Distributed Hayabusa: Scalable Syslog Search Engine Optimized for Time-Dimensional Search

Hiroshi Abe
Lepidum Co. Ltd./Cocon Inc.
National Institute of Information and Communications Technology

Keiichi Shima
IIJ Innovation Institute

Daisuke Miyamoto
The University of Tokyo

Yuji Sekiya
The University of Tokyo

Tomohiro Ishihara
The University of Tokyo

Kazuya Okada
The University of Tokyo

Ryo Nakamura
The University of Tokyo

Satoshi Matsuura
Tokyo Institute of Technology

ABSTRACT

Network administrators usually collect and store logs generated by servers, networks, and security appliances so that when network trouble and/or security incidents occur, they can identify the source of the problem by investigating the contents of the logs. The size of the system needed to store and search the log messages tends to increase as the size of the managed network becomes large. A fast log storage and search system called Hayabusa was previously proposed that optimizes a time-dimensional search operation. In this paper, we propose a simple distributed system that adds scalability to the existing Hayabusa system. The evaluation results show that the Distributed Hayabusa system consisting of 10 servers (with multiple worker processes on each server) is 36 times faster than a standalone Hayabusa system. The time required to perform a full-text search over 14.4 billion data records is only about 7 s, which is sufficiently low for the daily operations of administrators managing a very-large-scale network.

CCS CONCEPTS

• **Information systems** → **Distributed retrieval**; Data structures; • **General and reference** → *Performance*; • **Computer systems organization** → *Parallel architectures*;

KEYWORDS

Data Processing, Parallel Processing, SQL, Parallel System

ACM Reference Format:
Hiroshi Abe, Keiichi Shima, Daisuke Miyamoto, Yuji Sekiya, Tomohiro Ishihara, Kazuya Okada, Ryo Nakamura, and Satoshi Matsuura. 2019. Distributed Hayabusa: Scalable Syslog Search Engine Optimized for Time-Dimensional Search. In *Asian Internet Engineering Conference (AINTEC '19), August 7–9, 2019, Phuket, Thailand.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3340422.3343636

AINTEC '19, August 7–9, 2019, Phuket, Thailand

ACM ISBN 978-1-4503-6849-0/19/08…$15.00
https://doi.org/10.1145/3340422.3343636

1 INTRODUCTION

Network operators examine the health of a network by examining statistical information based on the logs continuously generated by network devices. When a problem occurs, a network administrator identifies the cause by searching through the log and fixes the problem to keep the network stable. In addition, network administrators sometimes use the log to determine what kind of incident has occurred as well as how to handle security incidents. In large-scale networks, many network devices, servers, and security devices output logs that record a large number of communications every day. Moreover, network administrators operate a system to store this large number of logs and search through the contents at high speed.

Clustering systems and proprietary management software are often used to handle large log search and storage systems. In this case, network administrators must spend time managing their search and storage systems in addition to their primary tasks. However, they do not need to manage search and storage systems if log storage and search systems can be built without complex clustering systems. This would enable them to focus on crucial tasks such as network problem handling and security incident analysis.

In this paper, we propose a system that can accumulate a large number of logs output from many multi-vendor devices at high speed and can search through them at high speed. We also propose a conceptual model of a distributed system in which search performance is improved when the system is scaled out. The search speed of the distributed system dramatically improves even if the number of logs handled by the system increases. In this study, we evaluate the storage and search performance of our proposed system with real log data captured at the Interop Tokyo 2017 network [1], which consists of more than 600 servers and network security devices provided by a large number of different vendors.

This paper is organized as follows. Section 2 describes related work and prior research. Section 3 introduces the architecture of the proposed system, called Distributed Hayabusa. Section 4 introduces an implementation of the Distributed Hayabusa system. Section 5 and 6 respectively present and discuss the evaluation results of Distributed Hayabusa. Finally, Section 7 presents our conclusions and future work.

2 RELATED WORK

Relational databases are often used for log analysis tasks such as firewall logs analysis and log volume monitoring. However, if a relational database is used to store log data, the database schema is typically designed to store log data. Hence, the storage program must parse the unstructured log data and convert them to structured data before storing them in a relational database, which is computationally costly. The system designer must also consider data separation, i.e., the use of daily, weekly, monthly, or annual tables, which depends on the amount of data to be stored. Once the system designer has designed the table format and schema, this structure is not easy to change while the system is running.

