A Performance Benchmark for Stream Data Storage Systems

Siqi Kang[1,2(✉)], Guangzhong Yao[1,2], Sijie Guo[3], and Jin Xiong[1,2(✉)]

[1] SKL Computer Architecture, ICT, CAS, Beijing, China
{kangsiqi,yaoguangzhong,xiongjin}@ict.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] StreamNative, Beijing, China
guosijie@gmail.com

Abstract. Modern business intelligence relies on efficient processing on very large amount of stream data, such as various event logging and data collected by sensors. To meet the great demand for stream processing, many stream data storage systems have been implemented and widely deployed, such as Kafka, Pulsar and DistributedLog. These systems differ in many aspects including design objectives, target application scenarios, access semantics, user API, and implementation technologies. Each system use a dedicated tool to evaluate its performance. And different systems measure different performance metrics using different loads. For infrastructure architects, it is important to compare the performances of different systems under diverse loads using the same benchmark. Moreover, for system designers and developers, it is critical to study how different implementation technologies affect their performance. However, there is no such a benchmark tool yet which can evaluate the performances of different systems. Due to the wide diversities of different systems, it is challenging to design such a benchmark tool. In this paper, we present SSBench, a benchmark tool designed for stream data storage systems. SSBench abstracts the data and operations in different systems as "data streams" and "reads/writes" to data streams. By translating stream read/write operations into the specific operations of each system using its own APIs, SSBench can evaluate different systems using the same loads. In addition to measure simple read/write performance, SSBench also provides several specific performance measurements for stream data, including end-to-end read latency, performance under imbalanced loads and performance of transactional loads. This paper also presents the performance evaluation of four typical systems, Kafka, Pulsar, DistributedLog and ZStream, using SSBench, and discussion of the causes for their performance differences from the perspective of their implementation techniques.

Keywords: Stream data storage system · Performance benchmark · Performance evaluation

W. Gao et al. (Eds.): FICC 2020, CCIS 1385, pp. 107–122, 2021.
https://doi.org/10.1007/978-981-16-1160-5_10

1 Introduction

Nowadays Internet companies are generating tremendous amount of data everyday. They keep recording various events all the time. For example, social media platforms keep recording state changes of their users; online shopping platforms keep recording transactions as well as clicking events of their users; data centers keep recording machine states, including CPU, memory, network and disk. These data could contain great value. Through efficient processing of these data, the companies can recommend preferences for users, formulate more accurate advertising strategies for businesses, or propose more efficient solutions for data center resource scheduling. Therefore, stream processing platforms are widely used to analyze real-time data quickly and efficiently [24–27]. The characteristics of stream processing are real-time, scalability and openness [1]. The processing objects of stream processing are data streams. Writers can continuously append data to the data stream, and the data stream can simultaneously accept multiple readers to start reading data from a certain position in the data stream. In general, stream data has the following characteristics [2]:

- Real-time: Stream data are generated in real time, and each piece of data describes in detail what happened.
- Continuous flow: Stream data are generated ceaselessly, that is, the applications always generate new data without stopping.
- Append-only: New data are written into the data stream in an append mode, and no modification operation will be performed on the existing data.
- Sequentiality: Stream data are generally appended to the data stream in chronological order.

Since the continuously generated stream data contains rich value, it is important to persist them for later processing using streaming or batch queries. To support various stream processing scenarios, stream data storage systems should provide high read and write performance. Many such storage systems are developed either in academia or industry to store stream data. However, each of them only satisfies some of the characteristics of stream data. Moreover, these systems have different interfaces, functions, technologies, and performance.

According to their technologies, existing stream data storage systems can be divided into two categories: *pub-sub systems* and *shared log systems*. Typical pub-sub systems include Kafka [3], Pulsar [4], RocketMQ [5], etc. They abstract their data as a *topic*, and provide a publish/subscribe interface to the topic. They can provide fast read and write to stream data by using technologies including partitioning, sharding, multi-layer architecture and replication. Distributed log systems, such as Corfu [6], vCorfu [7], Tango [29] and BookKeeper [8], abstract their data as *logs*, and provide read and write interfaces to logs. They emphasis on data consistency guarantees by using the transaction technology and a global sequencer.

Due to the wide diversities of stream data storage systems, each system only uses its own performance measuring tool for evaluation. Currently, it is impossible to evaluate different systems using the same benchmark tool. Therefore, it is

difficult for users to choose the best-performing system according to their needs. However, to design a benchmark tool for performance evaluation of different systems is non trivial, because they have different design goals, access interfaces, semantics, and technologies. There is no such benchmark tool yet which can evaluate these systems.

