration direction.

Contents

List of Tables

List of Figures

Acknowledgements

Thanks to my supervisor Dr Anna Roubickova. I was impressed with her patience and enthusiasm. Although I was not assigned any topic at the beginning of the semester, she encouraged me to propose my project actively. She always gives professional engineering advice and encouragement when the project is in trouble. Without her help, the project would not have been started and carried out thoroughly.

Thanks to Dr Amy Krause, one of EPCC's Principal Architects. Her professional advice on high-performance clusters helped me steer the project. Without her enthusiastic help, my project would not have been executed smoothly.

Thanks to Alexandre Dumas, a prestigious writer. His sentence, "All human wisdom is summed up in two words --' Wait and hope.'", lit me up in many sleepless nights during the project.

Chapter 1 Introduction

Big data analysis has become an indispensable tool in business, finance, the Internet of Things and other fields. Big data is usually produced quickly, in huge quantities, and various forms. A continuously generated stream of data is one possible form. Here are some examples about stream data processing: 1.The sensors in the Internet of Things continuously detect the weather outdoors, many sensors are deployed in a particular area, and scientists want to analyze the local air movement patterns. 2. On social media, millions of people continue to voice their opinions in the community, and research institutions want to understand the sentiment in the crowd. 3. In the bank's online transaction system, a large amount of transaction data is generated every day. The bank needs to detect the abnormal behaviour of the transaction and promptly remind the customer of the risk. In these examples, the performance of stream processing applications usually has business-driven solid requirements. More generally, stream processing applications can be divided into Event-driven applications, Data analysis applications, and Data pipeline applications [3].

The pursuit of performance in stream processing applications is consistent with HPC major. Stream processing framework deployment and implementation is attrative knowledge for me. At the same time, business-driven requirements have also prompted us to investigate areas related to stream processing performance, including benchmarking and performance tuning. In this project, we aim to compare the ease of use and performance metrics of Spark and Flink in development and evaluate the two stream processing frameworks through benchmarking. Further, we initially explore the potential applications of stream processing on high-performance clusters.

In the face of continuous data flow, we need general tools to process data while meeting latency, throughput and horizontal scalability requirements. Such development tools that implement various stream processing applications are called stream processing frameworks. Standard stream processing frameworks include Spark, Flink, and Strom. Among them, Spark uses micro-batch to realize stream data processing; Flink is mainly designed for stream processing and realizes the unification of stream and batch. For performance testing of processing applications, we need to use benchmarking tools. Benchmarking tools provide artificial data, workloads, and performance test systems. We surveyed four widely used stream processing benchmarks, summarizing their strengths, weaknesses, and challenges in benchmark design.

For the experimental design, we selected HiBench among the four stream processing benchmarks, which provides a richer workload [4]. We explored Spark and Flink throughput, latency, and scalability. In experiments, we build a qualitative model of the

response of stream processing performance to data pressure, distinguishing between steady and non-steady states of the system. A comprehensive index is proposed to determine the state of the system. After finding the maximum data pressure acceptable to the system, we explored the effect of stream processing framework configuration on performance at a steady state.

The report's structure is arranged as follows: Chapter 2 mainly describes the project's background. Sections 2.11-2.16 describe the computational model behind stream processing applications, the deployment environment, and the internal hierarchical structure. Section 2.1.7 discusses stream processing applications' performance measures and influencing factors; Section 2.2 gives an overview of the four open sources Stream Processing Benchmark Tool. In Chapter 3, we summarize the components used in stream processing applications and how they work, and in Section 3.5, we summarize performance tuning recommendations. Chapter 4 describes performance assumptions for workloads and stream processing applications. In Chapter 5, the performance of three aspects of stream processing was explored using the HiBench benchmark, and the performance results were visualized. Finally, in Chapter 6, we summarize the progress and limitations of the project, as well as follow-up research directions.

Chapter 2 Background and literature review

The project's background and related research will be synthesized and reviewed in this chapter. In Section 2.1, we examine the real-world application and deployment details of stream processing from the perspective of big data processing. Additionally, we summarize the relevant literature and experimental strategies. Then, in Section 2.2, we present the general features and design challenges of the Stream Processing Benchmark and survey four specific open-source benchmarking projects.

2.1 Background

This section introduces the components and terminology involved in stream processing. As a big data processing paradigm, stream processing implements one record at a time at the processing granularity, which is a crucial difference compared to batch processing. The deployment and architecture of stream processing are also mentioned, reflecting the distributed design of stream processing frameworks.

2.1.1 Big Data processing

In recent years, with the rapid development of the information industry, the speed of data generation far exceeds the speed of processing[5]. These vast volumes and diverse data sources are difficult to process with traditional computing architectures, so they are called Big Data. Big data usually has 5Vs characteristics[6]:

1. Volume, a massive amount of data, usually reaches GB or TB.

2. Variety, which means the diversity of data sources and types, which usually comes from the complexity of the business, joint includes semi-structured and unstructured data.

3. Value, indicating that the information density of the data is very low, and it is necessary to use appropriate data mining methods in a large amount of data to obtain business-related value.

4. Velocity indicates that the data processing speed is breakneck, and the timeliness of the processing process is relatively high.

5. Veracity shows the importance of data from real business to the analysis results.

Big data often contains valuable information. As a result, Big Data processing is widely used in healthcare, transportation, finance, and the Internet. Big data processing technologies have emerged, including cluster computing, distributed storage and more. As shown in Figure xx, an extensive data processing application's processing process can be divided into different parts. The receiver is responsible for receiving data from the data source. The data is sent to the subsequent task work for processing to obtain valuable information. The valuable information is output by the sink worker to the external storage system.

Big Data processing applications aim to achieve high throughput, low latency, high availability and scalability. This is what companies are looking for. In this business-driven context, Big data processing usually needs to run on a cluster, where multiple processing units work together to improve processing performance. The parallel computing paradigm and the corresponding data processing engine improve the utilization efficiency of computing resources and simplify the development of big data processing applications.

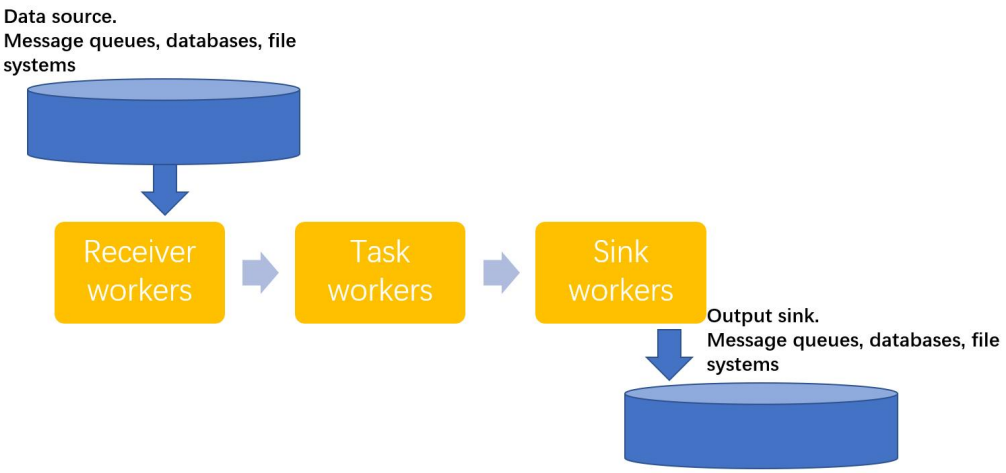


Figure 1: Flow chart of Big data processing

2.1.2 Stream and batch

Stream processing and **batch processing** are two typical paradigms of data processing. Therefore, both follow the flow chart shown in Figure. Therefore, the two processing paradigms are very close. The core difference between the two is that the granularity of the task worker (processing engine) work is different. The batch process processes fixed-size data each time and sends the result to downstream workers after processing a data block. Correspondingly, stream processing processes one record at a time, and the task is sent to the downstream operator (basic function unit) immediately after processing a record.

A single batch calculation usually processes a limited number of tasks, and the task work is automatically closed after processing a fixed amount of data. In contrast, in stream computing, the task work will continue and is responsible for continuously processing

the transmitted data stream. Batch computing and stream computing are very close and can be converted into each other under certain circumstances. Naturally, stream processing can complete a single batch of computing tasks. A small number of consecutive batch computations can simulate the effect of stream computing, the technique is called **micro-batch**.

In this context, the stream computing model implements all the functions of a single batch computing and provides the ability to process a variable amount of data. Therefore, Tyler Akidau [7] believes batch computing is a special kind of stream computing in theory. A well-designed stream processing system can completely replace the batch system, but it is limited to the current technology. When processing data with fixed volume, the current stream processing engine still has a gap in throughput compared with batch computing.

Stream processing pursues real-time data calculation, can query the intermediate results(called **state**) of calculation in real-time, and realize incremental calculation. It has advantages in system latency and focuses on the characteristics of the **Velocity** of Big Data. Typical stream processing frameworks include Flink and Storm. Batch processing pursues data throughput and can process large amounts of data quickly. It has advantages in the system's throughput and focuses on the characteristics of **Volumes** of Big Data. Typical batch processing frameworks include Hadoop and Spark. It is worth mentioning that, according to the level of latency, data processing can be divided into real-time computing and offline computing, which can be implemented in any computing paradigm (stream or batch).

Stream processing and batch processing usually use different data sources due to the difference in processing. Batch computing usually processes **bound data** with limited voloum, and the data source generally is the file system. Streaming computing usually processes **unbound data**, using message queues as the data source. The Figure2 shows how records are organized in batches and streams.

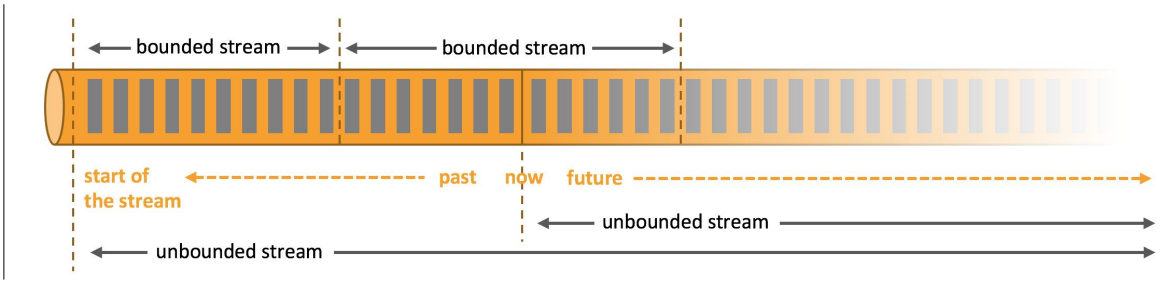


Figure 2: The illustration of stream and batch data.[8]

However, overemphasizing the underlying implementation of the data processing engine (concepts of streaming/batch) in term context may also be misleading. In modern stream processing, more complex applications include data with different characteristics simultaneously, and the performance requirements will also change. In application development and business, the data processing engine should focus more on the data's characteristics and the business's performance requirements. Therefore, Google

Research [9] advocates using the concepts of unbounded and bounded data instead of stream processing and batch processing because the latter implies the computational details of two different kinds of computing engines rather than focusing on the characteristics of the data itself. For unbounded/bounded data, computing models such as batch, micro-batch and stream processing can all provide the same level of accuracy. The processing engine only needs to make choices in the underlying implementation (streaming/batch) according to business needs. This idea also called stream-batch unification. Therefore, the trade-off between stream/batch unification and performance has been the subject of advanced data processing development.

2.1.3 Distributed parallel computing model

With the rapid growth of computing demand, single computing unit has been challenging to meet. Distributed parallel computing is a common way to achieve high performance. High performance means high throughput, low latency, and scalability. Distributed, relative to centralized computing means that computing needs to coordinate multiple independent, network-connected computing systems to work together. Parallel computing, relative to serial computing, refers to the use of computing resources to solve multiple tasks at the same time. Usually, the processing units of parallel computing will share a memory, and the processing units will simultaneously create multiple threads to solve tasks concurrently. Technically, the main difficulty of distributed computing lies in the non-determinism of the computing environment, which requires the management of heterogeneous computing architectures and unstable resource availability. In addition, task decomposition and resource scheduling are also important links to improving distributed computing.

Computational models are proposed to overcome barriers in distributed computing. Common computing models include the Mapreduce batch model and Google dataflow model. Mapreduce [10] is the earliest batch computing model and is used by Google for sorting large amounts of data. The model consists of four stages of computation, using ideas from functional programming. In the Split stage, the input file is split into fixed-size batches. During the Map phase, the data is converted into key-value pairs. The Shuffle phase sorts, transmit, and partitions the mapped data. Data located in the same partition is written to disk, resulting in a file in which the data is arranged in an orderly manner. Finally, the reduce function merges the sequences in the same partition according to the critical value and writes the result to the distributed file system.

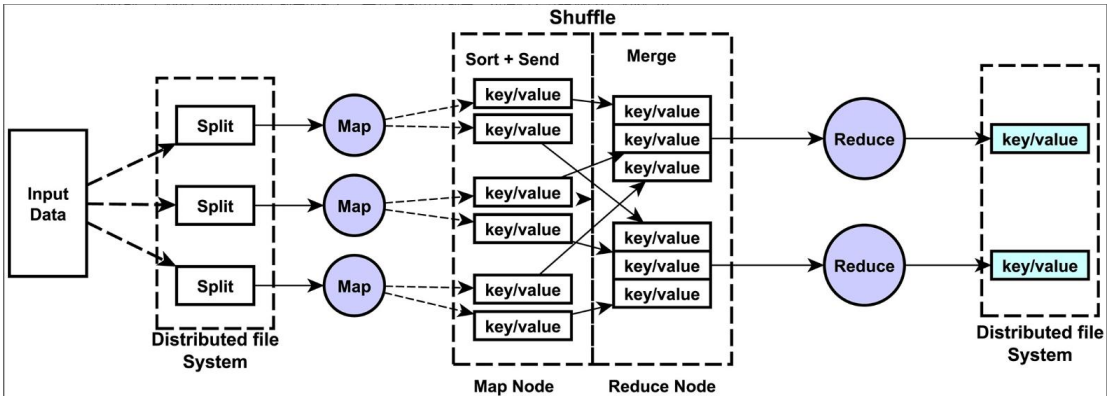


Figure 3:Mapreduce architecture [11]

Google dataflow [9] is another computing model for stream processing. For concept, Dataflow distinguishes the difference between event time and processing time. The model also adds a variety of windows, such as scrolling windows, sliding windows, and session windows. In particular, the intermediate state of the computation can be managed and preserved. In practice, stream data is abstracted as immutable objects, and all operations on stream data are called operators. All complex applications can be abstracted as directed acyclic graphs with operators as vertices.

2.1.4 Streaming processing framework

Processing engine is some kind of computing model implementation. The earliest is Hadoop based on MapReduce, among which the processing engines that are widely recognized and applied in the industry include: Storm, Flink, and Spark. This project mainly examines the difference between the latter two. Stream processing frameworks usually have a layered API structure and higher-level libraries to implement complex application in specific area . The figure shows the hierarchical API structure of Flink.