When the system administrator performs a full-text search and log analysis, a Hadoop ecosystem [8] such as a MapReduce algorithm [14] or Apache Spark [22] are often used. Huge Hadoop and Spark clusters provide users with fast search services and good search performance. The storage capacity and processing resources of Hadoop are designed to be scalable. However, because a Hadoop cluster is integrated with complicated software, it can be difficult for a system administrator to manage a Hadoop system stably. Even simply building a Hadoop cluster requires specialized software.

If a system administrator tries to operate a Hadoop cluster simply, the hardware failure rate increases as the cluster size increases. System administrators must hence have sophisticated knowledge and experience to identify fault locations and perform stable Hadoop cluster operations.

The HDFS (Hadoop Distributed File System) [18] and Elasticsearch [6] used in the Hadoop ecosystem act as distributed storage and achieve high availability. They hold copies of the data, and there are elaborate mechanisms to prevent the complete loss of data in the event of a failure. However, accessing storage through complex processes to improve reliability degrades processing performance.

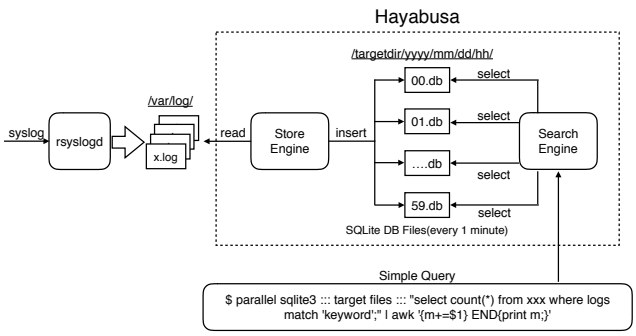

Figure 1: Architecture of Hayabusa.

Splunk [3] and VMware vRealize Log Insinght [11] are commercial solutions dedicated to log storage, indexing, and quick search. The search performance of these products is highly dependent on the size of the cluster that they operate. However, if the number of logs being processed increases, the system will need cluster expansion and additional licenses, increasing its price. Hence, achieving high performance and redundancy in commercial products can be prohibitively expensive.

BigQuery [10], a cloud service based on Google's Dremel [17], is a high-speed database service. A Google engineer demonstrated that BigQuery can scan 12 billion records in 5 s [1]. These servers run on the BigQuery back-end, built on thousands or tens of thousands of nodes, which incur vast operating costs.

There are also systems such as HBase [15] and InfluxDB [9] that specialize in time-series data. A time-series database is a data collection mechanism specialized for time-series accumulation and retrieval. These databases are good at accumulating numerical metrics such as monitoring data and most of the acquired data are stored in the form of key–value pairs. Furthermore, their implementation focuses on the compression of the received values and storage of a large amount of data in memory for high-speed processing.

2.1 Prior research

In previous studies on Hayabusa [13], the performance of Hayabusa was evaluated in a stand-alone environment implemented on a bare metal server. Hayabusa [13] was designed as a system to search a large number of syslog messages collected at high speed. Figure 1 shows the architecture of Hayabusa.

[1] https://www.youtube.com/watch?v=swsS12c1VGE

Hayabusa runs on a stand-alone server and performs high-speed parallel searches using multiple CPU cores. The architecture of Hayabusa can be divided into two main parts: the StoreEngine and SearchEngine. The StoreEngine, started by "cron" every minute, extracts a syslog message from a target file and converts it into a SQLite3 [7] file. Log data is divided into SQLite3 files every minute and then are processed in parallel by multiple processes. The directory where logs are stored is defined using a time-based hierarchy as follows.

```
/targetdir/yyyy/mm/dd/hh/min.db
```

Because Hayabusa embeds time information into the directory path structure, there is no need to hold information about time inside the database. This directory structure makes it possible to search logs for a specific time without specifying a query condition that takes a long time to process. The SQLite3 file in which the log is stored consists of a table in a format specialized for full-text search (FTS format). Quick log search is implemented by indexing specifically for full-text searches. Moreover, Hayabusa's database has only one table column for storing syslog strings. As a result, the Hayabusa database does not require a schema design, regardless of the type of log.

