Pravega Performance Analysis and System Implementation

January 5, 2022

Abstract

Recently, the streaming data processing has been a pressing need for lots of real-world applications, like social media, and evolving tons of stream processing methods. The mainstream system for solving this problem is Kafka. However, it cannot handle every problem well, for example, like long term data retention or partition auto-scaling, etc. To solve those architecture-level problems that Kafka has failed to solve, a new stream system Pravega emerges. In this project, I benchmark the performance of Pravega and Kafka based on different event sizes, partition numbers and data persistence options. After measuring the throughput and latency in experimental environment, I also design and implement a simple stream data processing system based on Twitter Developer API. With my implementation, the system could handle 30% of Twitter data in real world on a single machine and prove that Pravega has the ability to deal with practical problems.

1 Introduction

Streaming data is one of the most novel and challenging types of data nowadays, as tons of new data are generated every second. And the data comes from everywhere, websites, social medias, sensors, IoT devices, Cloud - and the list goes on. The other concept that are usually compared with streaming processing is batch processing. For batch system, the data is collected over time and when processing, it's ready to use. The stream, on the other hand, is particularly challenging because the data continuously generates and never comes to an end. As the data types and format differentiates over years, traditional storage system could not handle streaming data processing problem well. As a result, new kinds of storage system emerge, like Kafka, Pulsar, etc.

Distributed messaging systems such as Kafka[8] and Pulsar have provided modern Pub/Sub infrastructure well suited for today's data-intensive applications. However, it seems like they don't perform well on long term data rentation or indigestion of large data size. In order to improve the performance of messaging systems in all perspectives, another open-source project Pravega has been proposed. Pravega further enhances this popular programming model and provides a cloud-native streaming infrastructure. It solves architecture-level problems that former topic-based systems Kafka and Pulsar have failed to solve, such as auto-scaling of partitions or maintaining high performance for a large number of partitions.

In addition to benchmark Pravega and Kafka with meaningless auto-generated data in test environment. I also apply for Twitter Developer API in order to design and implement a simple real-time stream data analytic system. The system provides the opportunity for users to know current hot and popular topics as well as searching over the related Tweets that they are interested. By using Pravega as the message middle-ware and storage, the system could handle 30% of Twitter data in real-world on single machine.

2 Related Works

Big Data Technology in Processing. In big data technology, Apache Spark[10] is well known and the most commonly used platform. It is a lightning-fast cluster computing technology, designed for fast computation. Hadoop[1] is the foundation of Spark. It extends the MapReduce[3] model in Hadoop to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application. The stream processing modules in spark are spark streaming and structured streaming[5]. By comparing, the structured streaming is used as calculating method, due to its well optimized data structure, storage methods, end-to-end guarantees and etc.

Distributed Data Storage. Besides the processing unit in big data, the distributed database system is also important for scalability. Mostly used NoSQL databases for solving large-scale distributed storage problems are Redis, Cassandra, MongoDB, HBase, Hdfs[6], etc.[7].These database systems are designed based on different principles, according to the CAP theory. At present, almost all NoSQL choose consistency or availability on the basis of maintaining partition fault tolerance. For example, HBase sacrifices partial availability in exchange for complete consistency, and Cassandra, which is similar to HBase, sacrifices strong consistency in exchange for a guarantee of availability.

Kafka Benchmark Tools. Nowadays there are specialized tools to test Kafka's performance like Pepper-Box JMeter plugin or OpenMessaging. And Kafka has been compared with several messaging systems like Pulsar, RabbitMQ[4]. As supplementary info, Kafka has a built-in test framework named Trogdor, that can be used to perform fault injection or to execute workloads for a test system[2]. Apache Kafka also has its own benchmark tool which could be used in command lines, such as kafka-producer-perf-test and kafka-consumer-perf-test. With the help of these measurements, proper configuration will be tuning for specific circumstances.