In this paper, we present SSBench, a performance benchmark tool designed for evaluate stream data storage systems. SSBench abstracts the data and operations in different systems as "data streams" and "reads/writes" to data streams. By translating stream read/write operations into the specific operations of each system using its own APIs, SSBench can evaluate different systems using the same loads. In addition to measure simple read/write performance, SSBench also provides several specific performance measurements for stream data, including end-to-end read latency, performance under imbalanced loads and performance of transactional loads. Moreover, as a case study, we use SSBench to evaluate four typical systems, Kafka, Pulsar, DistributedLog [14] and ZStream [28], and analyze the causes for their performance differences from the perspective of their implementation techniques.

The paper is organized as follows. Section 2 provides an overview of stream data storage systems, including the differences between typical systems. Section 3 describes SSBench, the performance benchmark tool we designed and implemented. Section 4 presents using SSBench to systematically compare the performance of different systems. Section 5 discusses related work, and Sect. 6 concludes the paper.

2 Overview of Stream Data Storage Systems

In this section, we first discuss typical application scenarios and their requirements for stream data storage systems. Then, we summarize the critical technologies of existing stream data storage systems. And finally, we briefly survey several typical stream data storage systems, highlighting the differences between them.

2.1 Typical Application Scenarios

Several typical application scenarios of stream processing are listed as follows.

- **Website activity tracking.** In the LinkedIn social platform [9], it is necessary to record the information that each user searched for and followed. By using machine learning models, it can predict social connections, match users with suitable positions, and optimize advertising. By analyzing the events of each person clicking on the company and position, we can show users some similar job opportunities. This requires the data platform to quickly analyze and process the user's click event data in real time, and quickly make recommendations for users in a short time.

- **Monitoring data.** In the traffic management platform [10], The platform evaluates and monitors real-time congestion and accidents by monitoring GPS and SCATS sensor data. In addition, it also needs to monitor the content posted by local social platform to determine traffic conditions. In the network flow monitoring scenario [11], network monitoring data (such as TCP data packets sent and received within a certain period of time) are usually generated in the form of streams, which need to be continuously analyzed and monitored.
- **Social network platform.** In the Twitter open social platform scenario [12], users can send tweets with a certain hashtag, and users can click on a certain hashtag to view all the tweets under that topic. The real-time tweets sent by the user are appended to the data stream. The data stream is partitioned and partitioning rule is to hash the hashtags. This calculation method makes the tweets of the same hashtag be sent to the same partition. When some hot events occur, hot topics will appear. There will be a large number of read and write requests for a certain topic within a certain period of time. In a scenario where topics are used as the partition standard, hot partitions will be caused. Some partitions have less access, and some partitions have more access, which will cause data skew and access skew.
- **Commit log.** The system log needs to record all the operations of the system. It is a very important data source and contains many values. In LinkedIn [13] and Twitter [14], the stream data storage system is used as the write ahead commit log of the Espresso [30] and Manhattan databases [31] to record the modification operations on the database. Batch processing, stream processing, or the operation of data warehouses can analyze submission logs and extract important values.
- **Distributed transaction.** In the scenario of bank transfer [15], using the above-mentioned stream data storage system as the commit log of the balance database needs to ensure the correct execution of distributed transactions. There cannot be an intermediate situation where only one operation succeeds and another operation fails.

2.2 Requirements

The various scenarios involved in stream processing put forward several requirements for the storage of stream data:

- **Fast reading and writing.** It can quickly store a large amount of detailed data generated in real time, and can support fast reading for real-time data, so that the data can be quickly analyzed.
- **Efficient analysis.** Stream data has no semantics, and the stream processing system cannot only extract key fields for query analysis. If the data storage itself can be semantically aware, and the processing system can quickly extract important fields, query efficiency can be greatly improved.
- **Load balancing.** Stream data storage systems usually use partitions to increase concurrency. Partitions will inevitably cause hot spots. Stream data storage systems should load balance hotspot partitions to reduce the writing and reading of hot partitions and servers.

– **Distributed transactions.** For certain scenarios of stream data storage systems, it is required to provide support for distributed transactions and support the atomic execution of multiple write operations.