The SearchEngine accesses multiple SQLite3 files, which are defined as FTS tables created every minute, in parallel to improve search performance. The parallel SQL search queries use GNU Parallel [19] for each SQLite3 file. The results are aggregated using the "awk" and "count" commands through the UNIX pipeline.

Hayabusa works in a stand-alone environment, but has better full-text search performance than small-scale distributed processing clusters. However, hardware limitations exist in a stand-alone environment, and it is likely that its performance will eventually be overtaken by other, larger distributed processing clusters. Hence, in this proposal, we remove the limitation of the standalone environment for Hayabusa, as described in the following section.

3 PROPOSED DISTRIBUTED SYSTEM

In this research, our aim is to redefine standalone Hayabusa as a distributed processing system and to scale out search processing to improve performance.

3.1 Architecture

We first present the storage and scheduler for distributed Hayabusa. For the architecture, our aim is to retain processing performance while inheriting the simplicity of a standalone Hayabusa architecture to avoid system complexity. A distributed processing system such as Hadoop always generates multiple data exchanges by complex processes inside the system to operate correctly and robustly. However, we designed Distributed Hayabusa to maintain system simplicity and emphasize on processing speed without considering error processing and retrying processing in case of failure. The architecture of Distributed Hayabusa is shown in Figure 2.

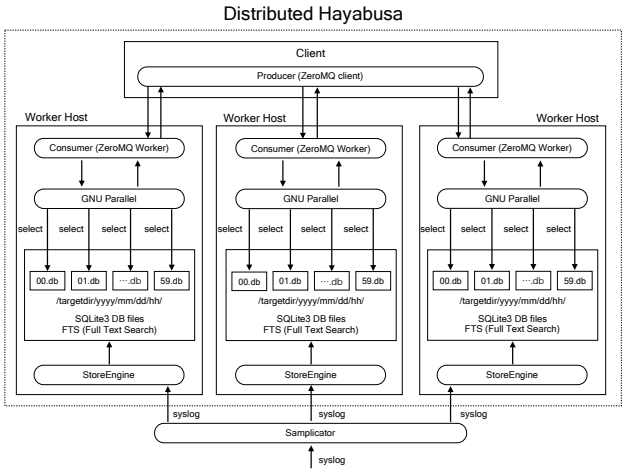

Figure 2: Distributed Hayabusa architecture.

3.2 Storage

The distributed Hayabusa storage maps time to SQLite 3 and the directory hierarchy, as in the standalone version. Clients can search for time ranges without specifying them in the query.

Further, to scale out the search process, distributed Hayabusa ensures that all processing hosts have the same data so that search queries can execute regardless of which host is used. This means syslog data must be replicated and delivered to all processing hosts. However, it ensures that the same result is returned, no matter which hosts processes a request. Replicating syslog also improves data integrity. Even if a processing host fails and data are lost, the data remain on another processing host to enhance fault tolerance.

3.3 Scheduler

Distributed Hayabusa uses load balancing with a remote procedure call (RPC) to schedule the search processing and process execution mechanism using the GNU parallel tool. When Distributed Hayabusa assigns search processing to distributed hosts, the search processing is assigned using load balancing, which equally distributes the processing and RPCs using the producer/consumer model. The search process received by each host and the query are executed as a parallel search using GNU parallel, which is equivalent to

the standalone version of Hayabusa. The result is returned to the client via a worker process using the RPC framework.

4 IMPLEMENTATION

4.1 Data replication

In this study, we used the open-source UDP Samplicator [5] to replicate and send syslogs to all processing nodes. The UDP Samplicator transfers the received UDP packet to the specified target host without changing the sender's address. This replication technique allows the destination host to receive UDP packets as if it had directly received data from the source. All processing hosts receive the same replicated syslog packet, as shown in Figure 2.

The UDP Samplicator performs UDP transfer processing in one process. Therefore, if it receives a large number of syslogs and the load rises, its CPU core usage will be 100%. In this case, the packet transfer process may not catch up, and data may be discarded. Therefore, we applied a patch to the UDP Samplicator using "SO_REUSEPORT" for the socket option and modified the source code to operate as a multi-process in the proposed system. As a result, when the UDP Samplicator receives a large number of syslogs, the process of copying and forwarding syslog packets employs multiple CPU cores and activates multiple processes.