Contribution. This paper benchmarks the performance of a emerging messaging system Pravega and mainstream messaging system Kafka in theoretical environment first and then implements a simple real-time streaming data analytic system which using Pravega as message middleware, to find out the actual performance of Pravega in practical scenario. This large system consists of many modules and lots of time are devoted in system design and implementation.

- Pravega outperforms Kafka in throughput as well as processing latency in most situations theoretically.
- Deploy Pravega into the system for load balance and overcome the peak burst of data generation.
- Scalable for all the components, message middle-ware, streaming data processing module, cloud-native vector-based database and backend handler.
- Did research for each components, make sure they are suitable for each other and achieve the state-of-the-art performance in their specific domain.
- Utilize Structured Streaming over spark streaming to get benefit from SQL engine optimization.
- The system can process the 30% of Twitter real-world data and generate current hot topics in real-time.

3 Benchmark of Pravega and Kafka

Performances of data insertion and reading differ from the key features of the incoming data as well as storage layout. By using the performance evaluation tools of Pravega and Kafka, the ability of address specific circumstances will be displayed clearly. A series of comparison test will be included: throughput on the size of each data event, throughput on the number of storage partitions and effect of data long term retention, etc.

The most accurate way to model use case is to simulate the load expected on the experimental machine. In this case, I choose the kafka-perf-test tools and pravega-benchmark tools as the measurement criteria.

In general, Performance tuning involves two important metrics: latency which measures how long it takes to process one event and throughput which measures arrive within specific amount of time. As a result, these two factors are what should be focused in the experiments.

3.1 Event Size

First, I analyze the how the latency and throughput vary according to the size of one message. There are several percentage latency display by the measurement tool, 50th latency, 95th latency, 99th latency, 99.9th latency and max latency. I choose the 95th latency as the comparison criteria, which means that how long will it take when 95% of messages are processed.

```
for RECORD_SIZE in 1 10 100 1000 10000
do
    echo "Record Size $RECORD_SIZE:" >> kafka_pperf_1M_size.txt
    kafka-producer-perf-test \
    --topic test\
    --num-records 1000000\
    --throughput -1\
    --producer-props bootstrap.servers=localhost:9092\
    batch.size=1000 acks=1 linger.ms=100000  request.timeout.ms=300000\
    --record-size $RECORD_SIZE >> kafka_pperf_1M_size.txt
done
```

Figure 1: Config of Kafka Benchmark

```
for EVENTS_SIZE in 1 10 100 1000 10000
do
    ./run/pravega-benchmark/bin/pravega-benchmark  \
    -controller tcp://127.0.0.1:9090  \
    -stream test \
    -producers 1 \
    -segments 1 \
    -size $EVENTS_SIZE \
    -throughput -1  \
    -events 1000000 >> pravega_pperf_1M_size.txt
done
```

Figure 2: Config of Pravega Benchmark

As the Fig. 1 and Fig. 2 show, the other parameters except event size are the same: 1 million events sent in total, only 1 partition or segment store the data, etc.