2.3 Critical Technologies

Based on the requirements of stream data storage, the key issues that the stream data storage system mainly solves are as follows:

– **Fast reading and writing.** Stream data records the detailed data that occurred at the time of recording. Storing the stream data is essential for data analysis and processing. Stream processing requires that stream data can be read and written quickly to support efficient stream processing analysis. Therefore, fast storage of stream data and provision of fast access are the most critical technologies for stream data storage systems.
– **Efficient analysis query support.** Stream data is required to support low-latency stream processing requirements as well as efficient query and analysis requirements. Therefore, the data transfer from non-semantic-aware stream data into column data stream data and the performance guarantee of low latency and high bandwidth have become challenges for stream data.
– **Automatic load balancing.** Load balancing needs to determine the partitions and nodes with heavier loads accurately in order to migrate the heavier loads. Therefore, determining the time point of load balancing trigger and the location of load migration is the focus of load balancing work. And it is also very important to ensure the correctness of load migration and correctness after failure recovery in the process of load balancing.
– **Efficient distributed transaction support.** Stream data storage systems should address the need for atomic completion of multiple write operations in storage. Distributed transactions need to solve the problem of isolation of uncommitted transaction messages. The characteristics of data streams cause the stream data storage system to only support append-only writes. For reads, users can read stream continuously from a certain position. Therefore, if we mix ordinary data, committed transaction data, and uncommitted transaction together, it is difficult to guarantee the isolation and transactional reading and writing will interfere with the reading and writing of ordinary events, which is a key point in distributed transactions.

2.4 Typical Systems

Pub-Sub Systems. Kafka is the most classic publish-subscribe messaging system. Kafka increases read and write throughput through partitioning in parallel [3]. In terms of read, sendfile technology is used to reduce multiple copies of data between kernel mode and user mode. Since the data in Kafka is appended to the topic in a non-semantic sense, the data cannot be quickly analyzed. For the load balancing of hot data, load migration will cause the migration of old data and waste cluster resources. The advanced version of Kafka has supported

distributed transactions. The mixed storage of ordinary messages and transaction messages can cause some problem. Short transactions must wait for long transactions to be submitted before their data can be seen by users. Therefore, this transaction isolation method is less efficient.

Pulsar is a relatively new publish-subscribe messaging system. Its data abstraction and interface are similar to Kafka. Pulsar also uses partitioning to increase the throughput of reads and writes [4]. Similarly, Pulsar does not provide a semantic-aware storage format and cannot perform efficient analysis and processing of data. In Pulsar, load balancing is also considered, but Pulsar's load balancing stays in the service layer. In addition, Pulsar does not yet support the distributed transaction feature, which does not meet all our requirements for stream data storage systems.

Shared-Log Systems. In Corfu [6] and vCorfu [7], the global log is responsible for ensuring consistency, global order, and transactionality, and the virtual materialized flow increases scalability. But in essence, every write must firstly go through the global log before being written to the partitioned stream. This shared log uses a centralized sequencer, and its throughput is limited by the speed at which the sequencer allocates address space. FuzzyLog [16] is based on the shortcomings of Corfu/vCorfu. At the cost of strict order, it supports the causal order of each additional operation rather than a global order by establishing a directed acyclic graph of events. But FuzzyLog is not orderly for each partition. In some strictly orderly scenarios, there may be errors.

The distributed log storage system BookKeeper [8] is a log stream service that provides persistent storage. It uses striping and read-write separation mechanism to provide high performance. However, BookKeeper does not support semantic awareness for storing data, and cannot provide cross-ledger transactions, which does not meet all our requirements for stream data storage systems. DistributedLog [14] is a distributed log warehouse. It uses log stream to represent the continuous data stream. It uses BookKeeper as a storage component. So DistributedLog has the same problem as BookKeeper.

ZStream [28] is a stream data storage system that divides a stream into more fine-grained partitions, called ranges. ZStream asynchronously converts row stream data into column stream data, and supports efficient columnar reading performance. By dynamically splitting and merging the partitions, the imbalanced load of the partitions is solved. And the transaction buffer technology solves the problem of distributed transaction isolation and the interference of transactional requests to ordinary requests.

3 Design of SSBench

In order to compare the performance of systems, a general performance evaluation tool is needed. The results of the evaluation of each system under the same dimension, unified environment and unified test conditions can be comparable. As far as we know, there is still a lack of such a performance evaluation tool for

stream data storage systems. So we designed and implemented a unified stream data storage system benchmark tool called SSBench, which abstracts based on the common characteristics of each system. SSBench aims at evaluating the performance of different systems. It encapsulates the read and write operations of stream data that is separate from the specific system, and supports multiple read-write mode test scenarios. In addition, we also considered the scalability of SSBench itself to facilitate the evaluation of adding new systems.

3.1 Architecture and Functions

Architecture. The system architecture of SSBench is shown in Fig. 1. The top layer is the user interface. By providing users with a simple and easy-to-use operation interface, users can input simple test parameters and specific system commands to define test scenarios, start test programs, and obtain test results.