4.2 Distributed search

Search processing requests are queued and processed by the producer/consumer model. The processing host corresponding to the consumer acquires the queued processing request. At this time, the producer balances the load so that processing requests can be distributed uniformly to each host.

Various software can implement the producer/consumer model, but in this research, we used ZeroMQ [16], which can execute the processing at high speed and implement the client and worker processes as a library. ZeroMQ is used as a high-speed distributed message queue and can quickly implement various messaging patterns such as "Request/Response," "Publish/Subscribe," and "Push/Pull." In the proposed system, we implemented the producer/consumer model using the Push/Pull pattern.

As shown in Figure 3, ZeroMQ Push/Pull patterns are processed in the following order.

1) The producer queues requests (push).
2) The consumer receives a request from the producer (pull).
3) The consumer sends the result to the result collector (push).
4) The results acquired by the result collector are summarized (pull).

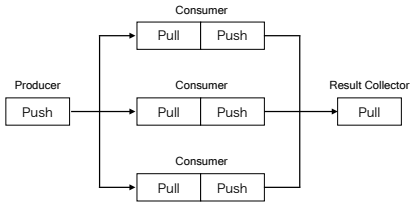

Figure 3: Push/Pull pattern.

```
import sys
import zmq

context = zmq.Context()
sender = context.socket(zmq.PUSH)
sender.bind("tcp://*:5557")
receiver = context.socket(zmq.PULL)
receiver.bind("tcp://*:5558")

cmd = 'parallel target-data "SQLite3 Query Strings"'

sender.send(cmd.encode('utf-8'))
message = receiver.recv()
print(message.decode('utf-8'))
```

Figure 4: Example client code.

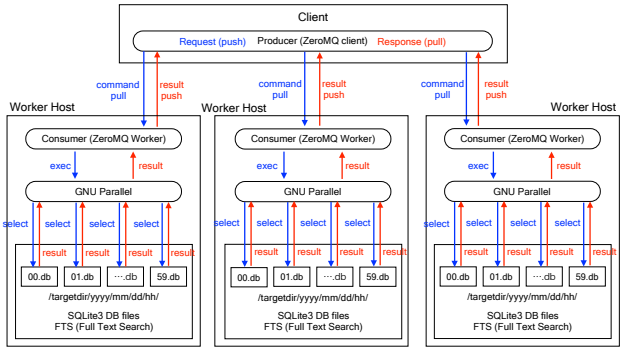

Figure 5: Push/Pull pattern using ZeroMQ on Distributed Hayabusa.

In our proposal, the client has two roles, producer and result collector. This implementation allows clients to submit requests, queue them, and obtain results in one process. The source code of the client is illustrated in Figure 4.

As shown in Figure 5, the client queues the processing request to be input to the processing host. The worker operating on each host then pulls the processing request. The worker sends the result to the client after the request has been executed, and the client aggregates the result. The client waits for a connection from the worker using TCP port 5557 and pushes a processing request onto a queue. The client then receives the processing result on TCP port 5558 and counts the results.

```
import zmq
import subprocess

context = zmq.Context()
receiver = context.socket(zmq.PULL)
receiver.connect("tcp://client:5557")
sender = context.socket(zmq.PUSH)
sender.connect("tcp://client:5558")

while True:
    recv = receiver.recv()
    cmd = recv.decode('utf-8')
    res = subprocess.check_output(cmd)
    sender.send(res)
```

Figure 6: Example worker code.

Table 1: Experimental environment.

EC2 instance	c4.4xlarge
vCPU	Intel Xeon CPU E5-2660 (2.9 GHz/16 cores)
Memory size	30 GB
Disk size	SSD 8 GB (OS) + SSD 50 GB (Data)
OS	Ubuntu 16.04.4 LTS (Xenial Xerus)

The source code of the worker is shown in Figure 6. The worker blocks the connection from the client until the client pushes the request onto the queue. Then, the worker pulls the processing request from TCP port 5557. Next, the worker processes the request that has been pulled and executes the command included in it. Then, the worker pushes the result to the client's TCP port 5558.