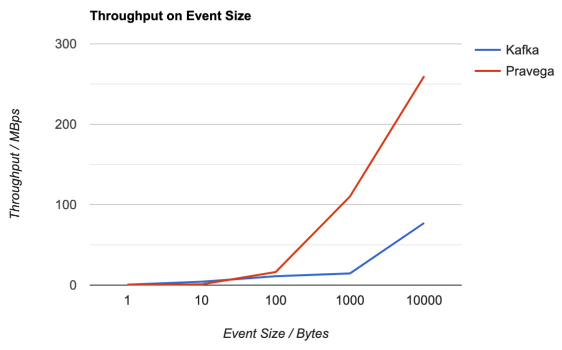


Figure 3: Throughput on Event Size

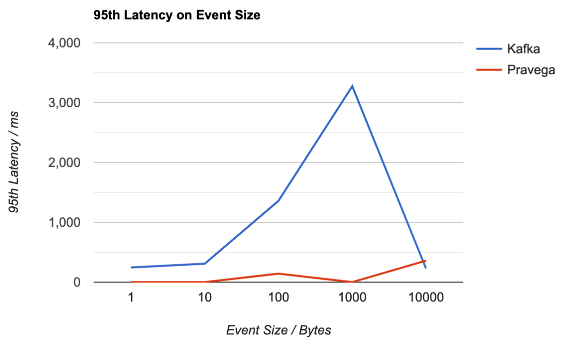


Figure 4: 95th Latency on Event Size

From Fig. 3 we can conclude that Pravega and Kafka's throughput increase correspondingly when event size in the range of 1 Bytes to 10K Bytes. And out of my expectation, the performance of Pravega in throughput outperform Kafka when event size is greater than 100 Bytes, however, Kafka does better when event size is small enough.

From Fig. 4 something interesting is revealed. At most of time, Pravega has lower 95th latency than Kafka, while they perform nearly the same when event size is around 10KBs. Neither of Kafka and Pravega's line chart are monotonically increasing, Pravega reaches lowest latency when event size is around 1KB and Kafka has its best performance when event size is about 10KBs.

3.2 Partition Number

For small events(About 100 Bytes), using multiple parallel segments in Pravega can provide low latency at low throughput, while using a single segment can achieve low latency at high throughput. For large events (10,000 Bytes), writing multiple segments (350MB/s) can achieve higher throughput than a single segment (260MB/s).

3.3 Long Term Retention

Long-term storage is one of the key components of Pravega's architecture, and Pravega will adjust the overall performance of the system based on long-term storage. Due to this feature, the user can use same paradigm to access both real-time and historical events stored in Pravega. The performance and correctness of long term retention is crucial for a storage system, so 95th latency and throughput are still used as the measurement criteria for evaluating how system perform with different data persistence options.

By comparing Pravega's performance after turning on and off the log flush function on the persistent log, we can infer that how Pravega's data persistence feature acts. With the same method, I compare Kafka in the two cases of enabling and disabling the persistence feature.

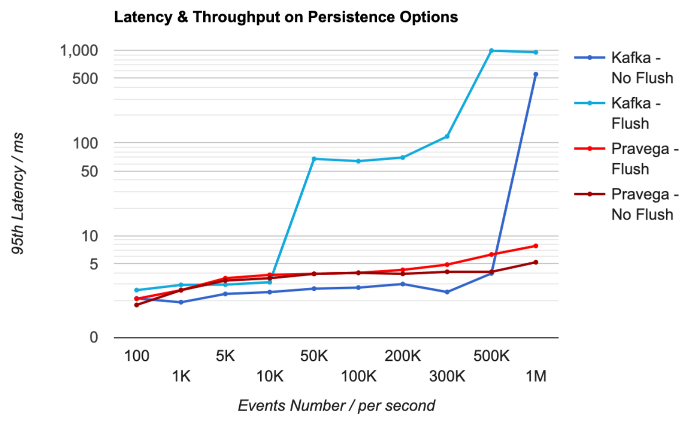


Figure 5: Latency & Throughput on Persistence Options

From Fig. 5 you can find that for stream of a single partition, Pravega with flush achieves a maximum throughput which is 80% higher than Kafka without flush. And setting Pravega to prevent BookKeeper from flushing data to the persistent log doesn't have performance gains, which proves the reasonableness of Pravega's default persistence features.