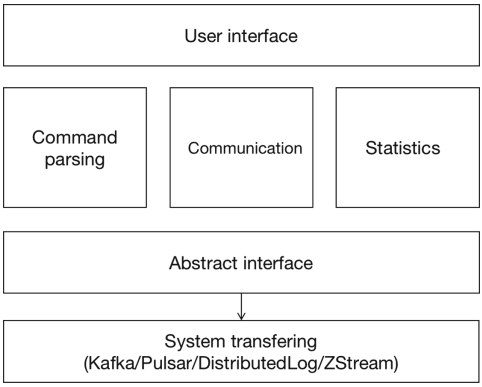

Fig. 1. Architecture.

The command parsing module determines the number of read and write client processes to start, the specific settings of each read and write task, and creates client processes on each test node by parsing user commands. The communication module is responsible for the communication between client processes. It includes the distribution of read and write tasks on multiple test machines, the maintenance of client life status, and the unified collection of test data. The statistical data module is responsible for collecting the performance of each client process in the process of performing read and write tasks. When the performance information generated, results will be output to the console and files.

In addition, SSBench also provides a more flexible evaluation method. Users can start a single client process through commands, and use scripts to start multiple client processes. Therefore, in multi-client read and write tasks, different parameters can be set for different client processes to achieve more flexible evaluation.

The abstract interface layer provides a unified read and write interface (send/read) for the stream data storage system, with the functions of sending and receiving messages.

Unified Interface. For the two major types of stream storage systems, the data abstraction and writing unit are different. In general, the message system data is abstracted as a topic, and the distributed log is a log stream. So we abstract these operating objects as streams. Reader reads streams and writer writes to streams respectively. Users of SSBench only need to specify how to read and write and does not need to care about how the specific system reads and writes.

For write operations of different stream data storage systems, the client can write one message at a time and return the result asynchronously. In addition, some systems can also write synchronously, that is, when writing a message, the writing process is blocked, waiting for the returned result before writing the next message. Therefore, we abstract a unified client-side write operation interface. For each system, you can choose to write synchronously or asynchronously, write one message at a time, and continuously loop to support continuous writing.

For read operations, different systems read slightly differently. For example, the Kafka consumer adopts the pull mode. Each read operation will pull a batch of data to read. The amount of a batch of data is determined by time. The default setting is 100ms. While the reading of Pulsar and DistributedLog needs to be obtained through asynchronous monitoring. Once a new message is available, then it returns the received new message. Due to the difference between the two methods, our framework does not support the read-by-item interface similar to the write interface. Instead, each system implements the function of continuously reading data. Therefore, we abstract a unified read interface, and each system reads data in its own way, returns the read results asynchronously, and the unified read interface completes the collection of performance results.

Therefore, for a specific stream data storage system, you only need to implement the above abstract interface, and you can use this framework for testing. So SSBench can support a unified test platform and compare the differences between systems horizontally.

Functions. For the design, we refer to the framework structure of YCSB (Yahoo! Cloud Serving Benchmark) [18]. In order to test the performance of different databases, YCSB abstracts the addition, deletion, modification, and query interfaces. Specific databases need to implement these interfaces, while the test scenarios are implemented by YCSB, such as database performance under different mixing ratio operations. In SSBench, we use the following functions to fairly test different systems:

1) Given a list of available client nodes, users of SSBench can specify the number of clients, that is, how many write processes and how many read processes. Then the test framework will evenly start all write (or read) processes on given nodes. Therefore, the write (or read) process can run on different machines, so SSBench is a distributed performance evaluation tool.

2) For the writing and reading process, the status of writing or reading, such as throughput and latency, is displayed in the form of a window (fixed time) and the running time can also be configured.

3) For the writing process, users of SSBench can specify the data format to be sent, such as fixed size, random content, etc. They can also specify the sending rate, such as fixed rate, unlimited rate, gradual rate, and random rate. In particular, some stream data storage systems support distributed transactions, so for the write process, users can specify whether it is a transaction write request, as well as transaction-related sending interval, number of sent and other parameters.

4) For the reading process, the user of SSBench can specify where to start reading, such as reading from the beginning or reading from the end. The former is to read from the oldest data, and the latter is to read from the latest data.

3.2 Common Read/Write Performance

For ordinary reading and writing, different systems use different technologies to ensure fast reading and writing. For example, for the replication mechanism, Kafka adopts the leader-follower model for multi-copy replication. First, a partition is sent to a leader node and then the followers pull data from the leader. Compared with Pulsar and DistributedLog, BookKeeper provides quorum replication mechanism. Data will be sent to three Bookies at the same time, and it only needs to receive most Bookie's confirmation so that the message can be known as persisted. Based on different copy strategies, we hope to understand the differences in write performance of different systems in their respective modes.