5 EVALUATION

We used virtual server groups on EC2 (Elstic Compute Cloud) provided by the Amazon Web Service (AWS) [4] to conduct a scale-out test in this study. In the scale-out test, the number of virtual servers increases from one to ten, and the search speed is evaluated. In addition to the processing hosts, one client host is prepared to request distributed queries. The specifications of the experimental host are listed in Table 1.

5.1 Evaluation data

Analysis of actual data shows that the syslog reception rate of Interop Tokyo's ShowNet [2] in 2017 was about 50,000 receptions per minute on average. The data were collected from the beginning of the exhibition period for three days (from June 7th to 9th). ShowNet expects further increases in syslog reception in the future. In this verification, we verified the scale-out search in a distributed environment using 100,000 syslogs per minute.

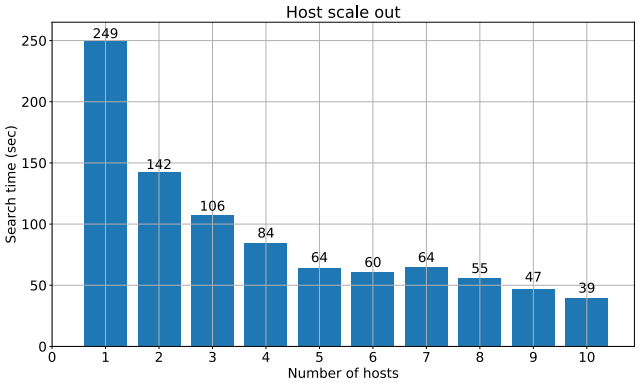

Figure 7: Host scale-out performance test.

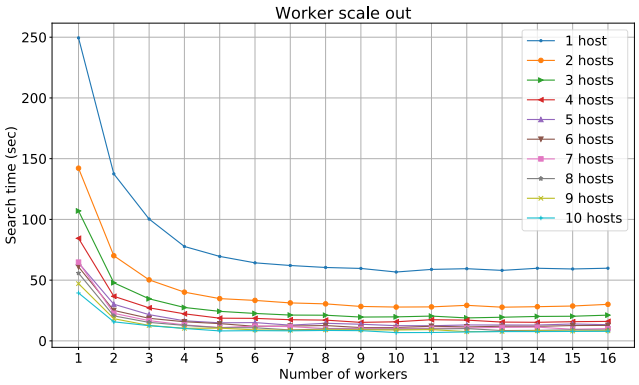

Figure 8: Worker scale-out performance test.

5.2 Host scale-out

To investigate the scale-out performance of the processing host, we tested whether the processing time could be reduced when the number of hosts increased. In this evaluation, the number of processing hosts increased from 1 to 10, and the client repeatedly executed 100 requests for data per day. The target record size for 100 requests is 14.4 billion records. The processing result is shown in Figure 7.

The results show that the search processing time for one host is about 249 s. The processing time decreases as the number of hosts increases; the processing time for ten hosts is approximately 39 s. Each processing time here is the average value of 10 trials.

5.3 Worker process scale-out

Next, we performed processing on 1 to 10 hosts and increased the number of worker processes from 1 to 16. Here, 16 processes were chosen because this is the number of vCPU cores in the virtual machine. We tested how much the performance improved if the number of worker processes increased up to the number of vCPU cores. The processing results are shown in Figure 8.

When the number of workers is around 10, as the number of hosts increases from 1 to 10, the processing speed is maximized. The processing takes about 249 s for one worker with one host, but the processing time reduces to about 6.8 s for 10 workers with 10 hosts. In this experiment, each processing time is also the average value of 10 trials.

5.4 Comparison with AWS's Elastic MapReduce

Next, we compared Distributed Hayabusa with the Elastic MapReduce (EMR) service provided on AWS. EMR is a service that allows a user to build Hadoop ecosystems such as Apache Hadoop/Hive/Spark. EMR can also refer to S3 data directly without preparing the HDFS environment by putting data in Amazon's S3 service.

In this experiment, we used EC2 instances (c4.4xlarge) with the same performance as instances used for the evaluation of Distributed Hayabusa to evaluate the EMR. EMR needs one master node to manage a cluster and a core node that processes data. In the evaluation, we increased the number of core nodes from 2 to 10 to confirm the scale-out performance as the number of hosts increased. We used EMR version emr-5.12.0 (Spark: Spark 2.2.1 on Hadoop 2.8.3 Yet Another Resource Negotiator (YARN) [20] with Ganglia 2.7.2 and Zeppelin 0.7.3), and selected Apache Spark's main package as the evaluation application.