4 Practical System Design

4.1 System Overview

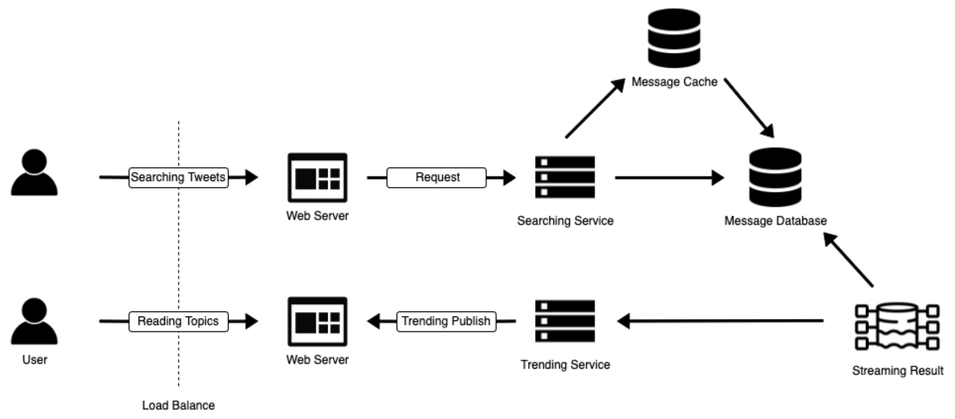


Figure 6: System Overview

After benchmarking the performance of Pravega in experimental environment, I decide to implement a simple real-time streaming data processing system with Twitter Developer API, to figure out whether it performs well in practical application and reaches my expectation. Fig. 6 shows the simple overview of how users interact with the system. User could browse real time hot topics and search related tweets. The requests load sending to web server will be balanced by Nginx. The trending service will publish topics ranking word cloud to web server periodically. By sending search request, the searching service will access message database as well as message cache for results.

4.2 Data Flow Pipeline

In this section, I will demonstrate the data flow pipeline and introduce the streaming part in detail, which is nearly transparent in user view.

As Fig. 7 shows, the entire pipeline of this system which includes 5 parts in total, Data Storage section, Pravega Message Buffer section, Spark Streaming Data Processing section, Customized Function section and Web Server section.

At the very beginning, it is the raw Twitter data as the message producer of the system, which is the real-time blurred Tweets generated by Twitter Developer Streaming API. It will be stored in Json file first for permanent back up and also be sent into Pravega as messages. Through Pravega, the Twitter data will be passed to spark for pre-processing. After the data is processed, it will be sent back into a new stream of Pravega. In this way, different phases of data could be accessed simultaneously.

A copy of the result will be stored into the database, and also transferred to message consumers, which are the two functions in this system. The trending function will only use the result from Pravega & Spark directly, because the hot topics in real world change quickly, so there is no need to store the ranking. The searching function has to access database for original tweets and milvus[9] for tweets vector information. Redis is implemented as the cache of the system to improve the data locality. After the first time some content is searched by the user, the result of searching will be stored into Redis temporarily and expire automatically after a period of time. The system also includes a simple web server for visualization, which uses Vue as the front-end framework and flask as the back-end framework.

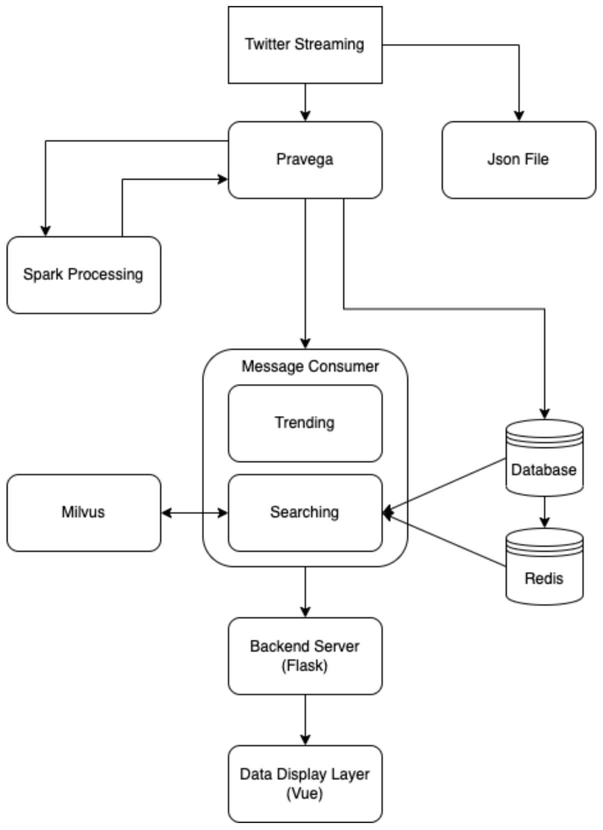


Figure 7: Pipeline including all stages

5 Module Details

5.1 Pravega

Pravega is implemented as the message middle-ware in this system. It is a distributed publish & subscribe messaging system, which is used in real-time streaming data architectures to provide real-time analysis. And there are three main reasons to use it.