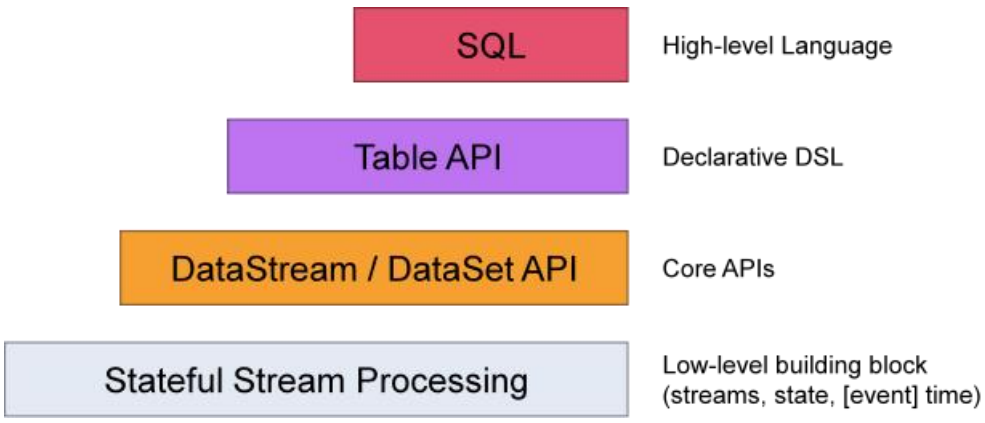


Figure 4: Layered API structure in Flink[12]

In addition to the stream processing framework mentioned above, with the development of big data processing technology, there are more than a dozen open source processing engines, including Apache Apex, Gearpump, tigon, Hazelcast Jet, Kuiper, Wallaroo.

These stream processing engines have similarities and have different emphasis on functions.

2.1.5 Computing clusters

The cluster is an environment in which distributed parallel computing is deployed. In order to make a group of independent computers work together, the concept of the cluster was proposed. A cluster is a group of computers that communicate with each other through a high-speed communication network (usually a local area network), and each computer in the cluster is called a node. Communication details are hidden when the cluster provides services to outside users, which can be regarded as a relatively independent system. In clusters, reliable transmission protocols are high likely to ensure secure transmission. A typical secure protocol is the Secure Sockets Layer (SSL) protocol in the transport layer, based on asymmetric encryption. One of the common cluster architectures has a controller/agent structure. That is to say, a controller node in the cluster is responsible for interacting with users and scheduling tasks, and multiple worker nodes are responsible for jointly performing specific computing tasks.

For the possible deployment of stream processing computing, we can deploy stream engine in the following four types of cluster environments: virtual machine, HPC cluster, cloud computing and container.

The virtual machine is a laptop that could make the cluster portable. Technically, the hypervisor is added to the physical layer and operating system through virtualization technology. One or more fully isolated computer systems are assembled by simulating fully functional hardware on a single node. Each virtual machine has a complete and independent operating system, storage system, computing unit and network card. Virtual machines can be installed, removed quickly, and backed up at any time. A personal notebook can virtualize multiple computers to form a cluster, and the maintenance cost is significantly reduced. Standard virtual machine software includes VMware.

A high-performance computing cluster is another choice. The main feature of the HPC cluster is its extreme computing speed, capable of completing teraflops of calculations per second. HPC clusters are often used for computing or massive data processing to solve scientific problems like weather forecasting and gene sequencing. In pursuit of peak performance, there are usually quite a few improvements and trade-offs in network, memory, and CPU architectures. In particular, a specific scientific computing library or compiler is configured on the high-performance computing cluster to accelerate the program's execution. Modern HPC architectures are often heterogeneous, with accelerators such as FPGAs and GPUs. Currently, available HPC clusters in this project are Archer2 and Cirrus.

Cloud computing is a computing paradigm that provides computing services to consumers on demand through a network. Technically, cloud computing providers centrally build and manage computing resources, taking advantage of economies of scale. Therefore, cloud computing customers only need to understand the implementation of the computing infrastructure to obtain the required services. Users can use computing

and storage resources elastically on the cloud computing platform and pay according to usage. The cloud platform provides stream processing frameworks and test platforms, including Google Dataflow and Amazon Kinesis.

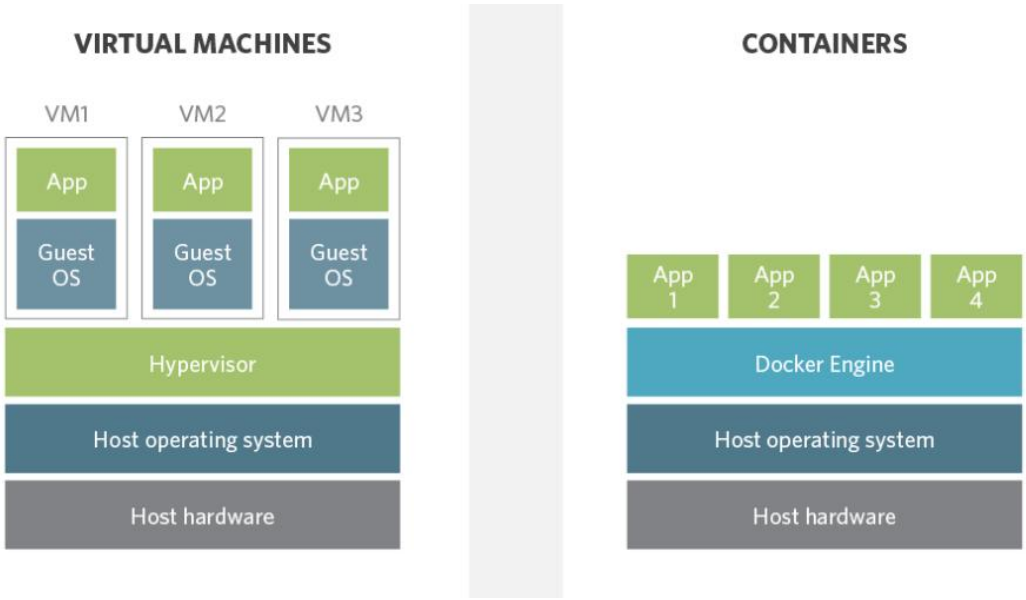


Figure 5: Architecture difference the virtual machine and the container[13]

Containers are lightweight OS virtualization; creating a container requires fewer resources than creating a virtual machine. The Figure5 illustrates the difference between the virtual machine and the containers. Processes in containers depend on static files called images, and stream processing engines can also be deployed as images. The container encapsulates the dependencies necessary to run the application, making the image migration in different environments more flexible, which indicates the container has better portability. The program runs in a resource isolation environment; that is to say, the containers are isolated from each other, and the upgrade of the container will not affect other containers. Standard tools for creating containers include Docker and Kubernetes.

2.1.6 Data processing architecture

In order to implement the flow chart of logical data processing mentioned in section 2.1.1, distributed big data processing usually requires modular deployment. At the software architecture level, as shown in the Figure6, the hierarchical structure includes four layers[14]. The lower layer provides the upper layer with necessary interfaces for interaction and is responsible for different functional requirements in Big Data processing.

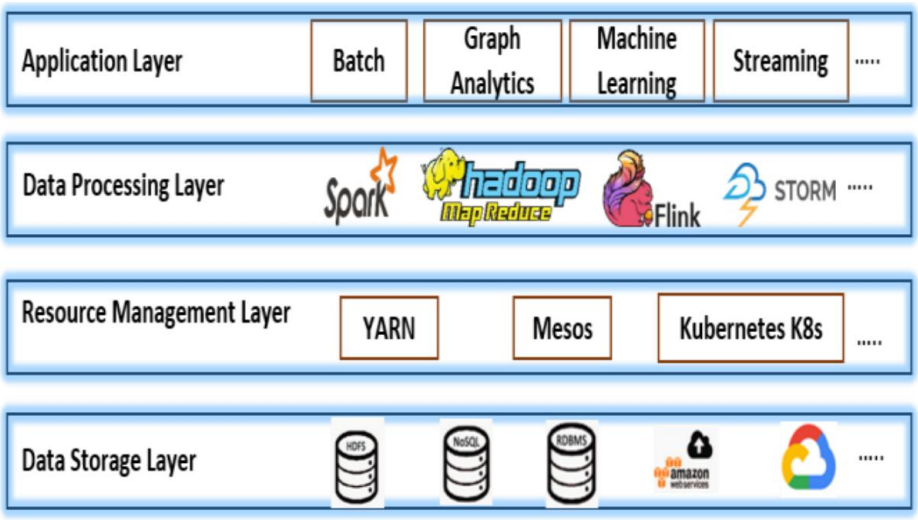


Figure 6: Layered computing architecture of Big Data processing[14]

The data storage layer is responsible for providing a variety of media for data storage. The data management layer manages the data and provides the appropriate data format. Standard data storage software in data processing includes HDFS (a distributed file system), Cloud storage, and NoSQL. Message queues can provide reliable data flow for stream processing, and standard message queues include Kafka. The resource management layer is responsible for allocating and scheduling computing resources. According to business needs, standard resource allocation software includes Yarn and k8s. The data processing layer and commonly used data processing platforms such as Hadoop, Spark, and Flink are responsible for data processing. The application layer implements business logic. The code is based on data processing receipts, and typical applications include streaming batch processing, machine learning, and graph analysis.

2.1.7 Performance of stream processing application

Stream processing performance metrics include throughput, lantency, CPU utilization, scalability, and even more. A stream processing system is a dynamic system that may run for a long time. Especially when dealing with unbounded data, the system must remain available for a long time. The performance of the system may fluctuate over a longer time interval. Some studies take this feature into account when designing indicators. Usually, some restrictions always need to be imposed on the index detection. Karimov et al. in 2018 [15] studied the metrics measurement of stream processing systems and analyzed the difference between latency, throughput, and standard program tests. They argue that latency in stream processing requires a distinction between event time and processing time. Furthermore, they are the first to propose the concept of **sustainable throughput**, defined in the paper as a system state that does not exhibit long-term backpressure (continuously increasing event time delay). Chu et al. [16] proposed a method to test throughput volatility based on a sliding window in 2020, which was tested to be more effective than traditional methods to evaluate the system's maximum throughput. They used the maximum stable throughput MST to evaluate the maximum intake capacity of the system, which is defined as the maximum throughput

that does not cause system throughput volatility. Two systems, Flink and Spark, are evaluated in Karakaya [17]. In the evaluation, a latency-limited stream processing system was used. Throughput was measured under three different scenarios: sustainable peak throughput under constant load, burst throughput at startup, and periodic burst throughput.

From Figure 7, Chu et al.'s research[16] visually demonstrate the complexity of metrics computation for stream processing applications. The graph shows how throughput (solid line) and latency (dotted line) vary with data pressure. In particular, the blue horizontal line represents the Maximum Stable Throughput of the system; P represents the number of data generators, representing the size of the data pressure. We can see that when the data pressure P is less than or equal to MST, the throughput gradually reaches the data pressure P after a period of increase. At the same time, the delay fluctuates at a certain level, and the system remains stable. When the data pressure exceeds a critical value, the system's throughput exhibits fluctuations, and the latency rises sharply. At this stage, the system is unstable, and performance testing is meaningless.

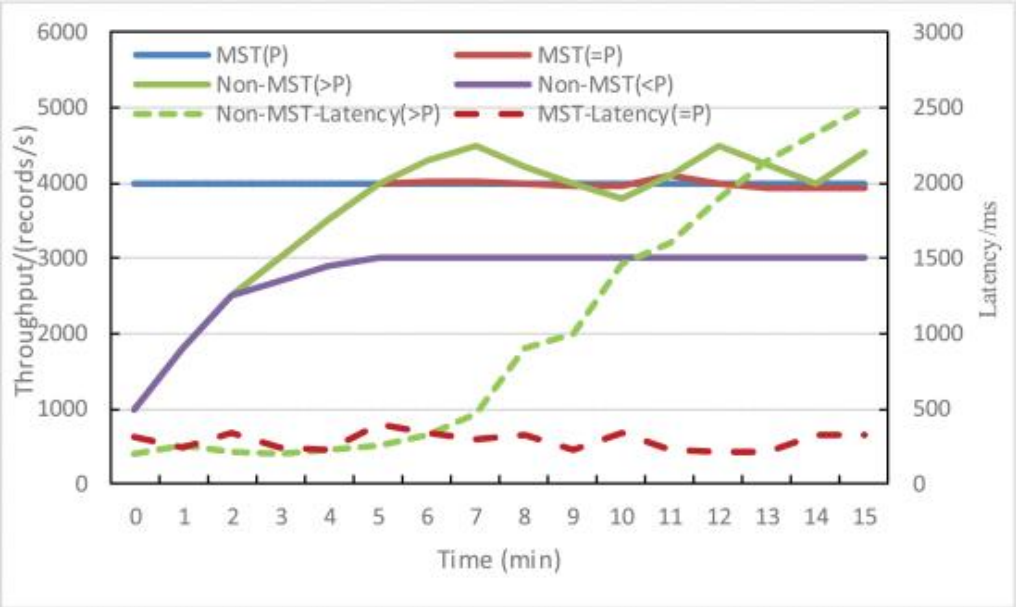


Figure 7: Performance metric changed over time for different data pressures[16]

What is more, quite a few factors can cause the performance of a stream processing system to vary. Common factors include skewness of the data and the configuration of the cluster. Liu et al. in 2021 studied the impact of deploying a cloud environment consisting of a container network on a stream computing engine [18], and they found that the container caused more performance penalties than the container network. Dessokey al. 2022 [14] discusses how the memory management layer affects the availability and performance of stream processing systems. Qian [19] et al. studied the impact of faulty nodes on throughput and latency in 2016 and found that Spark has more robust error tolerance than Strom. Guo et al. [20]proposed a guided machine learning (GML) approach to tuning up to 300 parameters in the configuration of Flink in 2021, achieving significant performance gains.

2.2 Benchmark

The benchmark is an essential tool for measuring the performance of stream processing applications. Several literature have given the details of big data processing benchmarks.Han et al. in 2018 gave an overview on benchmarking of Big Data systems [21], summarizing the essential components of benchmark architectures for big data processing. However, there are still few benchmark features and comparisons for stream processing systems. In this section, introductions about some benchmark suites will be given. We try to follow the design ideas of several widely used open source frameworks and find out why they are successful. Meanwhile, we also critically evaluate their work based on their papers and our understandings.

2.2.1 Stream processing benchmark overview

Due to the complexity of computer system architectures, it is difficult to judge the performance of software running on a cluster based on computer specifications alone. Benchmark is a widely used technique for evaluating the performance difference between computer systems. It is an essential and indispensable step in deploying applications and comparing technical components. For Big Data processing systems, benchmarking requires identifying the specific application domain, building the corresponding test load accordingly, and testing the performance metrics on different computer systems.

Benchmark is also a kind of software whose composition can be described from different perspectives. From a design perspective, the benchmark designer must provide the data set, workload and metrics. From the point of view of software components, the benchmark will provide three kinds of programs. One is the program that handles the test data and is responsible for loading the prepared data into the specified format. The second is to load the workload, allowing the computational engine to execute the workload. The third program performs the performance test on the computer system and issues corresponding performance reports.

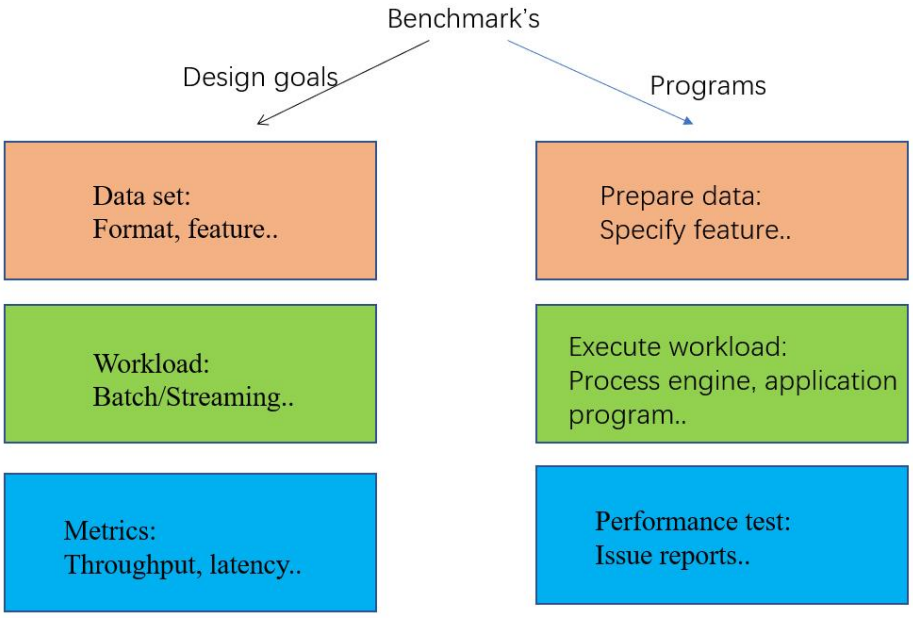


Figure 8: The components of benchmark of data processing application

Data sets can be broadly classified into three categories in terms of form. One is structured data, which is usually table structured data satisfying the relational database storage paradigm; they have transparent, logical relationships and constraints and are easy to store and manage using database systems. The second is semi-structured data, which are usually organized by tag pairs or key-value pairs but are relatively loose compared to structured data. Semi-structured data are usually stored using formats such as XML (eXtensible Markup Language) and JSON(JavaScript Object Notation), suitable for some specific application. The third is unstructured data, which usually does not follow a fixed data model and is difficult to store in traditional relational databases, such as audio or video data.