We prepared 1,440 syslog files of 100,000 lines each in S3, which is equivalent to the estimated daily load. The experiment was performed in a situation in which the target syslog file size is 14.4 billion lines. An amount of information equal to that accessed by Distributed Hayabusa is accessed by repeatedly executing data requests from the client 100 times.

The client executed the PySpark code shown in Figure 9 on the master node and performed search processing on each core node. The fifth line represents the reading of the target log data from S3. Line 6 loads the data into Spark's resilient distributed dataset (RDD) [21] cache function. RDD is a distributed shared memory shared among multiple core nodes, and data can be searched for at high speed on the various nodes.

The processing result is shown in Figure 10. Because EMR cannot build an environment with one host, the results start at two hosts. Each reported processing time is the average of five trials. We observe that the processing performance of the full-text search scales out when increasing the number of hosts in Spark on EMR. In this experiment, we confirmed that Distributed Hayabusa operates 17 times faster than a configuration of 10 EMR Spark hosts for a full-text search result for the same syslog data.

```
import time
from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)
lines = sc.textFile("s3://abe-work/ssd2/benchmark-log/
    files/100k/100k-*.log")
lines.cache()

for i in range(5):
    start = time.time()
    [lines.filter(lambda s: 'noc' in s).count() for i in
        range(100)]
    elapsed_time = time.time() - start
    print elapsed_time
```

Figure 9: PySpark code.

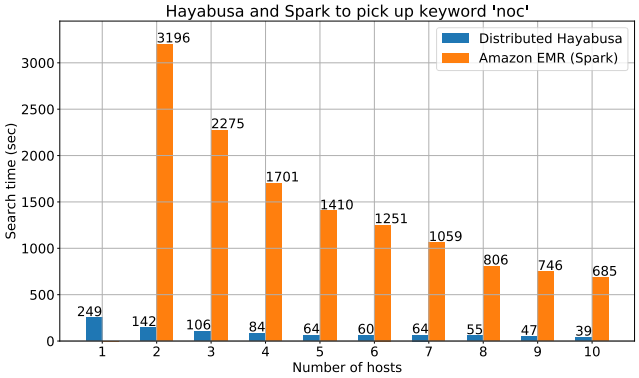

Figure 10: Hayabusa and Spark time comparison.

6 DISCUSSION

6.1 Effect of scale-out on search performance

The results in Sections 5.2 and 5.3 show that when the host numbers were increased, the search was approximately 6.3 times faster than the search time of one processing host. Moreover, the search time, which took about 249 s on one processing host, was reduced to about 6.8 s, which is approximately 36 times lower, as a result of the combination of scaling out both the number of hosts and the number of workers.

The number of records targeted by our experiment was 14.4 billion. The results show that Distributed Hayabusa could thoroughly scan the 14.4 billion records in about 7 s, indicating that we could achieve a full-scan speed comparable to Google's BigQuery, which uses more than several thousand servers.

Distributed Hayabusa was able to realize such high-speed scanning with ten processing hosts. We hence conclude that the Distributed Hayabusa architecture can realize a high-performance distributed processing system that is both reasonable and cost-effective.

6.2 Parallelization of log data accumulation

In this research, we used a method to duplicate the same syslog data to each processing host to cope with distributed queries and hence improve search performance. This method is essentially an act of duplicating a large amount of data, which means that as the amount of data increases, and more waste occurs in the network bandwidth and the data that must be stored. Of course, it is possible to set the number of replications, as in Hadoop HDFS, and to distribute and hold data on multiple hosts. In that case, the metadata management mechanism manages the data. Data access is via the metadata management mechanism, which may reduce processing performance.

In our proposed system, there are bandwidth and capacity problems due to the data duplication. However, there is no need to relocate data, as in other distributed file systems, in the event of device failure. Moreover, devices are excluded merely from management cluster targets. It can correspond by when trouble happens in cluster.