Therefore, for ordinary read and write performance evaluation, this paper evaluates the difference in write performance of different systems in the synchronous write mode, and summarizes the impact of different replication mechanisms based on the read and write performance of stream data storage systems.

3.3 Column Read/Write Performance

For column read and write evaluation, this paper mainly evaluates the difference in read and write performance of different systems in a multi-field scenario. Users can customize the number of event fields, field types, the number of fields of each type, and the number of read and write client processes to evaluate the impact of different field numbers on the performance of different stream data storage systems.

3.4 Imbalanced Load Performance

SSBench can simulate unbalanced load scenarios. User-defined evaluation of the number of streams and the ratio of write speed in each stream can verify read and write performance under load balanced and load imbalanced conditions. For ZStream, due to its own implementation of load balancing, this paper uses SSBench to adjust ZStream to a better performance state by setting different load balancing parameters of ZStream.

3.5 Transactional Load Performance

Distributed transactions simulate distributed transaction scenarios in stream data storage. Users can set the number of streams involved in each transaction, the number of data written in each stream, and the interval between transaction sending to evaluate the performance of distributed transactions. For long transactions, users can set the duration of long transactions and the proportion of long transactions in all transaction requests to evaluate the impact of long transaction requests on performance.

4 Performance Evaluation of Typical Systems

For the evaluation of reading and writing of ordinary messages, Sect. 4.1 uses the synchronous writing method to evaluate the write throughput of Kafka, Pulsar and DistributedLog when writing multiple copies. For column storage characteristics, Sect. 4.2 uses multi-event data sets to compare and evaluate ZStream and Kafka. For the load balancing feature, the evaluation results of ZStream, Kafka and Pulsar are compared in the case of balanced and imbalanced loads in Sect. 4.3. For distributed transactions, Sect. 4.4 evaluates the comparison results of ZStream and Kafka.

This paper uses a cluster platform with five nodes interconnected, each of which is configured with two Intel Xeon E5645 CPUs (each CPU has 12 hyper-threaded cores). The nodes are connected by 10 Gigabit Ethernet. The stream data storage system cluster uses 4 nodes, one of which serves as the master node to start the NameNode (for HDFS) or ZooKeeper (for ZStream, Kafka, and Pulsar) services. The other four are used as servers or clients. Storage nodes use SSDs as data storage.

4.1 Common Read/Write Performance

In this experiment, SSBench creates a write process, which creates a stream, and writes messages to this stream synchronously at a rate of 100000 msg/s (each message size is 1KB).

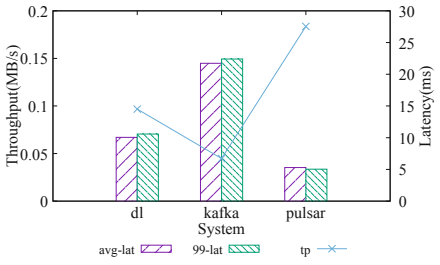

Fig. 2. Write performance in sync mode.

The results are shown in Fig. 2. It can be seen that Pulsar has the lowest and most stable write latency, followed by DistributedLog, and Kafka is the worst, which are about 2 times and 3 times that of DistributedLog and Pulsar, respectively. Kafka also has the worst write throughput, reaching only 30% of Pulsar and 50% of DistributedLog. The reason is Kafka is a two-step serial replication. First, a message is sent to the leader node, and the two follower nodes pull data from the leader node. While Pulsar and DistributedLog are one-step parallel replication, they send a message to all replica nodes. DistributedLog and Pulsar use BookKeeper to store data persistently, and the message will be sent to multiple Bookie nodes at the same time. They only need to receive the confirmation returned by most Bookie to consider that the message has been persisted in all Bookie nodes. Since the Bookie node uses SSD as the log disk, data can be written sequentially at high speed, so the write latency of DistributedLog and Pulsar is low.

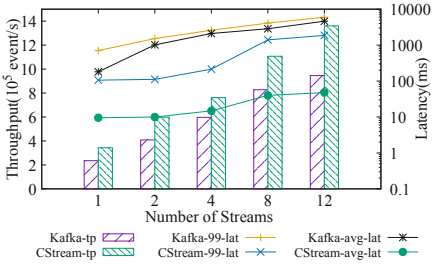

Fig. 3. Write performance. Each event has 6 fields and totally 38 bytes.