The workload is an essential part of the benchmark, which corresponds to the application scope of the benchmark. Different aspects of the application scenarios of the compute engine are usually tested in a benchmark. Workloads can be divided into application scenarios, computational models, computational engine API levels, or focused utilization of computational resources. For example, a typical workload can be classified as compute-intensive, IO-intensive, and network-intensive computing by computational resources; and can also be divided into micro-benchmarks and macro-benchmark by tested scope size. The following Table1 lists some standard classifications of workload.

Metrics reflect how well a computer system performs. The performance metrics being tested usually reflect the actual needs of the application, including response time, transfer rate, throughput, and resource utilization. For stream processing applications, the goal is usually to achieve low latency, high throughput, high scalability, and high availability. Moreover, a profiling tool is helpful software that collects program runtime information (such as CPU usage, network bandwidth, and cache hit rate). In many cases, Profiling can cooperate with a benchmark to analyze system performance.

Computation resources	Compute-intensive, IO-intensive, and network-intensive computing
Tested scope	micro-benchmarks and macro-benchmark
Computational paradigms	SQL query, batch processing, stream computing, graph computing, machine learning
Application area	Search engines, social networking, e-commerce, media, gaming, self-defined
Calculation latency	Online calculation, offline calculation, real-time calculation
Data sources	Artificial data, real application data

Table 1: Classifications of workload.

2.2.2 Yahoo streaming benchmarks and extension version

Yahoo streaming benchmarks (YSB) is a widely used stream processing benchmark originally released in 2015[22]. YSB was initially used by the Yahoo Storm team to explore the performance of the Storm stream processing framework. It was subsequently extended with benchmark versions of Spark, Flink, and Apex.

YSB has only one workload that simulates a production environment for identifying relevant advertisement events. This should be used mainly to read JSON formatted streams from Kafka; the rate of generated streams is fixed and can be adjusted through configuration. Subsequently, the semi-structured data stream is filtered to remove unnecessary fields. The ad event is then added with the fields stored in Redis[23](a memory-based non-relational database). Finally, this data stream is aggregated by the time window, and the result is stored in Redis as the Figure9 shows the architecture of YSB. The designers believe that the in-memory execution nature of Redis ensures that the join and sink steps of the data stream will not be an execution bottleneck. During testing, stream jobs keep running for 30 minutes. When throughput keep stable, latency and throughput information (at the millisecond level) is read from Redis by timestamps, and performance reports are generated accordingly. Since YSB can record the delay of all messages in the time window, the statistical analysis of performance indicators will be more accurate (such as box plots).

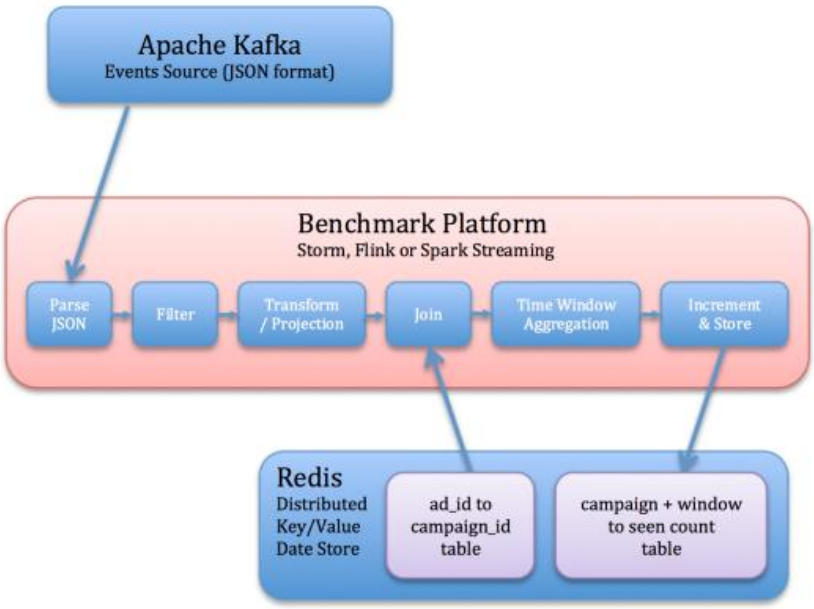


Figure 9: The structure of YSB’ s workload[23].

The workload used by YSB simulates a very simple application scenario. YSB is widely used, and a large number of stream processing tests have used this load, which is quite repeatable. Moreover, the frequency of using Redis performance test is very high, which can be used to explore the changes of Spark within a batch cycle. Chintapalli et al. (2016) [23] tested an early version of stream processing in experiments, linearly increasing the rate of data flow, to observe changes in latency and throughput under different stream processing frameworks. The experimental results show that Flink and Storm have advantages in latency, and the throughput shows a linear response. While Spark exhibits fluctuations in data pressure due to micro-batch latency, it also has higher throughput.

It is worth mentioning that the Ververica team made an extended version of YSB in 2016 [24]. They made some modifications and additions to the original version's application layer and data storage layer. These extensions include using the native windowing API, using Redis directly to eliminate fault-tolerant state overhead, and using the Flink node to store in-memory data. The Ververica team replaced the data source from Kafka with a built-in datagen source and found a 37x improvement in throughput performance! They believe[25] that in YSB, the bandwidth between Kafka and stream processing clusters becomes the bottleneck of the job. The discovery of the extended version of YSB means that the dependent components (Kafka, Redis) outside the stream processing framework significantly impact performance, and the checkpoint overhead for maintaining fault tolerance and consistency is also significant.

2.2.3 HiBench streaming benchmark

HiBench suite is a Intel benchmark tool, including stream processing tests for Spark streaming, Flink, Storm, Gearpump(another stream framework). The current latest version is 7.1.1[4], and the community is still active. HiBench suite consists of three parts,

including HadoopBench, SparkBench, and StreamingBench. HiBench can be deployed on cloud platforms, containers and locally.

Initially, in 2010 [26], HiBench developed the performance of Hadoop deployed on the cloud platform and evaluated the speed, throughput, resource utilization and bandwidth of HDFS of the Hadoop framework[27]. In 2016, SparkBench was developed to comprehensively compare the performance of Spark and Hadoop on different workloads. In 2020, StreamBench was developed to compare four standard stream processing frameworks. The data source in HiBench comes from HDFS, and Kafka is used as the message queue. After the stream processing engine, the result is finally written to Kafka. Performance metrics are measured by Kafka, which saves a Topic about performance detection.

HiBench was mainly used for detecting batch processing (Hadoop,Spark) frameworks[28], so the data source configuration was originally the size of the file in HDFS. However, in stream processing, the stream data rate from Kafka can also be configure. HiBench has a wide range of applications. HiBench offers a variety of loads in three components for a total of 29. It can be divided into six categories: micro, ml (machine learning), SQL, graph, Websearch, and streaming. Its performance detection and overall architecture rely on external components, which may be the bottleneck of the overall system. In addition, Kafka only provides statistical indicators for performance testing, which cannot accurately reflect the delay of each message. In addition, Kafka provides tests infrequently (usually on the second level), making it difficult to judge what happens in a small time interval (such as a batch in Spark).

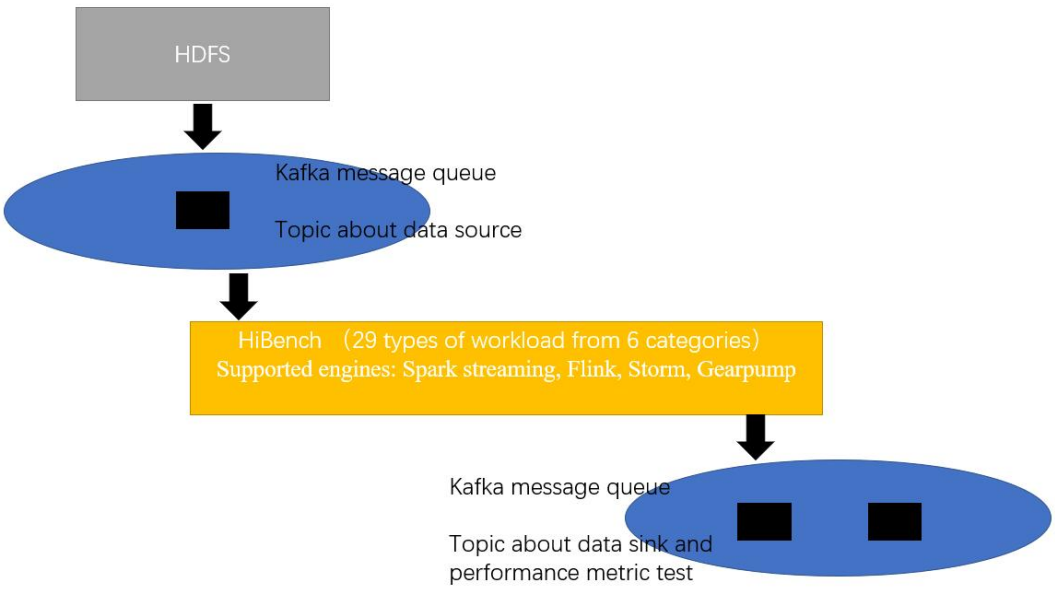


Figure 10: Architecture diagram HiBench StreamingBench

2.2.4 Nexmark streaming benchmark

The Nexmark[29] benchmark framework is mainly inspired by the stream processing benchmark supported by Tucker P et al.[30] and the Apache Beam Nexmark Suite [31]. Nexmark designed a more complex practical application scenario, a business model of an online auction system (shown in Figure11), and designed 23 queries. These queries relate to the specific business of the online auction platform and are derived from the original papers of the nexmark benchmark, Apache Beam and subsequent application scenarios. Moreover, these queries are given in the form of SQL statements.

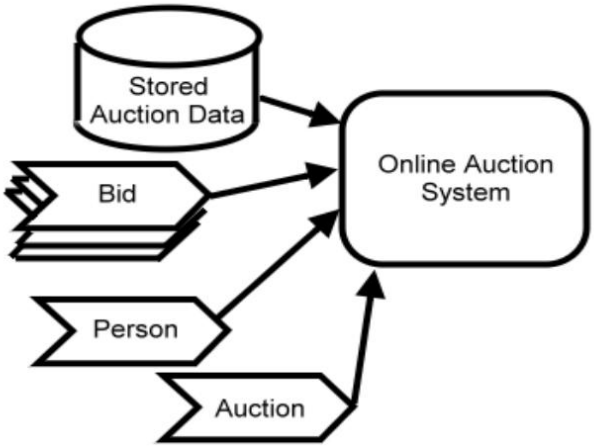


Figure 11: Structure of On-line Auction System [30]

Explanation: Stored Auction Data, Bid, Person and auction are four types of messages in online auction system, query is designed to process these messages.

Compared with other stream processing benchmarks, Nexmark has the following advantages: 1. There is a corresponding cloud platform deployment scheme, which can be quickly deployed on the cloud platform. 2. Pure memory data source generator, avoiding external source and sink dependency. 3. Automatically collect performance indicators. The metrics collecter design is similar to the yarn resource scheduler. In order to improve the accuracy of the CPU performance test, the performance metric API called Status.JVM.CPU.Load provided by Flink is avoided. 4. The test process is automated, the overall test time can be determined (running all queries for about 50 minutes in total), and a clear benchmark test report can be issued. The test process considers the system warm-up, which may impact the test's initial performance in the benchmark. 5. Open source, with a continuously updated community environment.

Nexmark also has obvious limitations. First for the stream processing engine but currently only supports Flink for testing and comparing the differences between different Flink versions. Using Nexmark to compare different databases needs to scale by itself. The reason given in the document is that SQL libraries such as Spark are not enough to correspond, and many query functions cannot be benchmarked. Secondly, for the performance test, the current lack of latency test, the benchmark designer believes that the latency measurement method depends on the specific business. This part needs further improvement.

2.2.5 Summary

In the previous section, we discuss some technical details and design architecture of stream processing benchmarks, aiming to summarize the benchmark characteristics and guide the experiments. Next, we summarize three aspects of stream processing benchmarks: comparisons between the benchmarks mentioned above, challenges of design benchmark, and possible application scenarios.

Four open-sourced, widely-used stream processing engines are discussed: Yahoo streaming benchmarks (YSB), extension version of YSB, HiBench, Nexmark. These benchmarks show prominent staged development characteristics, with advantages and disadvantages listed below.

	Data source	Workload	Process engines	Performance test
YSB(Yahoo streaming benchmark)	Semi-structured，steady streaming data rate is configurable by Kafka.	AD identify .	Storm, Spark, Flink, Apex	From Redis by time stamps, high frequency.
Extension version of YSB	Semi-structured, steady streaming data rate configured by internal generator	micro-bench marks of YSB and some extension.	Storm, Spark, Flink	By time stamps.
HiBench	Semi-structured,steady streaming data rate is configurable by Kafka.	29 types of workload from 6 categories.	Hadoop, Spark, Flink, Storm, Gearpump	From Kafka, low frequency, simple performance report.
Nexmark	Structured, Linearly varying streaming data rate with different proportion configured by internal generator	23 kinds of SQL queries.	Only Flink	Similar to yarn resource scheduler. Automated testing and report generation

Table 2: Characteristics of some stream process benchmark

Overall, in order to simulate a production environment, Benchmark must rely on some components, but these components may become performance bottlenecks in the application. Extension YSB and Nexmark provide built-in data sources to solve this problem. At the same time, the data source usually only provides a fixed rate of data flow, and Nexmark provides a more fine-grained configuration. Additionally, the time stamp is a standard performance statistics technique that provides an accurate distribution of

performance over time (for each message). When Kafka is used as a performance testing tool, it only provides some delay and throughput statistics. In addition, stream processing engines can provide performance metrics structure for monitoring clusters but are generally considered inaccurate. Finally, workloads vary widely across benchmarks and involve different levels of APIs in stream processing frameworks, so test results are not comparable across benchmarks.

Designing or selecting good benchmarks is challenging. A reliable benchmark must be available, repeatable, relevant, and comparable. That is, the test metrics involved in a benchmark should accurately reflect the value of the computer system for processing the business. One of the workloads should be closely related to the application hotspots in the field. That is why industry-leading corporate teams develop standard benchmarks. We can deduce that a good Benchmark should have the following characteristics: 1. The Benchmark should preferably be open source to ensure that the results are widely available and comparable in the same field; 2. It should be able to produce clear and easy-to-understand reports; 3. It should be designed by designers familiar with the business to determine the test load and ensure a specific authority; 4. Benchmark design should take into account the speed of application development. Moreover, the benchmark should have a certain degree of extensibility to facilitate future workload and test metrics to expand.