Decoupling. By using Pravega, it is easy to expand the system no matter in the upstream layer or downstream layer. Currently in this system, there is only one producer - the Twitter streaming data API, and two consumers - the trending function as well as the searching function. However with Pravega, all these parts are developed independently. Other data sources could be added into this system as new message producer, like Facebook, Instagram, Pinterest, etc. And customized functions could be implemented as message consumers without the need to modify the rest of the system. By decoupling producers and consumers, they can be developed, deployed and scaled separately.

Cushion. Pravega is effectively a durable cache that buffers the unprocessed messages, providing a cushion for the system to handle peak load or consumer failure. When there is a burst of upstream Twitter data, the downstream scripts may not handle it in time. So by using Pravega, a fixed consuming rate can be set which is acceptable for the entire system. Now the developer streaming API provides 1% of real-time Twitter data. As I measured, the average of Tweets per second can be accessed is about 70 and actually the system could handle this volume easily.

Scalability. Pravega could run as a cluster on one or more servers that can span multiple nodes in AWS. The partitions of the data are distributed over the servers in Pravega cluster with each serve handling data and accessing request for a share of the partitions. And each partition is replicated across a configurable number of servers for fault tolerance. One server acts as the leader which handles all read and write requests for the partition and zero or more servers are followers which passively replicate the leader. Each server acts as a leader for some of its partitions and also a follower for others so that the workload is balanced within a cluster successfully and automatically.

In addition, with only one common machine as the server node, Pravega could deal with nearly millions of write request per second. By adding more nodes into the Pravega cluster, the fault tolerance will be promised and the data processing speed will be higher regardless of the overhead of internal communication.

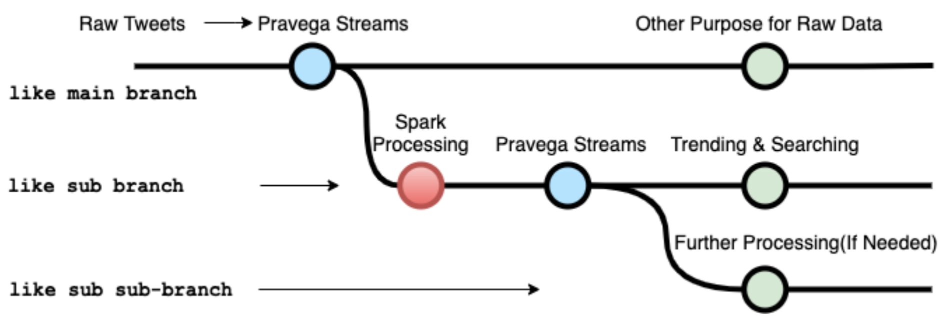


Figure 8: Streaming Data Flow in Pravega

In this system, two forms of data are flushed into Pravega, one is the original twitter data and the other is pre-processed data. The former one has to be stored into database for future use and transfer to Spark for pre-processing. The latter one is the data processed by Spark and will be consumed by applications, which are hashtag ranking & related Tweets searching in this system. With publishing different period of processing streaming into different topics in Pravega, the data in one single stream will be processed only once. Moreover those data could be used just like the fork function does. Every phase of the streaming is similar to a key component in a pipeline, but more easy to establish and assemble.

5.2 Back-end and Data Display server

To facilitate the user, I set up the front-end using Vue for UI display and build the back-end server using Flask, as shown in Fig. 9.

The back-end server receives HTTP requests from different modules, like data display module, vector storage computing module, data storage module and data feeding module.

The data display module consists of two parts, one is trending section, the other is the searching section, as shown in Fig. 10. The trending topic for hashtag is shown in word cloud images. The bigger the word is, the hotter the topic is. The section below is the search area, each response consists of the date posted, the message content and the distance with the query text.