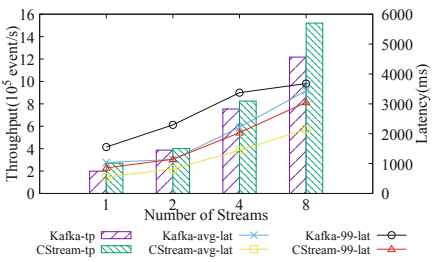

Fig. 4. Read performance. Each event has 6 fields and totally 38 bytes.

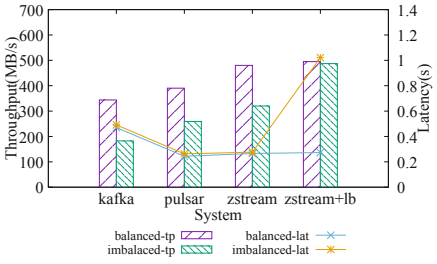

Fig. 5. Write throughput under balanced and imbalanced load.

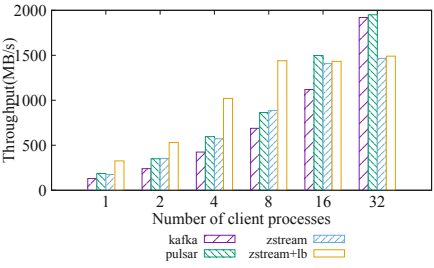

Fig. 6. Read throughput under imbalanced load.

4.2 Column Stream

In this experiment, SSBcnch creates a stream, and creates a read process and a write process. The writing process is responsible for writing multi-field data

to the image system. Each event includes six fields, including types Int, Long, Float, Double, Boolean, and String. The total length of each event is 38 bytes, to verify the difference in read and write performance.

Figure 3 shows the write performance. It can be seen from the figure that the write performance of ZStream is significantly better than that of Kafka. The difference in write performance of Kafka is mainly due to its leader-follower replication strategy. ZStream uses BookKeeper's Quorum mechanism, which leads to low latency of writing. Figure 4 shows the read performance. When the number of streams increases, the end-to-end latency of both ZStream and Kafka changes. Because of its lower write latency, the end-to-end latency of ZStream is also lower than that of Kafka. In addition, due to its high network utilization, it also shows higher throughput.

4.3 Load Balance

This experiment compares ZStream with Kafka and Pulsar under load balancing and load imbalancing conditions. For ZStream, we first test the situation when load balancing is not enabled, and then enable the load balancing for evaluation.

Figure 5 shows the comparison of the write throughput. It can be seen that for the three systems, the write performance will suffer a 30%–50% loss under imbalanced load condition. Because when the load is imbalanced, the write speed of a single stream is limited by the speed of a single thread to write data. For the load balancing situation with ZStream+lb (ZStream enabled load balancing), the partition will not be triggered for load balancing, so the performance is consistent with the original ZStream load balancing situation. In the case of load imbalancing, the partition with large traffic can be divided into two sub-partitions, so writing increases the degree of parallelism, it can show higher write performance. Pulsar also supports load balancing, but Pulsar only migrates the partitions with large traffic to the lighter load machine, and does not split the large traffic. Therefore, during the experiment, there will always be a partition with large traffic, which is constantly migrating between the two data nodes, but the throughput has not been improved.

Figure 6 shows how the read throughput of the three systems varies with the number of client processes when the load is imbalanced. As can be seen in the figure, ZStream with load balancing enabled increases in concurrent reads due to splitting, and can maintain a performance advantage of 30% to 50% when the number of read processes is small, and is limited to the ZStream reader when the number of read client processes is large. Performance disadvantages are shown without aggregation processing and multi-process sharing.

4.4 Distributed Transactions

In this experiment, a write client process continuously sends write requests and transaction requests with four read client processes read data in real time. Transaction requests are sent every 100 ms.

Figure 7 and 8 show the end-to-end latency with the write time of each long transaction and the proportion of long transactions. The end-to-end latency changes relatively smoothly with the long transaction execution time and the proportion of long transactions. In Kafka, as the long transaction request time and the proportion of long transactions increase, the end-to-end latency is on the rise. For Kafka, reading of ordinary messages has to be blocked by uncommitted long transactions. Especially when the long transaction takes a heavier part and the execution time of the long transaction is long, most of the real-time message reading will be in a blocking state, so Kafka shows poor end-to-end latency. ZStream uses transaction buffers. When long transactions are not committed, ordinary messages and short transaction messages can still be read in real time. Only long transaction data has a long end-to-end latency, and the overall performance is 70% to 90% better than Kafka.

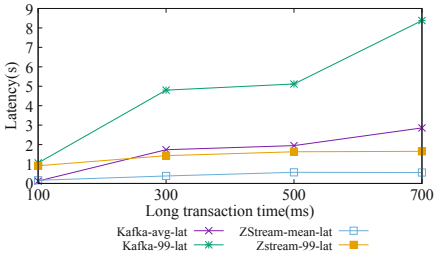

Fig. 7. End-to-end latency of different long transaction time settings.