The primary purposes of benchmarks are diverse. Firstly, the benchmark can evaluate whether the hardware meets the application requirements in application development. The performance benchmark is established under a given load by changing the hardware configuration of the computer system, such as increasing the number of servers (horizontal expansion), optimizing the network environment, and increasing the amount of memory. Examine the impact of hardware on performance, including scalability, availability and fault tolerance.

Secondly, the benchmark could evaluate software performance. The performance benchmark is established under a given hardware environment. By changing the software library or algorithm that the workload depends on, such as replacing a different data processing engine, or optimizing the algorithm, examine the impact of software on performance, including engine performance or library performance. The benchmark in this experiment is mainly used for this purpose. From the application development perspective, the benchmark can evaluate the performance of the software processing business and the appropriate software selection. On the other hand, the benchmark test results also provide a baseline for developing data processing software, which is conducive to improving subsequent versions.

Thirdly, the benchmark can simulate a natural production environment. The performance benchmark is established for given computer systems and applications by changing the properties of a given data source, such as data skewness, data scale, or the rate of stream data generated. Increasing data pressure will lead to a higher probability of system crashes, a standard method to exploit system bottlenecks. Benchmark should provide configuration parameters to modify the data feature. In benchmark tests, the data source is usually artificial data. For real-world applications, this process is called stress testing.

The most common use case of this type in stream processing is to continuously increase the data rate, monitor data such as performance inflection points or how to reach maximum throughput.

Chapter 3 Stream processing architecture

Stream processing systems are usually distributed. It means that there needs to be a resource manager to coordinate the assignment of tasks and handle node crashes. At the same time, the generation and storage of data also depend on different components. The Figure12 below shows the direction of data flow in a stream processing system among different components. In Section 2.1.4, we present the hierarchical architecture in the data processing. In the first four sections of this chapter, we give technical details of the components in the stream processing architecture. In Section 3.5, we summarize how the configuration of the stream processing engine can optimize performance.

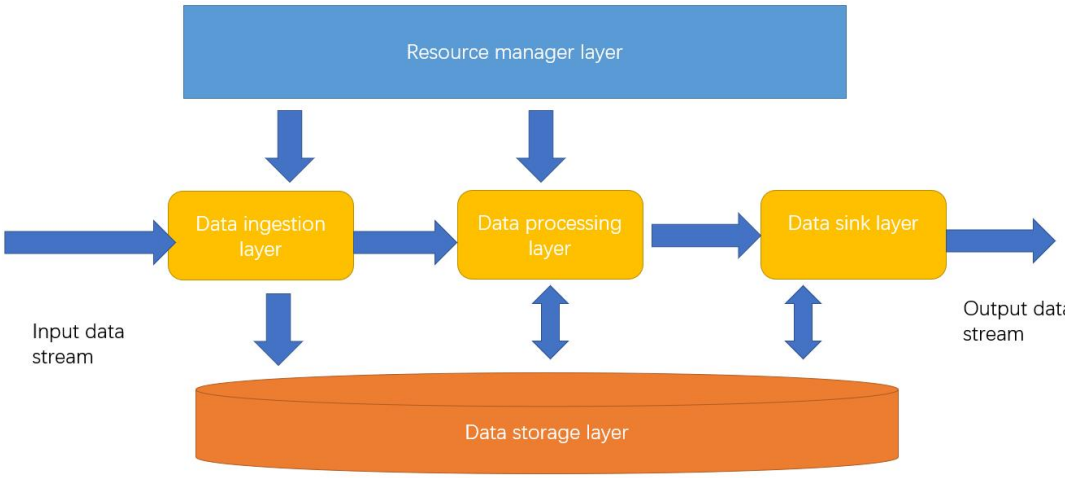


Figure 12: Distributed stream processing architecture diagram with data flow direction

3.1 Hadoop

Hadoop MapReduce is Google's earliest used to solve the problem of web indexing and querying in 2003[10]. Many different components within Hadoop can complete the complete batch process. At the same time, these components can also provide external services independently. As a distributed computing framework implementation, MapReduce is different from OpenMP, MPI, and other computing models in design. We mentioned this model in Section 2.1.3. HDFS is a distributed file system that can quickly access large amounts of data and provide external services. In addition, Hive allows MapReduce to be implemented through SQL query statements.

HDFS distributes large amounts of data processing to clusters of inexpensive hardware. There are two main types of processes in HDFS： The name node divides vast files into small pieces of files and stores them in a distributed manner. Each block holds a specific amount of data for access, usually set to 128M. The divided nodes are called DataNodes, and these data blocks are fault-tolerant through backups. HDFS can provide and store data for Kafka. Yarn is a resource scheduling component and is compatible with Spark and Flink to deploy on the cluster. Similar Zookeeper is also a resource coordination component, and the operation of Kafka depends on Zookeeper.

3.2 Kafka

Kafka was developed by Linkedin Corporation, and it is a message streaming platform based on Zookeeper. Kafka was open-sourced in early 2011 and supported multiple partitions and replicas to solve end-to-end event streams. Kafka is based on a publish-subscribe message engine system, which means it can continuously import and export data from other systems. In addition, Kafka can also store and process event streams. Kafka provides high-throughput and low-latency data streams and implements high-concurrency message delivery between consumer and producer groups. Kafka achieves excellent scalability and fault tolerance through the partitioning and backup mechanism[32].

The data unit in Kafka is called an event or a message. The event can be regarded as a row record in a database. A message usually has a key, value, event stamp and other information. Kafka uses batch processing, messages will be written and read from Kafka in batches, and batch generation refers to a group of messages. Additionally, events are organized and persistently stored in topics, similar to tables in a database or folders in a file system.

As a distributed system, Kafka consists of a client and some servers, which can be deployed on a local environment, cloud environment, virtual machine and container. Client applications that publish messages to topics are called producers, and producers are used to continuing to send messages to a topic. Client programs that subscribe to topic messages are called consumers, and consumers are used to processing messages produced by producers. A topic in Kafka In Kafka, a topic always has several producers and consumers(shown in the Figure 13). These producers and consumers are wholly decoupled from each other; the producer does not need to wait for the consumer. Publish and subscribe mechanism ensures high scalability in Kafka.

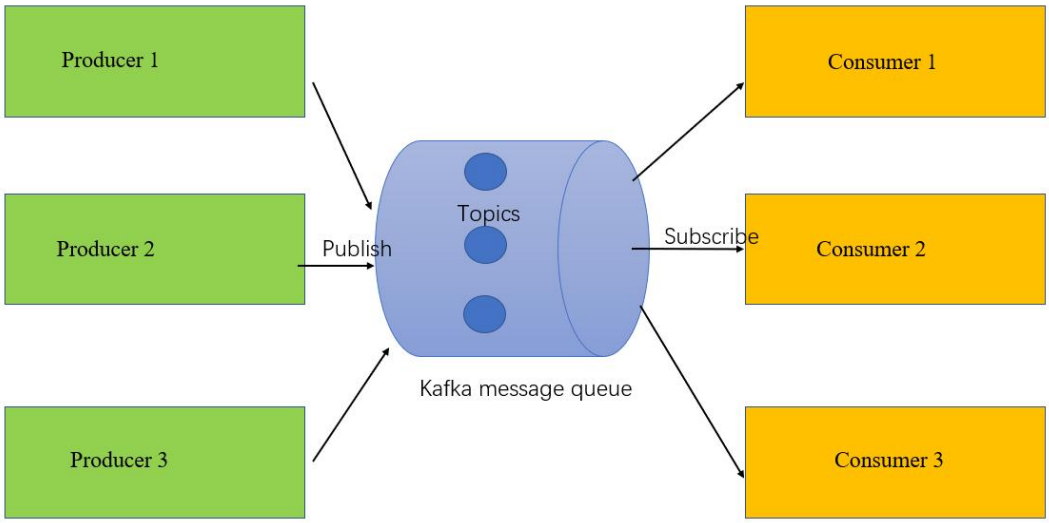


Figure 13: The consumers publishes and the producers subscribes the topic

In Kafka, topics are divided into multiple partitions. Each partition is an ordered set of message logs. As shown in the Figure 14, producer1 publishes a new event in the topic, and this message will only be sent to one of the partitions. Messages with the same key are sent to the same partition. Each partition is ordered, and each message in the partition is assigned a continuous numerical ID called offset, which ensures the order of events. This distributed storage method increases the system's scalability, provides load balancing capabilities, and allows multiple client programs to read and write data on topics simultaneously.

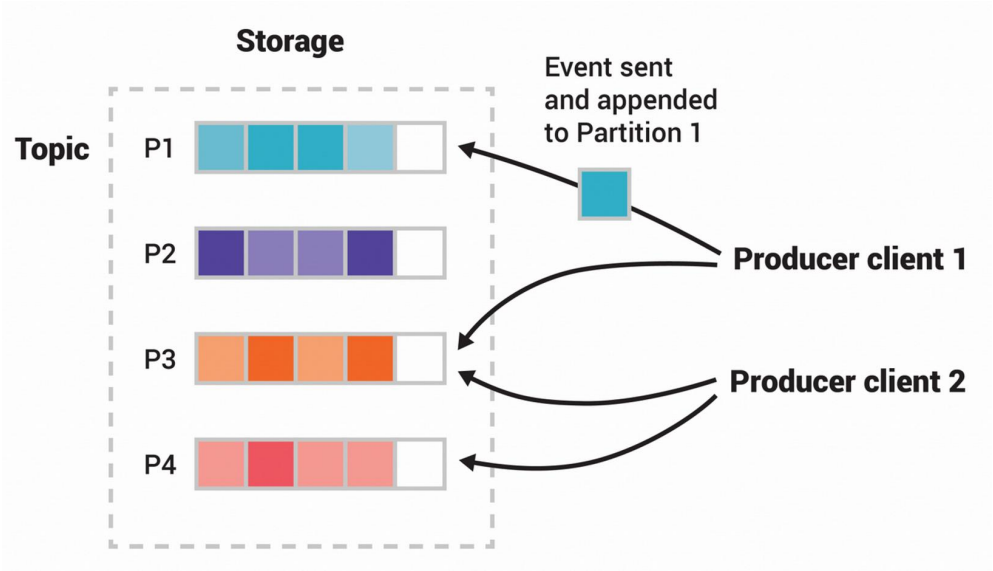


Figure 14: Partitioned topic in kafka[32]

Kafka provides a variety of APIs for different application scenarios. The most crucial application of Kafka is the middleware for messaging, which is used for messaging between applications. In a complex distributed application, one part of the program

generates data, and another receives messages. Kafka can be responsible for coordinating and scheduling message delivery, backup and fault tolerance of messages. When the data traffic is substantial, it has an apparent buffering effect to protect the regular operation of the system. In addition, Kafka can provide a stream processing interface to implement basic stream processing applications. Such as developing event-driven decision-making systems or data transfer between storage systems. Kafka can also provide a log system. Logs are continuously generated streams and can be stored in Kafka topics. Finally, Kafka can also perform performance indicator detection, which is used to collect various performance data of distributed applications and provide centralized feedback, and accordingly, provide performance including or implement application alerting systems.

The data detection process of Kafka is shown in the following figure. Kafka provides end-to-end latency detection, representing the time interval between when the producer produces data and when the consumer pollutes the data. Kafka subdivided end-to-end delay into five parts, including produce, publish, commit, catch-up, and fetch time, and provides a responsive interface. Usually, the delay on the consumer side refers to the fetch time of the consumer polling the brokers. In addition, Kafka also keeps statistics of streaming data.

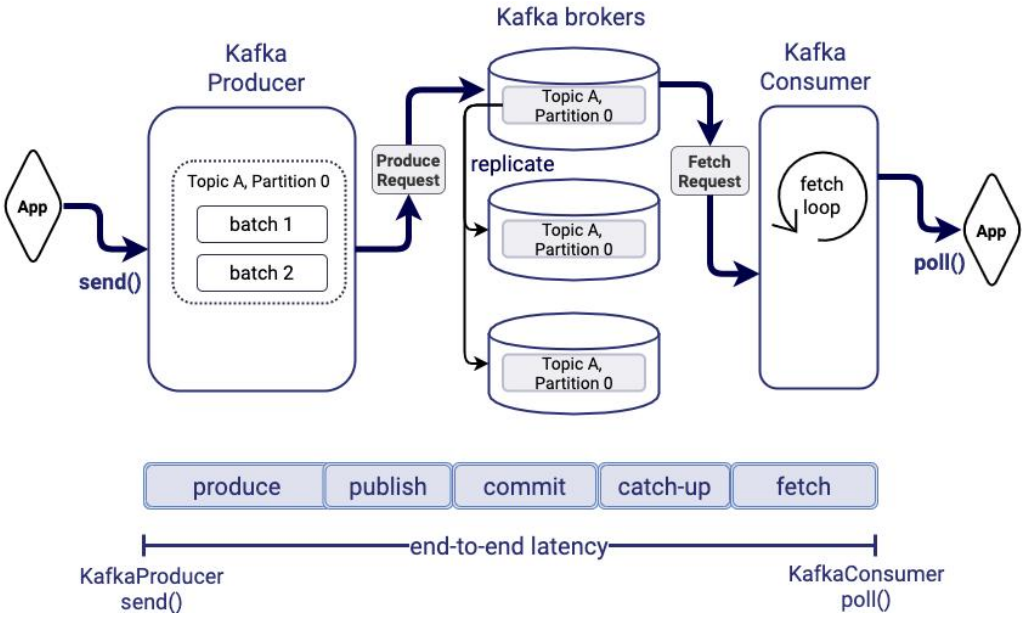


Figure 15: Kafka Latency Test Architecture Diagram[33]

3.3 Flink

Flink is an open-source stream processing framework developed by the Apache Software Foundation, first released in May 2011[34]. 2010 to 2014, the Stratosphere project was initially designed for research on information management in the cloud[35]. The first stable version of Flink was released in September 2015 as an enterprise-distributed big

data processing engine. Currently, the latest version of Flink is 1.15.1 and has become one of the mainstream processing frameworks for big data processing.

Flink has many features of an exemplary stream processing framework: 1. Flink provides the ability to compute batch data based on stream processing, enabling batch stream consistency. 2. Flink supports exactly-once semantics. Exactly once means that the message sent to the system can only be processed by the consuming end simultaneously. 3. supports event time, the time at which an event is generated. Most other stream processing frameworks use process time, which is the time of the system host when the time is transmitted to the computing framework. Event time supports chaotic arrivals. 4. highly fault-tolerant state management prevents the state from being lost during computation due to system exceptions, thus producing correct results even in the event of system downtime or exceptions. Figure16 shows that the state is the data generated during the computation stored in memory. Subsequently, computations continue based on these intermediate results so that the original count data does not need to be re-read from external storage. The distributed checkpoints technology ensures excellent fault tolerance by storing persistent **state** information during execution and automatically resuming tasks from the **checkpoint**, ensuring consistency in processing.