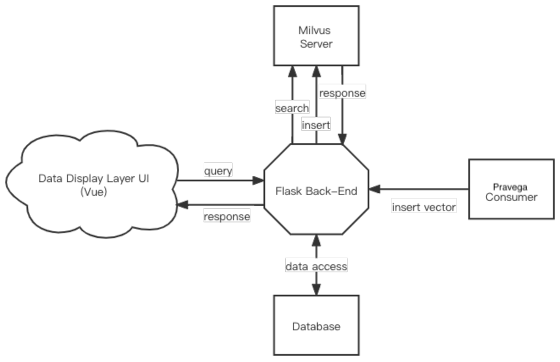
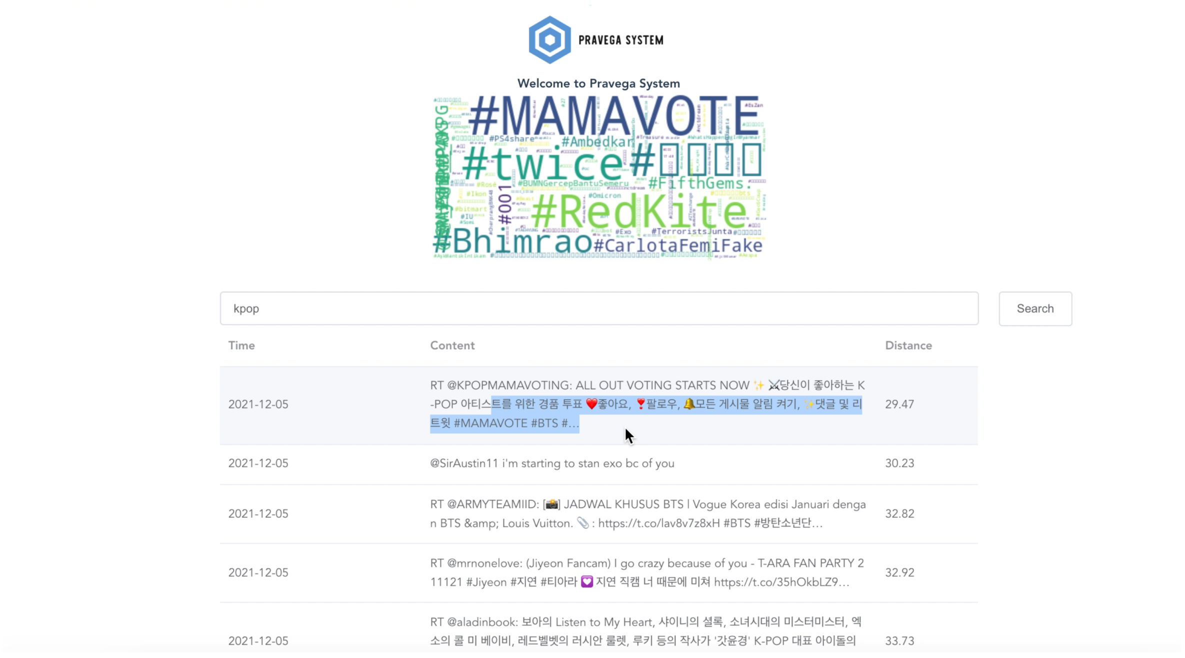