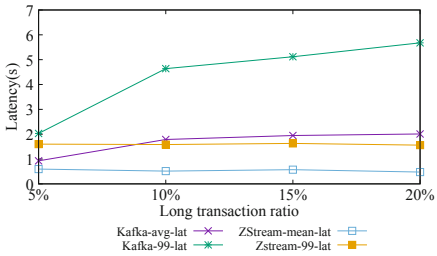

Fig. 8. End-to-end latency of different long transaction ratio settings.

5 Related Works

5.1 Benchmarks for Storage Systems

fio is an IO test tool that can run on a variety of systems [19]. It can test the performance of local disks, network storage, etc. fio simulates different loads by specifying the type of I/O to be tested, configuring I/O size, block size, engine, I/O depth and other parameters to simulate different loads to verify storage performance under different loads. Filebench [20] is an automated testing tool for file system performance. It tests the performance of the file system by quickly simulating the load of a real application server. It can not only simulate micro-operations (such as copyfiles, createfiles, randomread, randomwrite), but also simulate complex applications (such as varmail, fileserver, oltp, dss, webserver, webproxy). Filebench is more suitable for testing file server performance, but it is also an automatic load generation tool, and can also be used for file system performance.

The goal of the YCSB project is to develop a frame work and a set of common workloads to evaluate the performance of different key value stores and

cloud service storage systems [18]. Users can install different database systems in the same hardware environment, use YCSB to generate the same workload, and compare and evaluate different systems. YSCB is to test the performance of different databases through abstract addition, deletion, modification, and query methods. Specific databases need to implement these methods, while the test scenarios are implemented by YCSB, such as database performance under different mixing ratio operations.

YCSB++ is an extension of YCSB to improve the testing of advanced database functions [21]. YCSB++ includes multi-tester coordination for load increase and eventual consistency measurement, multi-stage workloads for quantifying the consequences of work delays and the benefits of expected configuration optimization (such as B-tree pre-splitting or batch loading), and high-level features in abstract API benchmarks for explicit consolidation.

We learns a lot from YCSB's method. Similar to YCSB, SSBench abstracts the basic interfaces such as read/write APIs for the basic operations, and each specific stream data storage system only need to implement these methods. In addition, SSBench also generates specific workloads such as imbalanced loads and transactional loads, and enables users to evaluate their system on more scenarios.

5.2 Benchmarks for Data Processing Systems

In order to help users choose the most suitable platform to meet their big data real-time stream processing needs, Yahoo has designed and implemented a stream processing benchmark program based on real-world scenarios and released it as an open source [22]. In this benchmark test, Kafka and Redis were introduced for data extraction and storage to build a complete data pipeline to more closely simulate actual production scenarios. The benchmark test platform scenario for stream processing is different from the storage scenario, and does not meet the needs of the storage system evaluation in the paper.

OLTP-Bench is a scalable DBMS benchmark test platform [23]. The main contribution of OLTP-Bench is its ease of use and scalability, support for transaction mixing, strict control of request rate and access allocation, and the ability to support all major DBMS platforms. In addition, it also bundles 15 workloads with varying complexity and system requirements, including 4 comprehensive workloads, 8 workloads from popular benchmarks, and 3 jobs from real applications load. The OLTP evaluation evaluates the transaction characteristics of the database management system. It has a single function and does not meet the multiple requirements of the storage system in the stream processing scenario.

6 Conclusion

This paper implements a benchmark tool based on stream data storage system called SSBench, and uses it to evaluate and compare the performance of multiple stream data storage systems. Our experimental results show that for common

read and write operations, different replication mechanisms and caching strategies can bring great performance differences. For column data read and write, unbalanced load read and write, and distributed transaction scenarios, ZStream has 30% to 90% advantages over the other systems.