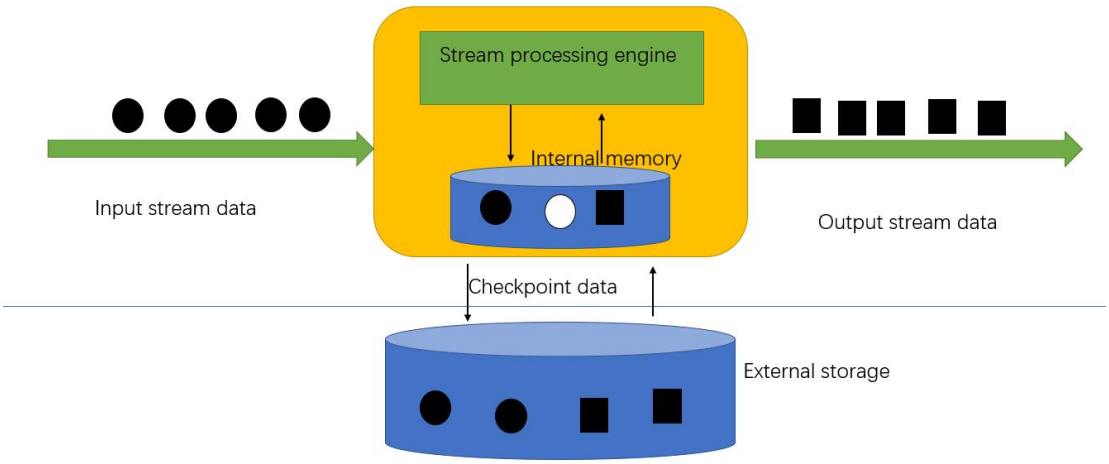


Figure 16: The status management based on checkpoint

5.Flink supports multiple window operations and stream processing applications where stream data is aggregated and calculated within a specific range via windows. Window operations allow for stream data aggregation over a range of windows. Flink window classification based on time, count, session, and data-driven. More flexible than other frameworks to provide API. 6. Independent memory management system provides a configuration of memory allocation, which could improve the application's potential performance ceiling. It provides a smaller granularity of control than other frameworks; almost all parts of the Flink process are adjustable in size[36].

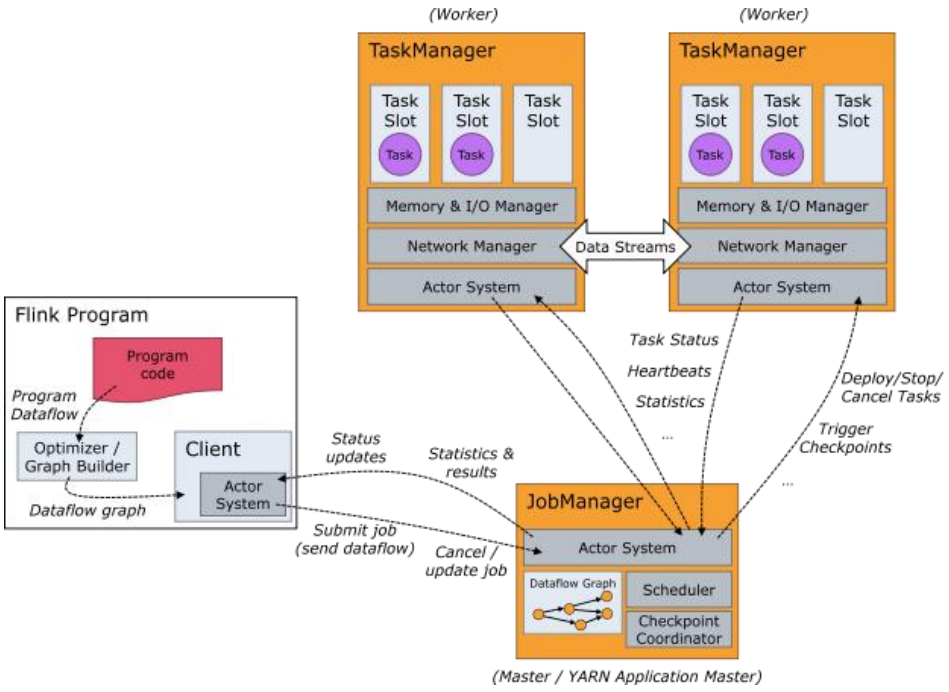


Figure 17: Flink program architecture diagram[37]

For Flink program architecture, as Figure 17 shows, there are two main types of processes in the Flink program: a JobManager and one or more TaskManagers. Job manager load coordinates Flink distributed execution, decides to execute tasks(optimized dataflow graph) at the right time, stores and reads checkpoints, and recovers from execution failures. The job manager accepts tasks that need to be deployed, starts tasks using slot resources, establishes network connections, accepts data and starts data processing. Job manager includes three components: ResourceManager, Dispatcher and JobMaster.

Taskmanager is responsible for executing one or more subtasks (threads) concurrently and caching and exchanging data between processes. Each thread resource is called a slot, the smallest unit of resource scheduling in Flink, and each slot can execute a task. Therefore, the number of slots in a worker represents the **potential number of concurrent** processing. Each Taskmanager is managed by managing the task slot resource pool. Threads in each sub-work share memory resources, TCP links and heartbeat information. It means that multiple slots in the Worker can increase **parallelism** and thus improve the efficiency of processing data.

3.4 Spark

Spark is a distributed open source processing system for big data workloads developed by Apache Corporation and first released in October 2012[38]. Apache Spark is a widely used computing engine with a complete data-parallel processing library covering multiple domains. Spark is compatible with multiple languages, including Python, R, Scala, and Java. Spark can be managed by spark standalone cluster manager, Mesos or Yarn. Spark is designed to read and write data from HDFS or Kafka. The basic

abstraction is **RDD**(Resilient Distributed Dataset), which is immutable, partitioned collection elements and can run in parallel. Spark can process in memory, reducing the steps of the Job, thus increasing the execution speed. Spark can also process batch and streaming data. Spark is primarily designed for batch processing, and streams are considered a particular case of batch processing(micro-batch).

The Application is written by the user, which includes a driver program and executors on the cluster. The Driver process can run the Application's primary function and create the corresponding SparkContext to prepare the running environment of the Spark application. The code of the Spark application can be run in the worker cluster, and any node can start an executor process. The Executor process is responsible for running the Task and saving the data. It is the container that executes the **Task**. Each Task is responsible for computing the data of a partition; the number of tasks represents the **potential concurrency number** of the Spark application. Multiple tasks form a Job, which acts on RDD. The scheduling unit of Job is called Stage.

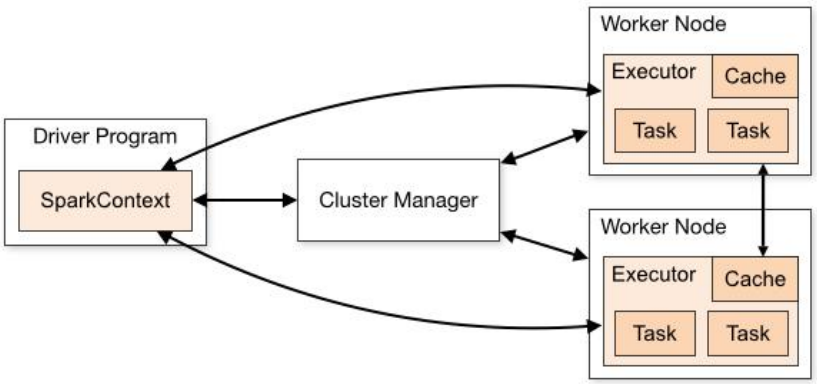


Figure 18: Execution process of Spark application[39].

As shown in Figure, the Driver process establishes the spark context when running a Spark program. The driver applies for related resources and assigns and monitors tasks. The resource manager allocates resources to the executor and starts the executor. Then Sparkcontext establishes a DAG (Directed Acycle graph) according to the dependencies of the RDD; the DAG graph is parsed into Stages, and then submitted to the scheduler for processing. Then, the executor applies a task to Sparkcontext, and the resource manager issues the task to the executor to run and provide the program. Finally, the task runs on the executor and returns to the task manager, writing data and releasing all resources.

3.5 Performance tuning

We need to understand some technical details of the stream processing framework to fully understand the benchmark, including the source of performance penalty and optimization, especially the reasonableness of the workload settings. Moreover, the project's purpose is to evaluate different frameworks(Flink and Spark) fairly. Their development difficulty and performance caps in different scenarios should also be considered.

Stream processing is designed to move most of the computation into memory. Cluster resources (CPU, network bandwidth, memory) can all become performance bottlenecks. Performance tuning methods revolve around the full utilization of computing resources. As a rule of thumb, the main performance bottlenecks of stream processing programs are network bandwidth and memory resources, both of which need special attention. For clusters with specific hardware resources, there are three main ways to optimize the performance of stream processing programs. **1.Adjust data serialization; 2. Increase concurrency; 3.Optimize memory management**, corresponding to network bandwidth resources (mainly), CPU resources, and sufficient memory resources.

Firstly, serialization is commonly used in data and access and communication processes. Serialization refers to converting an object into a binary byte stream, saving it as a file, or transmitting it to other nodes through a network port. When this object is re-read, the operation that is performed is called a deserialization operation. Serialization operations involve a variety of computing resources (disk I/O, data serialization, and network). They account for a high proportion of distributed computing, so the impact of serialization on overall performance cannot be ignored. The serialization process takes too long or too many bytes, making it a performance bottleneck.

When the distributed big data processing engine executes, it needs to use data on other nodes. These network-related operations involve serialization and deserialization, such as map, flatMap, reduceByKey and other transformations. For example, the following process occurs in the Spark program: the required data objects in the code are serialized locally by the driver program. Then the serialized objects are transmitted to the remote executor process through the network and are deserialized into objects by the executor. Finally, the remote is executed in the node.

Two serialisation libraries are provided in Spark for concrete serialisation implementation. One is to use the ObjectOutputStream framework with Java to serialise objects by default. In this case, almost all objects can be serialised. The second is Kryo serialisation, which is usually about ten times that of Java's serialisation [40]. For Flink, native data structures such as Tuple are provided, which significantly improves the efficiency of serialisation. As shown in the figure, Flink provides a variety of serialisation methods, and most of the performance is higher than Kryo serialisation.

Secondly, memory tuning in Spark mainly uses the Java virtual machine, and a standard policy tuning is garbage collection. Flink provides a more refined and adjustable memory model. As shown in the figure, most of the memory sizes in the figure can be tuned. Multiple choice of memory tuning also raises the potential performance cap.

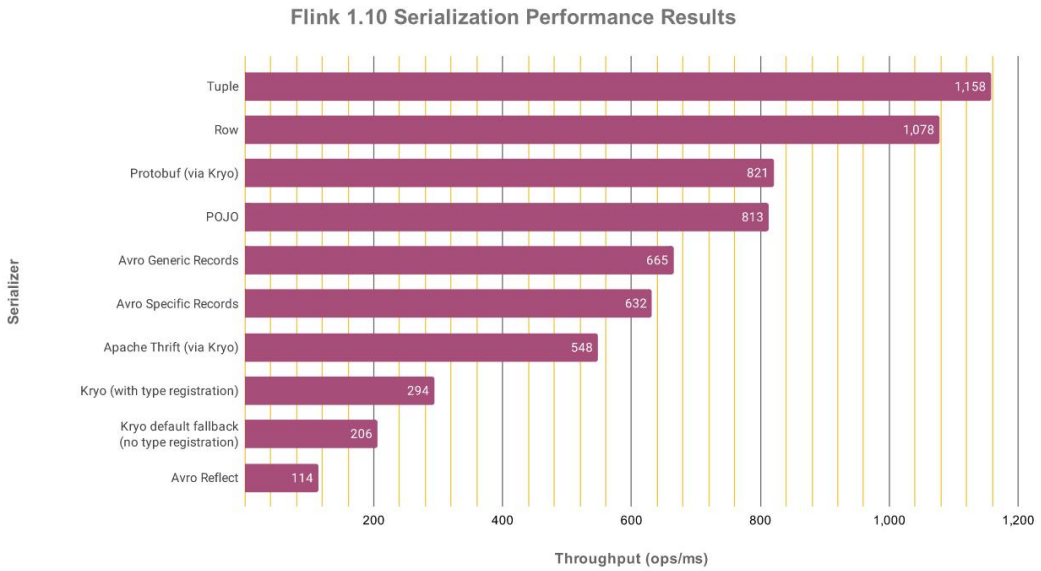


Figure 19: Serializer speed in Flink1.10[41]

Thirdly, concurrency in Flink is the number of slots, while in Spark, it is usually called Task. In different technical contexts, both are by increasing the number of threads. Adding threads ensures that the performance of the cluster can be fully utilized. It is generally recommended to set the concurrency to 2-3 times the number of CPU cores to use the cluster's performance fully [40].

Another benefit of increasing concurrency is improving the efficiency of shuffle operations. The shuffle operations (sortByKey, groupByKey, reduceByKey, join) build a hash table in each task to perform the grouping, which can often be large and generate an OutOfMemoryError. At this time, by increasing the degree of concurrency, the amount of data for each task can be reduced, thereby improving memory utilisation efficiency.

Chapter 4 Experiments design

In this chapter, we discuss the methodology of experimental design. We mainly use the HiBench benchmark. In Sections 4.1 and 4.3, we give the environment required for the experiments. In Section 4.2 we discussed the content of the workload and the operations represented. Section 4.4 to Section 4.6 describe the methodology of the experiments.

4.1 Cluster environment

In this experiment, we try to complete the experiment on a local laptop using a Vmware virtual machine. Because we are getting in the way of running Benchmark on HPC clusters (for reasons mentioned later), the results on the virtual machine cluster can serve as a baseline for cloud computing and container environments, which is helpful for follow-up research. The following Table3 shows the configuration of the cluster environment. We used a controller/agent cluster consisting of three virtual machines. Considering the resource consumption caused by too many virtual machines running locally, which is likely to cause potential performance bottlenecks, we use a small number of clusters. However, it is enough to explore the characteristics of distributed stream processing applications on clusters.

Cluster configuration	three CentOS virtual machines on the laptop, one as controller node and the other two as agent nodes.
Host configuration	Processor: Intel(R) Core(TM) i7-10750H CPU@2.60GHz 2.59 GHz;2*8=16 cores; Memory: 16GB.
Virtual machine configuration	CentOS 7.7.1908 is used on each node. Controller node 8 cores, 4 GB of memory; each agent node 4 cores, 4 GB of memory.
Network configuration	use NAT mode, which share IP address with hosts and configure ports separately as 20.20.20.141-143

Table 3: Related hardware configurations

In the local computer, we deployed three CentOS operating system virtual machines. The host has more computing resources (8cores and 4GB memory). The three computers are linked to each other through the NAT mode network. The nodes in the cluster are configured with the SSH(Secure Sockets Layer) protocol for password-free remote login.

4.2 Workloads

In Section 2.2, we compared the four available stream processing benchmarks and found that each has advantages. We finally chose HiBench's StreamingBench as the experimental workload. HiBench supports four stream processing frameworks, including Flink and Spark simultaneously, which is suitable for comparing the performance of different stream processing engines. In addition, HiBench provides more diverse and simple (close to Mico-benchmark) workloads, which makes performance analysis more accessible. Finally, HiBench also provides a mirror image of the cluster, which is also convenient for testing the future container network's performance.

We describe the HiBench feature in Section 2.2.3. Hibench uses Kafka as the data stream source and destination and HDFS as the data storage layer. Kafka measures the performance of the stream processing engine as a consumer and provides throughput and latency statistics as topics, and the final results are written into the performance report in CSV format. HiBench's StreamingBench has the following four workloads:

1.**Identity**

The workload does not do any specific business, and the stream processing engine is **idle**. It can be considered a necessary overhead for streaming application startups.

2.**Repartition**

Accept the data stream and change the data parallelism by adding or removing the number of partitions. It tests the efficiency of data **shuffling operations** in streaming frameworks.

3.**Wordcount**

This workload cumulatively counts the number of words of the same kind in the data stream for every few seconds. This load tests the performance of **stateful operations** in the streaming framework and the cost of using Checkpoints.

4.**Fixwindow**

This workload performs fixed-window-based aggregations. Hibench generates textual data streams containing web access events. The text is parsed into three fields, including IP, sessionid and browser, and the number of visits is accumulated every 10s fixed window according to IP. The fixwindow workload tests the performance of **window operations** in a streaming framework. Meanwhile, it is still a **stateful operation.**

4.3 Deployment

HiBench is a benchmarking component developed based on the Maven project build tool. HiBench mainly consists of Scala, Java, and Python/shell scripts. During the deployment process, we used jdk1.8.0 and the corresponding maven version to compile

HiBench7.0.For external software dependencies, we installed Zookeeper-based Kafka as a message queue. Hadoop2.8.3 as the data storage layer. Flink1.10.0 and Spark2.2.2 are the tested stream processing framework versions. One of possible software dependency version combinations is listed in Table4. It is worth mentioning that HiBench is not very compatible in software dependency, and errors caused by version mismatches often occur in deployment process. As the results, most software versions are not up-to-date.