Figure 9: Back-End and Data Display server



Time	Content	Distance
2021-12-05	RT @KPOPMAMAVOTING: ALL OUT VOTING STARTS NOW ✨ ⚔당신이 좋아하는 K-POP 아티스트를 위한 경품 투표 ❤️좋아요, ❣️팔로우, 🔔모든 게시물 알림 켜기, ✨댓글 및 리트윗 #MAMAVOTE #BTS #…	29.47
2021-12-05	@SirAustin11 i'm starting to stan exo bc of you	30.23
2021-12-05	RT @ARMYTEAMIID: [📸] JADWAL KHUSUS BTS l Vogue Korea edisi Januari dengan BTS &amp; Louis Vuitton. 📎 : https://t.co/lav8v7z8xH #BTS #방탄소년단…	32.82
2021-12-05	RT @mrnonelove: (Jiyeon Fancam) I go crazy because of you - T-ARA FAN PARTY 211121 #Jiyeon #지연 #티아라 💟 지연 직캠 너 때문에 미쳐 https://t.co/35hOkbLZ9…	32.92
2021-12-05	RT @aladinbook: 보아의 Listen to My Heart, 샤이니의 셜록, 소녀시대의 미스터미스터, 엑소의 콜 미 베이비, 레드벨벳의 러시안 룰렛, 루키 등의 작사가 '갓윤경' K-POP 대표 아이돌의	33.73

Figure 10: Data Display Layer

5.3 Twitter Streaming API

Twitter provides developers with API to generate and pass the data (real-time, streaming, and w/o intentional). It provides three tracks now, for more data to access, developer has to apply for the higher plan. In general, Twitter streaming API provides us with different kinds of queries. The most related one to this project is the sampled streaming data, which can access 1% of the real-time data users posted on Twitter. Besides, other useful queries like filtered stream, and likes, retweets lookup, are also popular. The response of the query will generate endless Json string and form the stream.

6 Evaluation

To see the demo for the entire system, check the link System Demo. Generally the system processes 1% Twitter real world data in real-time and responses to search queries in milliseconds. Due to the limited bandwidth of Twitter streaming data API, artificial data has to be generated to test the maximal workload tolerance of whole system.

The stress testing is conducted on a Macbook Pro(2018 Version), which is equipped with a 2.6 GHz 6-Core Intel Core i7 CPU and 16 GB 2400 MHz DDR4 Ram. Fig. 11 illustrates the maximum

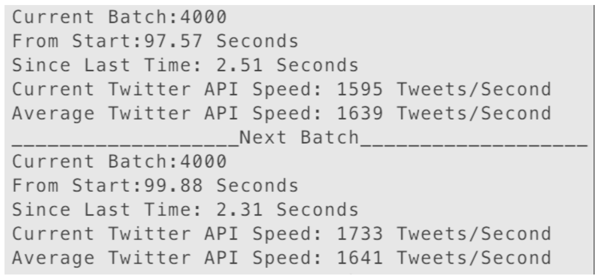

```
Current Batch:4000
From Start:97.57 Seconds
Since Last Time: 2.51 Seconds
Current Twitter API Speed: 1595 Tweets/Second
Average Twitter API Speed: 1639 Tweets/Second
___________________Next Batch___________________
Current Batch:4000
From Start:99.88 Seconds
Since Last Time: 2.31 Seconds
Current Twitter API Speed: 1733 Tweets/Second
Average Twitter API Speed: 1641 Tweets/Second
```

Figure 11: Stress Test

workload this system could handle, about 1600 Tweets per second, which is corresponding with nearly 30% of real world Tweets.

7 Conclusion

In this work, I compare the performance of two messaging systems first - Pravega and Kafka. Found that Pravega outperforms Kafka in throughput as well as processing latency in most situations theoretically.

The throughput of Pravega is higher than Kafka when event size is greater than 100 Bytes, however, Kafka does better when event size is small enough.

Neither of Kafka and Pravega's latency line chart are monotonically increasing, Pravega reaches lowest latency when event size is around 1KB and Kafka has its best performance when event size is about 10KBs. With different segment number, the throughput and latency may vary up to twice as much.

Pravega with flush achieves a maximum throughput which is 80% higher than Kafka without flush. And setting Pravega to prevent BookKeeper from flushing data to the persistent log doesn't have performance gains, which proves the reasonableness of Pravega's default persistence features.