References

1. Shahrivari, S.: Beyond batch processing: towards real-time and streaming big data. Computers **3**(4), 117–129 (2014)
2. The log: What every software engineer should know about real-time data's unifying abstraction. https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying. Accessed 16 Dec 2013
3. Kreps, J., Narkhede, N., Rao, J., et al.: KAFKA: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7. IEEE (2011)
4. Francis, J., Merli, M.: Open-sourcing pulsar, pub-sub messaging at scale, 8, September 2016. https://www.richardnelson.it/post/150096199100/open-sourcing-pulsar-pub-sub-messaging-at-scale
5. ALIBABA: How to support more queues in rocketmq? 23 September 2016. https://rocketmq.apache.org/rocketmq/how-to-support-more-queues-in-rocketmq/
6. Balakrishnan, M., Malkhi, D., Prabhakaran, V., et al.: CORFU: a shared log design for flash clusters. Proceeding of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2012), pp. 1–14, USENIX (2012)
7. Wei, M., Tai, A., Rossbach, C.J., et al.: vCorfu: a cloud-scale object store on a shared log. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI17), pp. 35–49 (2017)
8. Junqueira, F.P., Kelly, I., Reed, B.: Durability with bookkeeper. ACM SIGOPS Oper. Syst. Rev. **47**(1), 9–15 (2013)
9. Goodhope, K., Koshy, J., Kreps, J., et al.: Building Linkedin's real-time activity data pipeline. IEEE Data Eng. Bull. **35**(2), 33–45 (2012)
10. Panagiotou, N., et al.: Intelligent urban data monitoring for smart cities. In: Berendt, B., et al. (eds.) ECML PKDD 2016. LNCS (LNAI), vol. 9853, pp. 177–192. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46131-1_23
11. Bär, A., Finamore, A., Casas, P., et al.: Large-scale network traffic monitoring with DBstream, a system for rolling big data analysis. In: 2014 IEEE International Conference on BigData (Big Data), pp. 165–170. IEEE 2014
12. McCreadie, R., Macdonald, C., Ounis, I., et al.: Scalable distributed event detection for Twitter. In: IEEE International Conference on Big Data, pp. 543–549. IEEE (2013)
13. Wang, G., Koshy, J., Subramanian, S., et al.: Building a replicated logging system with Apache Kafka. Proc. VLDB Endowment **8**(12), 1654–1655 (2015)
14. Sijie, G., Robin, D., Leigh, S.: DistributedLog: a high performance replicated log service. In: IEEE 33rd International Conference on Data Engineering. ICDE, pp. 1183–1194 (2017)
15. Mehta, A., Gustafson, J.: Transactions in Apache Kafka[EB/OL], 17 July 2017. https://www.confluent.io/blog/transactions-apache-kafka/
16. Lockerman, J., Faleiro, J.M., Kim, J., et al.: The FuzzyLog: a partially ordered shared log. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018), pp. 357–372 (2018)

17. Kreps, J.: Benchmarking Apache Kafka: 2 million writes per second (on three cheap machines), April 2014. https://engineering.linkedin.com/Kafka/benchmarking-apache-Kafka-2-million-writes-second-three-cheap-machines
18. Cooper, B.F., Silberstein, A., Tam, E., et al.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing, pp. 143–154. ACM (2010)
19. fio. http://freshmeat.sourceforge.net/projects/fio
20. Tarasov, V., Zadok, E., Shepler, S.: Filebench: a flexible framework for file system benchmarking. USENIX Login **41**(1), 6–12 (2016)
21. PatilS, P.M.: YCSB++: benchmarking and performance debugging advanced features in scalable tablestores. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, Cascais, Portugal, vol. 9 (2011)
22. Chintapalli, S., Dagit, D., Evans, B., et al.: Benchmarking streaming computation engines: storm, flink and spark streaming. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1789–1792. IEEE (2016)
23. Difallah, D.E., Pavlo, A., Curino, C., et al.: OLTP-bench: an extensible testbed for benchmarking relational databases. Proc. VLDB Endowment **7**(4), 277–288 (2013)
24. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K.: Storm@twitter. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 147–156. ACM (2014)
25. Kulkarni, S., Bhagat, N., Fu, M., et al.: Twitter heron: stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 239–250. ACM (2015)
26. Carbon, P., Katsifodimos, A., Ewen, S., et al.: Apache flink: stream and batch processing in a single engine. IEEE Data Eng. Bull. **38**(4), 28–38 (2015)
27. Venkataraman, S., Panda, A., Ousterhout, K., et al.: Drizzle: fast and adaptable stream processing at scale. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 374–389. ACM (2017)
28. Shen, Y., Yao, G., Guo, S., et al.: A unified storage system for whole-time-range data analytics over unbounded data. In: Proceedings of 2019 IEEE Intl Conference on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking, pp. 967–974. IEEE (2019)
29. Balakrishnan, M., Malkhi, D., Wobber, T., et al.: Tango: distributed data structures over a shared log. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, pp. 325–340. ACM (2013)
30. Qiao, L., Surlaker, K., Das, S., et al.: On brewing fresh espresso: LinkedIn's distributed data serving platform. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1135–1146. ACM (2013)
31. Manhattan, our real-time, multi-tenant distributed database for Twitter scale, 2 April 2014. https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html