Benchmark and compile tools	
JDK	jdk1.8.0_121
Python	Python 2.7.1
Scala	scala-2.11.8
Maven	apache-maven-3.3.9-bin
HiBench	HiBench-7.0
Stream processing dependency	
Kafka	kafka_2.11-0.8.2.2
Zookeeper	zookeeper-3.4.10.tar
Hadoop	hadoop-2.8.3
Stream processing engine	
Flink	flink-1.10.0-bin-scala_2.11
Spark	spark-2.2.2-bin-hadoop2.7

Table 4: Version list of software dependency

4.4 Metrics test in HiBench

In HiBench, performance metrics are usually detected using Kafka. The measurements of performance are instantaneous. As shown in the Figure20, the measurements of latency and throughput are both on the consumer side. Latency in HiBench is fetch time, which is final part of end-to-end latency. Also, tested latency is always processing-time latency.

Here are some performance metrics records which we can get from single test supported by Hibench:

1. **Throughput (TPS)**: The number of requests processed per unit time, which represents the processing speed of the system.

2.**Quantile Latency(LTq)**: The q is a percentage number, LT99 represents the minimum time required to process 99% of network requests in cosumer end. LTq represents the response time of the system. HiBench uses LT50, LT75, LT95, LT98, LT99, LT999 in latency test. Meanwhile, the maximum latency and minimum latency are similar concepts.

3.**Mean Latency(LT_mean)**: The mean value of the instantaneous latency in cosumer end, which partly represents the stability of the system. Because the distribution of the response time of a stable stream processing system to the message is close to the Poisson distribution, and only a few messages are lost in the middle or arrive late due to retransmission. Average latency is susceptible to extreme values and is not suitable for representing system response time.

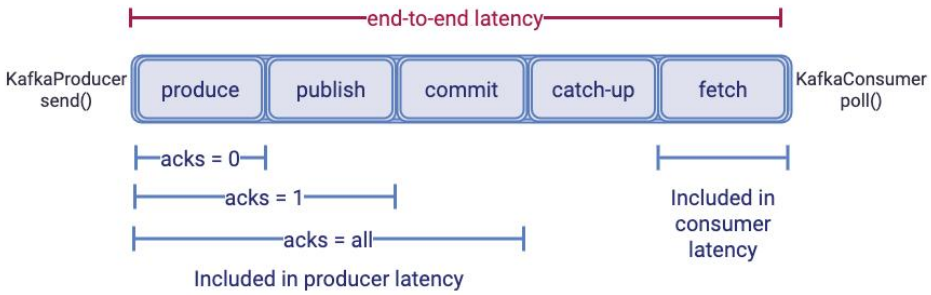


Figure 20: Components of end-to-end latency in Kafka[42]

Explanation: Fetch time is the latency tested in Kafka. Metrics of latency and throughput tested in HiBench are always consumer side.

4.5 Performance metrics and model assumption

In section 4.4, we gave the instantaneous performance metrics that HiBench can test, and what they specifically represent. However, stream processing applications are usually a continuously running system, whose performance metrics are not stable. Specifically, the instability is manifested in two aspects, one is the characteristics of the data source, and the other is the contention to system resources over time. In streaming systems, input data characteristics, such as skewness, rate, can also impact performance metrics. In HiBench, the data source rate is usually constant within a test. The data source generation rate is the main configurable parameter. Even if the rate of a data source is determined, its performance metrics will fluctuate over time, mainly due to changes in system resources allocation when processing an infinite amount of data. Therefore, we need to discuss the impact of data source rate on performance metrics, and how performance metrics change over time for a given data source rate.

In the description and discussion of system performance, we argue that latency, throughput, and data source rate are usually inseparable; they describe the state of a stable stream processing system at some point together. We have three reasons for this view. First, the processing method of streaming data (stream/micro-batch) naturally makes a trade-off between throughput and response speed. A single performance indicator is difficult to evaluate comprehensively when comparing stream processing frameworks. Second, with the data source Increasing rates, throughput, and response time in stream processing systems have inherent trade-offs, which means they are not independent. For example, in a single micro-batch, the increase in throughput can lead to an increase in response time. Third, it is almost impossible to have only unilateral

performance requirements in actual business requirements. This representation of performance helps meet business requirements.

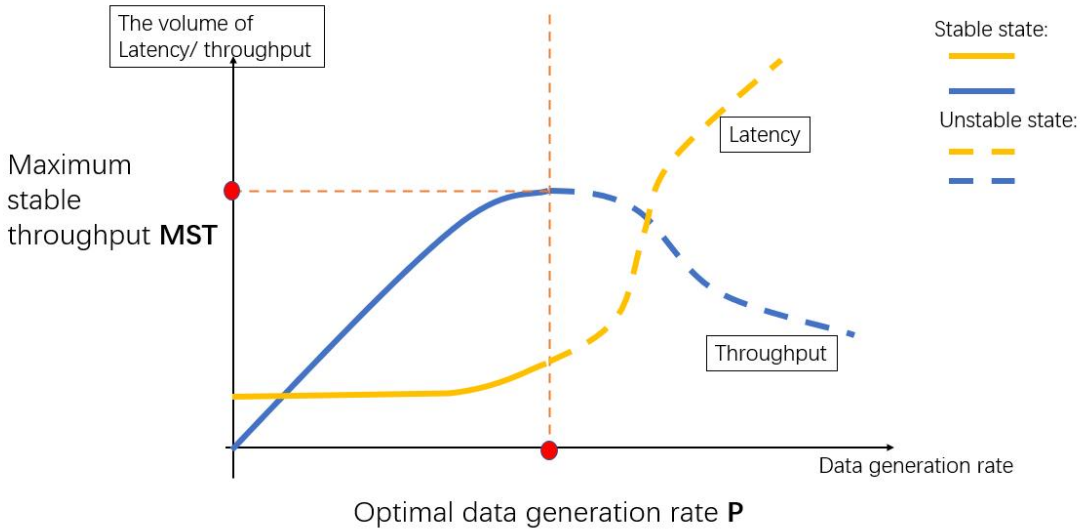


Figure 21: Performance model of stream processing with Maximum stable throughput MST

Explanation: The dotted line represents the instability of the system, and the MST represents the limit processing capability of the system.

As shown in the Figure21, a stream processing system typically changes as the data generation rate increases. Within a specific range, with the increase of the data generation rate, the throughput of the stream processing system continues to increase, and the response time increases gradually. At the same time, due to the relative scarcity of computing resources, the competition will become more and more fierce, and the system will become more and more unstable. When the data generation rate is more significant than a specific value, throughput starts to drop, latency increases dramatically, and the system tends to crash. We can assume a maximum throughput under an **Optimal data generation rate P**. Similar with defination in [16], this project defines **Maximum Stable Throughput(MST)** as the maximum throughput that can continuously ingest streaming data and keep the system stable. The maximum throughput reflects the processing power of the stream processing system's limit.

The problem now boils down how to determinate a system is in a stable state. There are two cases of unstable state, including the warm-up time before the application starts and because the data generation rate is too fast. Average latency might be a solution. However, a single indicator looks pale because it is difficult for us to get the distribution of latency. It also contradicts the principle mentioned above — using throughput and latency together to describe the system state. For the same reason, using the standard deviation of throughput is not very ideal. Therefore, we recommend the use of **composite indicators**. For example, the system is stable when it meets the following conditions simultaneously over a period of time: the response time of LT99 must be within 500ms,

and the average latency must be within 100ms, and standard deviation of throughput is less then 2% of means of throughput. This set of parameters depends on the specific workload. If the load is more complex, the constraints on latency and throughput should be reduced.

In conclusion, we have assumed **three essential principles** in the performance testing of stream processing systems:

1. Use quantile latency to represent system response time. Average latency is only a partial representation of system stability.

2. Use throughput, latency, and data generation rate together to describe the state of a stream processing system.

3. Performance testing is meaningful only when the system is stable, and unstable results should be excluded from experiments. The stability of the system requires joint metric to determinate. In this way, we can exclude unstable data caused by warm-up time and data generation rate exceeding **MST**.

4.6 Research methodology

With the above discussion on the performance indicators of stream processing applications, we can design the following three-step experimental plan.

1. For each workload, find the **Optimal data generation rate P** and the corresponding **MST**. The basic idea of finding P is to gradually increase the rate at which the data stream is generated and determine whether the system is in a stable state. Although it is mentioned in literature [16] that the adaptive method can improve the rate of finding P, in this experiment, we still use the **naive method**, increasing the fixed data pressure each time until the system is no longer stable. This experiment should be repeatable in both single- and multi-machine environments to assess the scalability of the stream processing framework.

2. Evaluate the performance state when the data generation rate is less than P. In this part, we use a multi-machine cluster and can use some of the data in the first step. We examine throughput and latency changes according to the measurement principles assumed in Section 4.5.

3. We will explore the impact of different configurations on cluster performance. In the experiments, we mainly discuss the impact of maximum throughput under a specific load. Possible configurations in Spark and Flink are discussed in Section 3.5. The specific experimental scheme will be introduced in the experimental part.

Chapter 5 Experiments and result analysis

In this chapter, we explored three aspects of the performance of stream processing applications. The flow chart of our experiment is shown below:

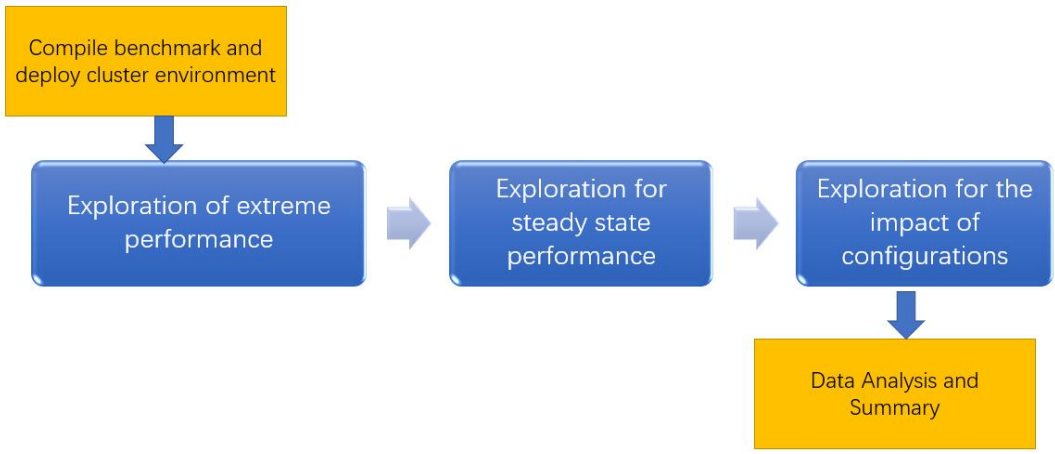


Figure 22: Experiment workflow chart in the project

Explanation: Boxes in the diagram represent specific steps. Each step needs to be performed in order.

5.1 Extreme performance test

In this section, we explored the performance ceiling of HiBench. Specifically, we tried to find the Optimal data generation rate P and the corresponding MST on single-machine and multi-machine clusters. According to our previous discussion, we need to use composite indicators to represent the system's stability. In this experiment, considering that the four workloads are relatively simple, combined with the experimental results, the criteria we used to judge the system's stability are: **the response time of LT99 must be within 5s, LT_mean must be within 5s, and the standard deviation of throughput must be less than 2% of the means of throughput.**

Initially, the data generator was configured with a message size of 200 bytes per message, producing 1000 records per second, or 0.2MB/s. We sampled every 10s through a script. After a 1-minute warm-up time, each load was run for 10 minutes to obtain a total of 60 sample points. Through a naive method mentioned in section4.6, we added 500 records each time; we increased 0.1MB/s rate for each time until the system did not meet the

stability criteria. We have conducted experiments on both single-machine and multi-machine clusters. For the four workloads, we obtained the **Optimal data generation rate P**, shown in the following table.

	Identity	Repartition	Wordcount	Fixwindow
Spark/single node	2.8MB/s	2.3MB/s	0.7MB/s	0.6MB/s
Flink/single node	1.8MB/s	1.3MB/s	0.6MB/s	0.9MB/s
Spark/3 nodes	5.2MB/s	2.5MB/s	1.8MB/s	1.6MB/s
Flink/3 nodes	2.7MB/s	1.4MB/s	1.4MB/s	1.9MB/s

Table 5: Optimal data generation rate P for each workload in single/ multiple nodes environments.

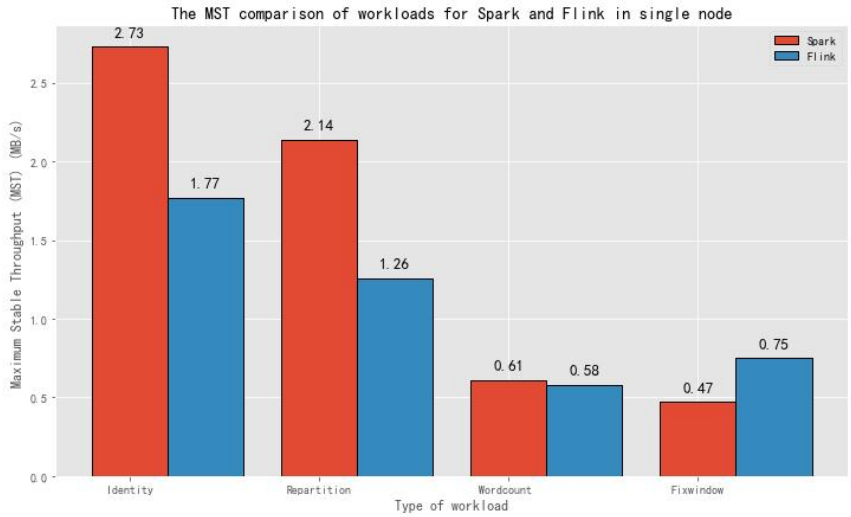
We measure the throughput and latency of the system as shown in Figure23 at the data generation rates shown in the table above. At this point, we are using the mean of 60 sample points. We noticed that the throughputs of the systems were all very close to (but less than) the rate at which the data was generated, suggesting that the systems are capable of long-term operation and are approaching the limits of performance.

Figure 23(a) shows the performance comparison of MST under a single node. For the empty (Identity) workload and the (Repartition) workload representing a single shuffle operation, Spark exhibits higher throughput than Flink, reaching 2.73MB/s and 2.14MB/s, respectively. Under these two workloads, the ultimate throughput of the system is 1.54 and 1.70 times that of Flink as a stream processing framework, respectively. This is mainly due to Spark's micro-batching approach. For more complex stateful computations, such as Wordcount, Flink and Spark show close MST, around 0.6MB/s. In addition, for the Fixwindow load representing window operations, Spark's throughput is weaker than Flink, only about 0.62 times the latter (reaching 0.47MB/s).