Besides the benchmark in experimental environment, I also design and implement a simple scalable real-time stream data analytic system. Lots of research about the pipeline design and components selection are carried out. By integrating these components, from data loading, to message buffering, from stream processing to vector searching, I optimized the pipeline to meet the higher requirements in dealing with huge data volume and fits into the goal of scalability. In general, the system achieves the performance in demand, which could handle 30% of Twitter data in real-time on a single PC and might also be viewed as design sample for stream processing with heavier computation. In the meantime, there are also definitely something to be further explored and worked on, see in the future work section.

8 Future Work

Distributed. At this point, the whole system is deployed on a single node. However, nearly all these section are extendable to deploy on different machines for better performance. By using Nginx reverse proxy, it could distribute user HTTP request to different web server node. Pravega could partition its workload into different machines and achieve load balancing automatically. Spark provides master and slave cluster implementation. Usually Redis could handle heavy workload in single node due to single thread design. There is also Redis cluster API for deploying it on multiple machines.

More Features of Pravega. According to the Pravega introduction website, as a emerging messaging system, it aims to solve the problems that Kafka failed to solve. As a result, many interesting features are not involved in this project, like segments auto-scaling, ingestion of large data such as video, Consistent state replication, etc. All these could be discussed in future experiments.

Bottleneck. Based on intuition, the bottleneck of the whole system is most likely caused by slow data processing in Spark. So optimization in Spark calculation will be the main direction of future works. In the meantime, I will also change the message middle-ware from Pravega to Kafka to compare their performance in practical applications.

Acknowledgement

Thanks for the tutorial and patient guidance from Pravega community. When I was stuck at a certain stage for days, I found Pravega Slack Channel and expressed my confusion about the deployment of the system. The developer gives prompt explanation even though it is near the holidays. I would like to help more people start to learn Pravega in the future.

References

[1] Apache Software Foundation. Hadoop.

[2] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792. IEEE, 2016.

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[4] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM international conference on distributed and event-based systems*, pages 227–238, 2017.

[5] Armbrust et al. Structured streaming: A declarative api for real-time applications in apache spark. SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Shvachko et al. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, page 1–10, USA, 2010. IEEE Computer Society.

[7] Thusoo et al. Hive: A warehousing solution over a map-reduce framework. 2(2):1626–1629, aug 2009.

[8] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[9] Jianguo Wang and Xiaomeng Yi. Milvus: A purpose-built vector data management system. SIGMOD/PODS '21, page 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Zaharia. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.

Appendix

My code repo is Here. There are multiple service need to be deployed for running this system.

Pravega & Kafka

Install Pravega and Kafka first, then run these two separate server for benchmark experiments.

```
./gradlew startStandAlone (execute in Pravega repo)
zkServer start & kafka-server-start /usr/local/etc/kafka/server.properties
```

Figure 12: Pravega & Kafka Shell Command

Twitter & Spark & Trending Consumer

Install Spark as well as required Python3 packages dependency first, and then run the following commands in /bin directory. When running twitter_pravega.py, a bearer token of Twitter developer account is required. The spark doesn't support Pravega data format directly, so we need to include the packages in Maven Center when submit spark tasks.

Docker & Milvus Server

Install Docker first. Then follow the steps in this link to install Milvus - Milvus Installation. Then in docker-compose.yml, map the local path to your milvus.yaml file onto the corresponding docker container path to the configuration file /milvus/configs/milvus.yaml under the volumes section.

```
3.python3 twitter_pravega.py
4.spark-submit --packages io.pravega:pravega-connectors-spark-3.1_2.12:0.10.1 spark_pravega.py
5.python3 word_count_consumer.py
```

Figure 13: Streaming Pipeline Shell Command

```
6.sudo docker-compose up -d # in the same directory with docker-compose.yml

Creating milvus-etcd  ... done # showing these messages means start successfully
Creating milvus-minio ... done
Creating milvus-standalone ... done
```

Figure 14: Milvus Shell Command

Web Service

Install Flask, npm first. Then run commands below in corresponding directory.

```
7.flask run #in flask\_system directory
8.npm run install #run once in frontend directory
9.npm run serve #in frontend directory
```

Figure 15: Web Service Shell Command