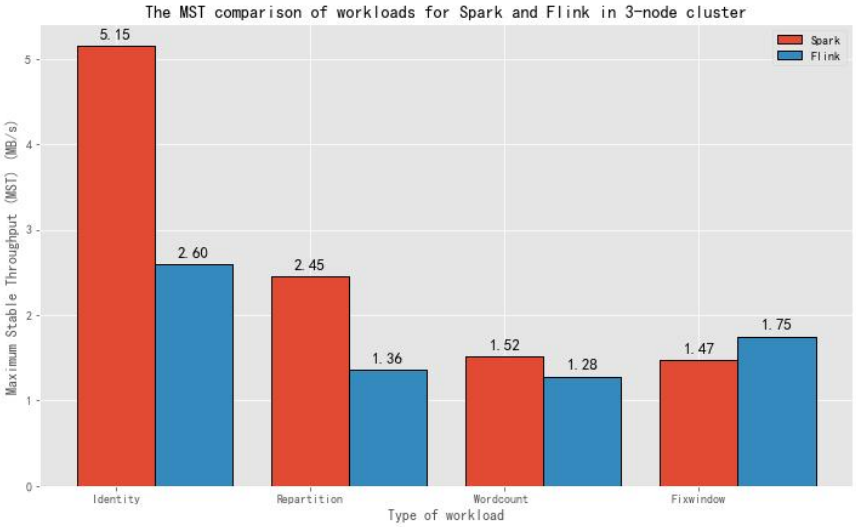
Figure 23(b) shows the throughput of a three-node cluster running the same load. For such a simple cluster, the throughput of all base loads exceeds 1MB/s. For Spark’s empty workload, the MST reaches a maximum of 5.15MB/s. Comparing Figure 23(a) and Figure 23(b), for all workloads, Spark and Flink improved, exhibiting similar relative MST as the cluster of nodes. This shows that both Spark and Flink hide the details of the cluster very well, three nodes can work like a complete system, and the performance mainly depends on the workload. In more detail, we can evaluate the horizontal scalability of the system. For a 3-node cluster, the controller node has eight cores, and the slave node has four cores. Each node has 4GB of memory. The relevant hardware configuration is shown in section4.1. Due to the uneven distribution of computing resources, we can only roughly evaluate the scalability of the stream processing framework on the cluster. For Identity workload, the throughput of Spark and

Flink increases by 1.88 and 1.46 times, respectively, in the baseline case. For Wordcount and Fixwindow, both perform more than 2x due to node increase.

It is worth noting that under the workload of Repartition, the system's scalability is not good, and the throughput of Spark and Flink is only improved by 1.14 and 1.07 times. We speculate that this may be because the network bandwidth becomes the performance bottleneck in the application, and the Repartition load is more dependent on network resources than memory and CPU resources.



(a) The MST comparsion for Flink and Spark for single node



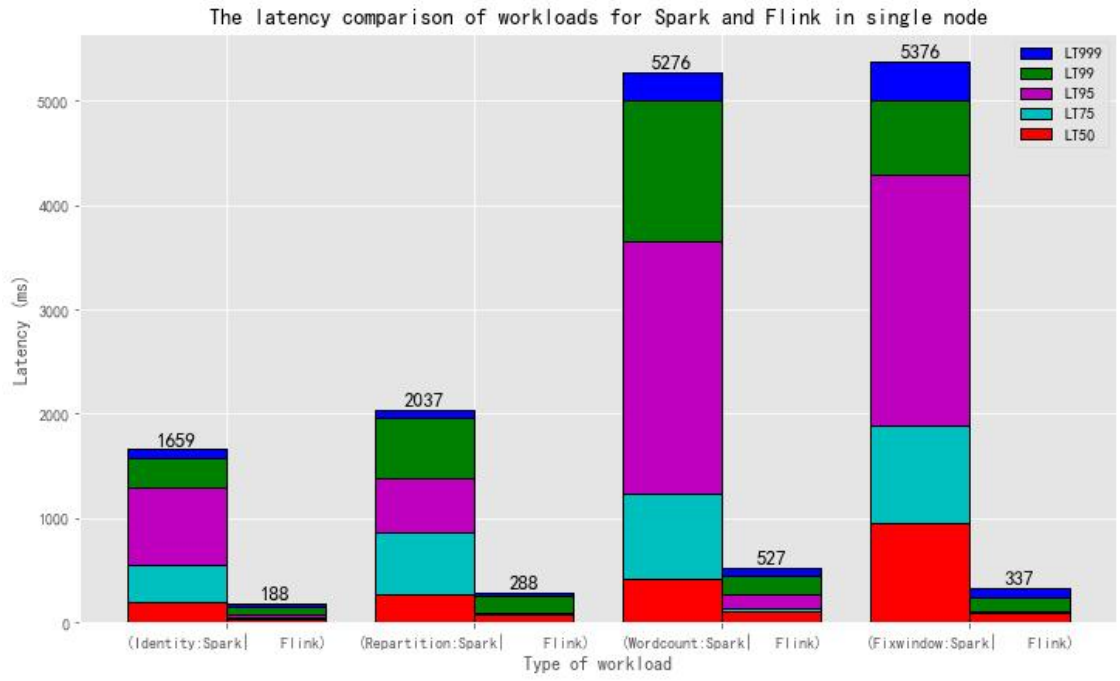
(b) The MST comparsion for Flink and Spark for 3-node cluster

Figure 23: The MST comparison of workloads for Spark and Flink in both single and 3-node cluster

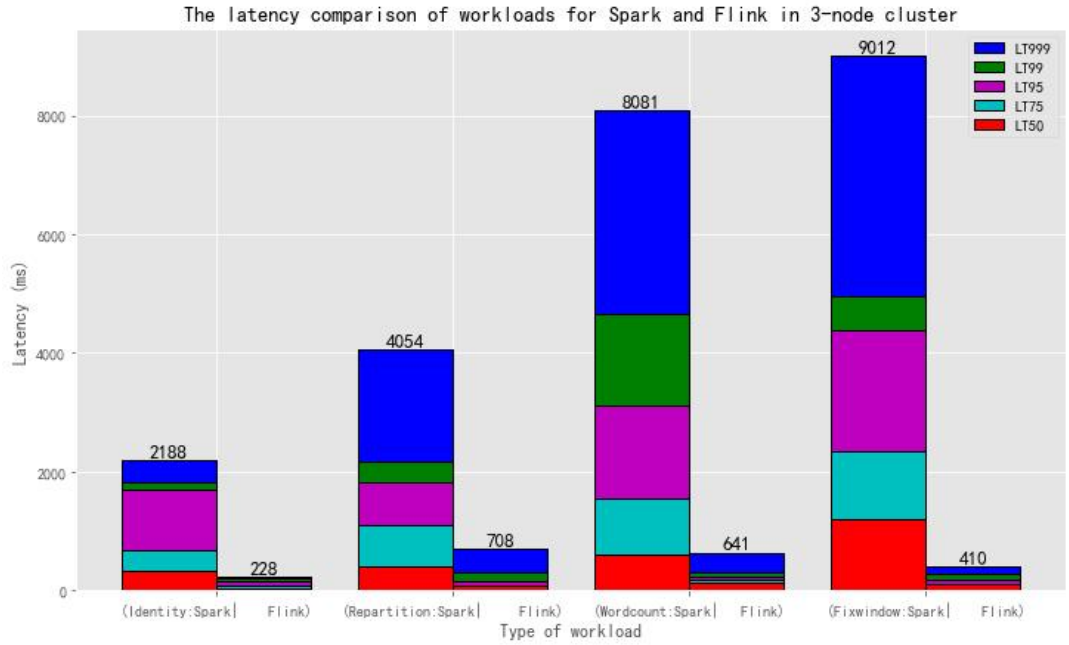
In our experiments, we also tested the quantile latency for each workload. Under the Optimal data generation rate P shown in Table, the delay information of Spark and Flink is shown in Figure24. Figure 24(a) shows the throughput performance under a single node. It is obvious that Flink have better latency performance; almost all operations can be completed within 0.5s. And Spark's LT999 latency is generally around ten times. Especially for window operations, the LT999 latency of Flink operations is only 6.26% of Spark. This may imply that Spark is not good at window operation processing. Note that Spark's latency is close to 5s in stateful operations (Wordcount and Fixwindow), which is related to the requirement that LT99 is less than 5s in our chosen criterion. This shows that the increase in latency causes the Spark application to be judged to be unstable.

Figure 24(b) shows the throughput performance under multiple nodes. In the Figure, we can still find that the latency of Spark is much higher than that of Flink. In the case of Identity, Spark's LT999 latency reaches 2188ms, which is 7.59 times that of Flink; in the case of Repartition and Wordcount, Spark's LT999 latency reaches 4054 and 8081ms, respectively, which is 5.72 and 12.6 times that of Flink; working in Fixwindow Under load, the performance gap between the two stream processing frameworks is the largest, close to 22 times.

Comparing Figures 24(a) and 24(b), we can find that the latency of each load is improved overall. The Optimal data generation rate P is more significant in a multi-node cluster, and the increased data pressure increases latency. LT999 (the blue part) has increased significantly, which may be caused and synchronized by data transmission between nodes.



(a) The lantency comparsion for Flink and Spark for single node



(b) The lantency comparsion for Flink and Spark for single node

Figure 24: The lantency comparison of workloads for Spark and Flink in both single and 3-node cluster

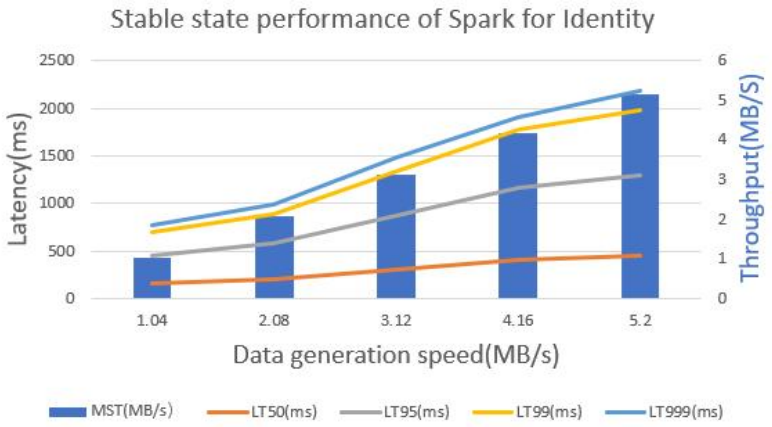
To sum up, for the experiments on the extreme computing performances of single and multiple computers, we can draw the following conclusions:

1. Under most workloads, Spark has a large throughput (Fixwindow is an exception), which is about two times that of Flink; Flink has apparent advantages in latency performance; almost all requests can be completed within 0.5s and only 5%-10% of Spark. Since Spark's processing method is micro-batch, and Flink's is stream processing, this result is basically in line with expectations.

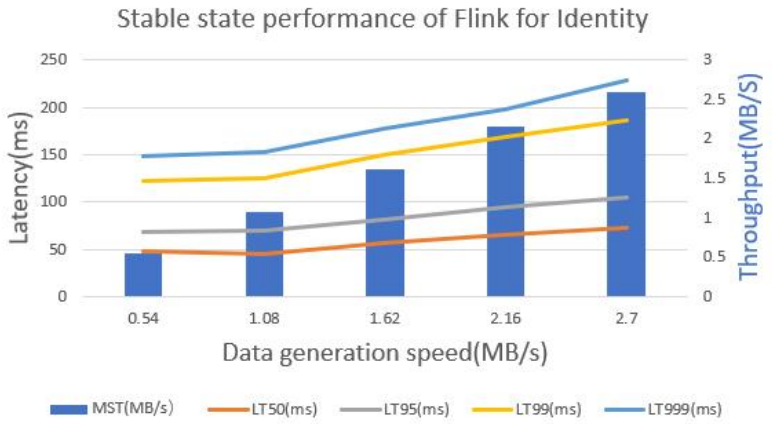
2. Stateful computing in Spark (Wordcount and Fixwindow) has a more significant impact on throughput. Precisely the Fixwindow operation might suggest that Spark is not good at Window operations. This may be related to Spark's state preservation. Spark requires additional RDDs to store intermediate data, while Flink's state management is more advanced.

3. For the two stream processing frameworks, the shuffle operation (Repartition) limits the scalability of the system throughput, while the scalability of CPU-intensive applications (Wordcount and Fixwindow) is stronger. This may be related to the communication method of the cluster, and the load scalability of network communication is weaker.

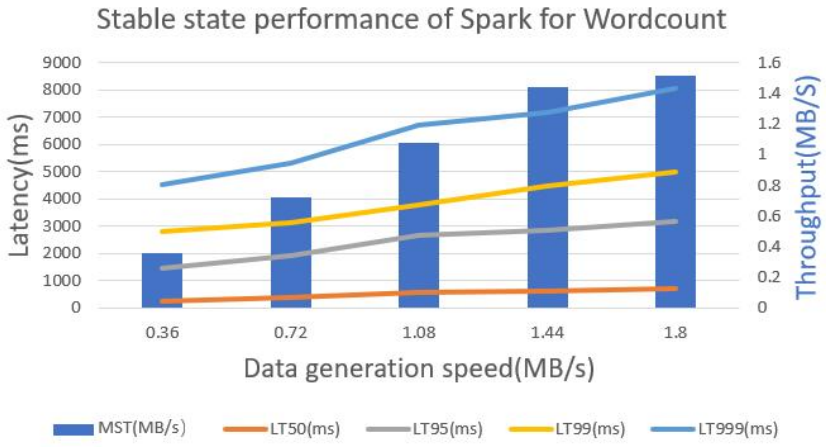
5.2 Stable state performance test



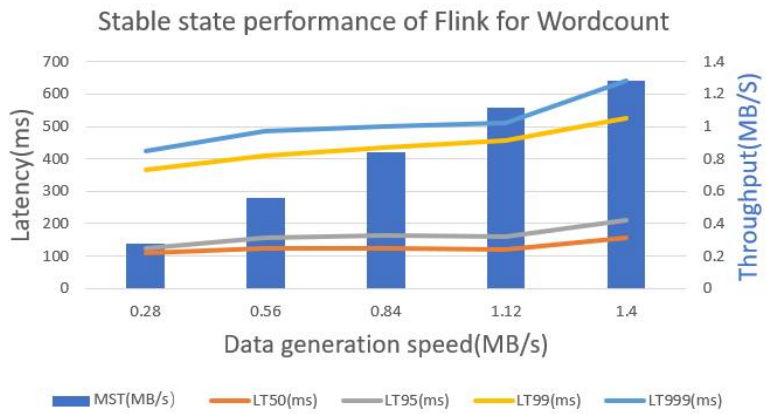
(a)Spark for Identity workload in stable state



(b)Flink for Identity workload in stable state



(c)Spark for Wordcount workload in stable state



(d)Flink for Wordcount workload in stable state

Figure 25: Throughput and latency as a function of data generation rate for Spark and Flink under Identity and Wordcount workload.

In this section, we explore how the throughput and latency of the system change with increasing data pressure at a steady state. Specifically, we need to measure throughput and latency under five data pressures with a data production rate of N*P/5 (N=1, 2, 3, 4, 5) under each workload. When N=5, data production rate will equal to Optimal data generation rate **P.**

As shown in Figure 25. The bars in the four graphs represent the throughput MST of the system, and the line shows the quantile delay of the system. Due to page limitation, we show the test results on a multi-node cluster with only Identity and Wordcount workloads.

Overall, in a steady state, the throughput that meets expectations is very close to the rate of data generation. The four graphs show similar trends, with both latency and throughput increasing with data pressure. For throughput, when in a steady state, the system's throughput is equal to the data generation rate. Since the increase of data pressure is a fixed value each time, the system's throughput increases linearly. This steady increasing trend will slow down at the critical point of system stability.

Flink's performance has a clear advantage for system latency in all cases， which is more obvious in the case of high data pressure. That is to say, the latency of Flink increases more slowly than Spark as the data pressure increases. Figure 25(a) and Figure 25(b) show the situation under the Identity workload; it can be seen that for LT99 latency, the latency ratio of Spark and Flink increased from 5 times under the initial data pressure (1.04 and 0.54MB/s) to 12 times the Optimal data generation rate P. Figure25(c) and Figure25(d) show the case of Wordcount. Similarly, the LT99 latency ratio of Spark and Flink goes from the initial seven times to the final thirteen times. Therefore, this data shows Flink has a more robust pressure tolerance in latency performance. In other words, in the case of less data pressure, the performance of Spark and Flink is closer. In addition,

we can also find that when the data pressure is not significant, the growth of the delay is relatively slow; when the data pressure is close to the Optimal data generation rate P, the growth of the delay is accelerated.

In summary, Flink's response time is shorter than Spark, but Flink latency increases more slowly than Spark as data pressure increases.

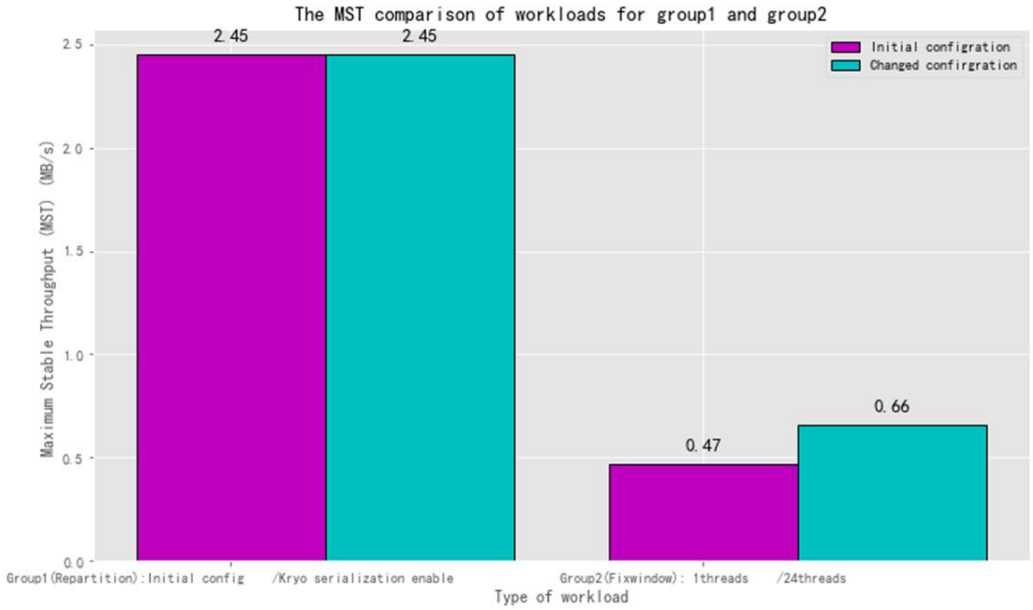
5.3 The impact of configurations

In this section, we discuss the performance impact of some simple configurations. We set up three experiments to show the effect of different configurations on extreme performance. Due to the time limitation of project, we only test Spark in Group1 and Group2. As shown in Figure26:

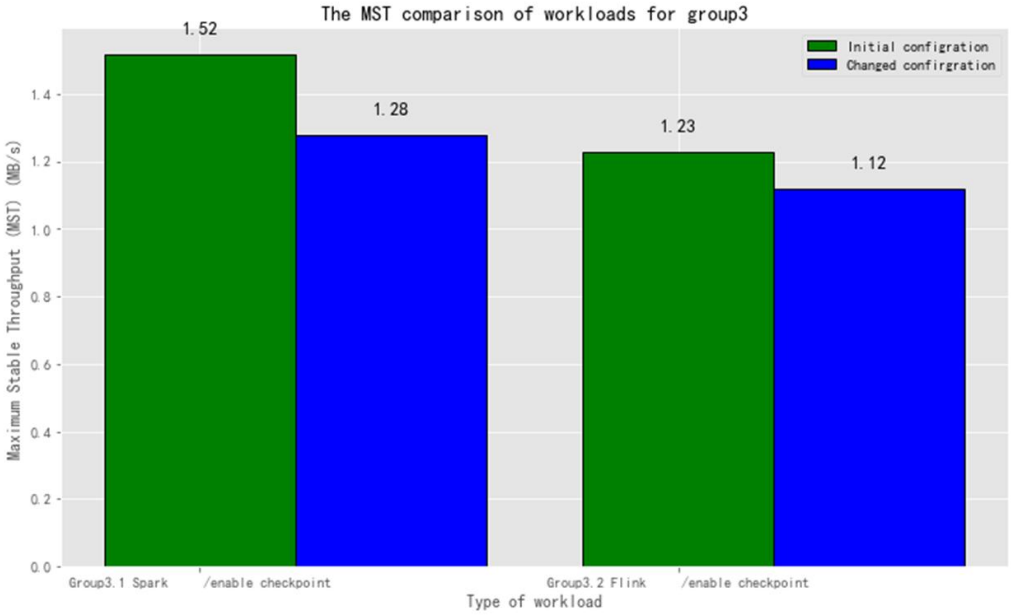
Group1. Repartition workload, the working cluster is on a multi-node cluster. The original Spark configuration is used on the left; the Kryo serialization is started on the right.

Group2. Fixwindow workload, the working cluster is on a single node cluster. The left side uses the original Spark configuration, the concurrency is the total number of cores (1threads). On the right side, Spark increases the concurrency by three times the number of cores (24threads) in local mode. Spark can change the default value by modifying Spark.default.parallelism[40].

Group3. Wordcount workload, the working cluster is on a multi-node cluster. The checkpoint mechanism is enabled for Flink and Spark, respectively. For figure, the original configuration has checkpoint disabled, which means the changed configuration(placed in right side in each group) is checkpoint enabled.



(a): Group1 and Group2



(b): Group3 for Wordcount workload

Figure 26: The impact of configuration on throughput for three groups.

From the Figure26(a), which repersents the results of group1 and group2. We can see that under the 3-node cluster, for the Repartition workload. Spark enabling Kryo serialization has no direct impact on performance; throughput and TP99 remain the same. We used local deployment mode in the single-node cluster. Spark enabling multithreading can increase the maximum throughput of Repartition workloads from 0.47 to 0.66MB/s. At the same time, the system's latency is also slightly reduced (not shown on the Figure26). The Figure26(b) shows that under Wordcount workload, the Flink and Spark cluster throughputs drop by 0.16 and 0.089 times, respectively. For Latency (not shown in the Figure26), both remain unchanged. This means that Flink has slightly less overhead than Spark for the checkpoint mechanism in stateful operations.

To sum up, for performance tuning methods methioned in experiment, it is valid to change MST and latency by changing configuration about stream processing framework.

5.4 Limitations and conclusion

In conclusion, we compared with Spark and Flink by HiBench. The advantage of Spark is higher throughput, while Flink's response time is significantly faster. When the data pressure is close to critical, Spark's MST is about twice that of Flink, and Flink's latency is less than 0.1 times that of Spark. Flink is better at stateful operations, especially window operations. In addition, CPU-intensive applications are more scalable than IO-intensive applications. Finally, some performance tuning operations are also verified to be effective. Overall, Flink is more convenient to develop and has an absolute advantage in latency performance. Therefore, when throughput is not the main requirement, it is recommended to use Flink to develop stream processing applications.

However, review the entire project, we find that the experiments have the following limitations:

1. The local virtual machine cluster is too small, and the discussion of scalability is generally not convincing.

2. Although a composite index is proposed to evaluate the system's stability with a set of parameters, its validity still needs to be tested.

3. The workload is relatively simple, and no real business is involved.

4. There is no profiling test about hardware, such as discussing CPU utilization, changes in network bandwidth.

Chapter 6 Project overview

6.1 Real progress

Here are some real progress during this project:

1. Developed and deployed some applications based on big data processing frameworks, including Hadoop, Spark, and Flink. Familiar with window operations and stateful operation development.

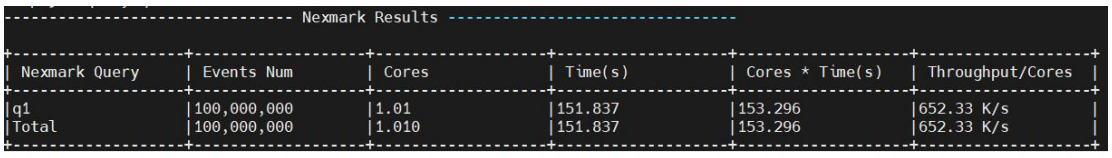
2. An overview of stream processing benchmarks is given. The architecture and design ideas of four standard open source stream processing benchmarks are discussed and compared.

3. A composite metric including latency and throughput is proposed. This indicator is used to evaluate the stability of the stream processing system and then measure the limit performance of the system. This method avoids relying too much on experience to measure the maximum throughput of the stream processing system to a certain extent.

4. On the local virtual machine cluster and supercomputing, HiBench was used to test the throughput and quantile delay of Spark and Flink in the extreme state and steady state. The performance under different workloads is analyzed and evaluated.

6.2 Problems and Future plan

To get larger scale cluster to verify the scalability, we have tried running benchmarks on supercomputing Archer2 and Cirrus to discuss the performance of streaming operations on HPC clusters. The benchmark we used was Nexmark. Unfortunately, this attempt only succeeded on a single node, as shown in Figure27. The main reason for the failure of the multi-node test is that SSL connection is temporarily not supported among the computing nodes of the supercomputing cluster. Many distributed applications in the stream processing framework need to use SSL to send and receive corresponding data, including node status information and the collection of performance on each node.



```
------------------------------ Nexmark Results ------------------------------

+------------------+------------------+------------------+------------------+------------------+------------------+
| Nexmark Query    | Events Num       | Cores            | Time(s)          | Cores * Time(s)  | Throughput/Cores |
+------------------+------------------+------------------+------------------+------------------+------------------+
|q1                |100,000,000       |1.01              |151.837           |153.296           |652.33 K/s        |
|Total             |100,000,000       |1.010             |151.837           |153.296           |652.33 K/s        |
+------------------+------------------+------------------+------------------+------------------+------------------+
```

Figure 27: Nexmark results of query 1 running on the Archer2

For the future plans of the project, we believe that we can continue to explore the application potential of the stream processing framework in supercomputing. Although stream processing engines are currently primarily designed to scale horizontally on cheap hardware, future developments may have higher performance requirements. In addition, we can expand the scope of this benchmark to developing more complex applications or testing more stream processing engines. What is more, we can also improve existing stream processing benchmarks. We can combine the strengths of different stream processing benchmarks, such as in data generators and advanced performance detection techniques, to build more modular next-generation benchmarks at the software level.

Reference

[1] O. Marcu, A. Costan, G. Antoniu and M. S. Pérez-Hernández, *Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks,* 2016 IEEE International Conference on Cluster Computing (CLUSTER), 2016, pp. 433-442, doi: 10.1109/CLUSTER.2016.22.

[2] Lee, Seungchul, Donghwan Kim, and Daeyoung Kim. *Comprehensive Performance Comparison Between Flink and Spark Streaming for Real-Time Health Score Service in Manufacturing.* Advances in Data Science and Information Engineering. Springer, Cham, 2021. 483-500.

[3] Online at https://flink.apache.org/usecases.html(referenced 19/08/2022)

[4] Online at https://github.com/Intel-bigdata/HiBench(referenced 19/08/2022)

[5] Khan, Nawsher, et al. *"Big data: survey, technologies, opportunities, and challenges."* The scientific world journal 2014 (2014).

[6] Anuradha, J. *"A brief introduction on Big Data 5Vs characteristics and Hadoop technology."* Procedia computer science 48 (2015): 319-324.

[7] Online at https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/(referenced 19/08/2022)

[8] Online at https://flink.apache.org/usecases.html(referenced 19/08/2022)

[9] Akidau, Tyler, et al. *"The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing."* (2015).

[10] Dean, Jeffrey, and Sanjay Ghemawat. *"MapReduce: simplified data processing on large clusters."* Communications of the ACM 51.1 (2008): 107-113.

[11] Samadi, Yassir, Mostapha Zbakh, and Claude Tadonki. *"Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks."* Concurrency and Computation: Practice and Experience 30.12 (2018): e4367.

[12] Online at https://nightlies.apache.org/flink/flink-docs-release-1.12/zh/concepts/index.html(referenced 19/08/2022)

[13] Online at https://www.51cto.com/article/542401.html(referenced 19/08/2022)

[14] Dessokey, Maha, et al. *"Importance of Memory Management Layer in Big Data Architecture."* International Journal of Advanced Computer Science and Applications 13.5 (2022).

[15] Karimov, Jeyhun, et al. *"Benchmarking distributed stream data processing systems."* 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018.

[16] Chu, Zheng, Jiong Yu, and Askar Hamdull. *"Maximum sustainable throughput*

evaluation using an adaptive method for stream processing platforms." IEEE Access 8 (2020): 40977-40988.
[17] Karakaya, Ziya, Ali Yazici, and Mohammed Alayyoub. "*A comparison of stream processing frameworks."* 2017 International Conference on Computer and Applications (ICCA). IEEE, 2017.
[18] T. Liu, Z. Yang and Y. Sun, *Docker Container Networking Based Apache Storm and Flink Benchmark Test,* 2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS), 2021, pp. 49-52.
[19] Qian, Shilei, et al. *"Benchmarking modern distributed streaming platforms."* 2016 IEEE International Conference on Industrial Technology (ICIT). IEEE, 2016.
[20] Guo, Yijin, et al. *GML: Efficiently Auto-Tuning Flink's Configurations Via Guided Machine Learning.* IEEE Transactions on Parallel and Distributed Systems 32.12 (2021): 2921-2935.
[21] Han, Rui, Lizy Kurian John, and Jianfeng Zhan. *"Benchmarking big data systems: A review."* IEEE Transactions on Services Computing 11.3 (2017): 580-597.
[22] Online at https://github.com/yahoo/streaming-benchmarks(referenced 19/08/2022)
[23] Chintapalli, Sanket, et al. *"Benchmarking streaming computation engines: Storm, flink and spark streaming."* 2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE, 2016.
[24] Online at https://github.com/dataArtisans/yahoo-streaming-benchmark(referenced 19/08/2022)
[25] Online at https://www.ververica.com/blog/extending-the-yahoo-streaming-benchmark(referenced 19/08/2022)
[26] Huang, Shengsheng, et al. *"The HiBench benchmark suite: Characterization of the MapReduce-based data analysis."* 2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010). IEEE, 2010.
[27] Huang, Shengsheng, et al. *"Hibench: A representative and comprehensive hadoop benchmark suite."* Proc. ICDE Workshops. 2010.
[28] Samadi, Yassir, Mostapha Zbakh, and Claude Tadonki. *"Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks."* Concurrency and Computation: Practice and Experience 30.12 (2018): e4367.
[29] Online at https://github.com/nexmark/nexmark(referenced 19/08/2022)
[30] Tucker, Pete, et al. *Nexmark–a benchmark for queries over data streams (draft).* Technical Report. Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.
[31] Online at https://beam.apache.org/documentation/sdks/java/testing/nexmark/(referenced 19/08/2022)
[32] Online at https://kafka.apache.org/documentation/(referenced 19/08/2022)
[33] Online at https://www.confluent.io/blog/configure-kafka-to-minimize-latency/(referenced 19/08/2022)
[34] Online at https://flink.apache.org/usecases.html(referenced 19/08/2022)
[35] Online at http://stratosphere.eu/(referenced 19/08/2022)
[36] Online at https://nightlies.apache.org/flink/flink-docs-release-1.10/ops/memory/mem_detail.html(

referenced 19/08/2022)
[37] Online at https://nightlies.apache.org/flink/flink-docs-master/zh/docs/concepts/flink-architecture/ (referenced 19/08/2022)
[38] Online at https://spark.apache.org/docs/latest/(referenced 19/08/2022)
[39] Online at https://spark.apache.org/docs/latest/cluster-overview.html(referenced 19/08/2022)
[40] Online at https://spark-reference-doc-cn.readthedocs.io/zh_CN/latest/more-guide/tuning.htm(referenced 19/08/2022)l
[41] Online at https://runtime.pw/flink-serialization-tuning/(referenced 19/08/2022)
[42] Online at https://www.confluent.io/blog/configure-kafka-to-minimize-latency/(referenced 19/08/2022)