6.3 Simplicity of design and operation

We implemented the data replication mechanism and distributed the search using the producer/consumer model in the proposed system. Both the design and implementation are simple, and there are few processes to manage. Distributed systems like Hadoop integrate many complex software components. However, when a system problem occurs in Hadoop, the complexity of understanding the cause of the problem increases. Distributed Hayabusa constructs a distributed processing mechanism with very few components. Hence, when a problem occurs in Distributed Hayabusa, the problem can be quickly understood, which reduces the load of system operation management.

6.4 Comparison with other systems

In this study, we conducted a comparative experiment with Amazon EMR and obtained the result that distributed Hayabusa performs a full-text search about 17 times faster than EMR. This is due to some structural differences that lead to performance differences in processing. There are several places in which EMR can have a processing bottleneck.

Spark, which is used in EMR, runs on Hadoop's resource management mechanism YARN. YARN monitors resource allocation in Hadoop clusters, monitors and tracks running jobs, and manages access to shared datasets held by clusters. This makes it possible to manage the entire cluster soundly and control multiple jobs in a multi-tenant environment. Distributed Hayabusa has no resource management mechanism at present, and it is the worker that promptly executes requests received from clients. This lack of a resource management mechanism is one reason Distributed Hayabusa operates at high speed. However, because Distributed Hayabusa does not manage or track jobs, error handling and retry processing cannot be performed if problems occur. In the system architecture in this proposal, the user is not informed if a problem has occurred and can only infer this from the processing result.

Next, we discuss the process execution scheduler. Spark creates a directed acyclic graph (DAG) for task execution before scheduling the task. Spark also uses RDD to store data in distributed shared memory and share data between DAGs. In this way, Spark can complete the job at high speed by optimizing and sharing data between DAGs (without writing intermediate results to disk). The scheduling mechanism realized by Distributed Hayabusa depends on load balancing performed by the ZeroMQ client and the execution scheduling of GNU parallel. It works fast because it is simple and has little overhead. Distributed Hayabusa is not like Spark, which calculates the optimal execution plan and realizes processing while using distributed shared memory along with the execution plan.

In this experiment, EMR read data from S3. Normally, a Hadoop ecosystem would use distributed data on HDFS. HDFS distributes data to each host in block units when the data reach or exceed a specific size. In that case, the client accesses the data via the HDFS metadata mechanism, which slows storage access. Hayabusa maintains data as a one-minute SQLite3 database file. Each file has a fast search mechanism indexed in FTS format, which is specialized for full-text search. Furthermore, if a user wants to narrow down the search by time range, Hayabusa does not have to specify the time as a SQL query condition, which speeds up processing. In addition, because Hayabusa storage management does not go through a metadata mechanism, high-speed data access is possible.

6.5 Hayabusa2

The Distributed Hayabusa system proposed in this paper does not implement some functions such as resource management and specific scheduling mechanisms to speed up processing. Therefore, the architecture cannot perform retries if an error or exception occurs. In addition, there are problems with multi-tenancy and storage capacity that remain. Therefore, we designed Hayabusa2, which is an improvement of the proposed Distributed Hayabusa. The architecture of Hayabusa2 is depicted in Figure 11. Hayabusa2 introduces a scheduling mechanism (Request Broker) that can handle errors and exceptions. In addition, Hayabusa2 uses network storage, enabling it to store large amounts of

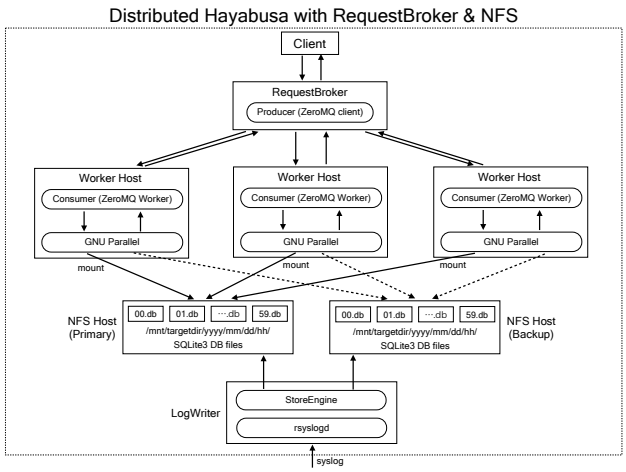

Figure 11: Hayaubsa2 architecture.

data without copying the data. However, Hayabusa2 now uses more processing overhead.

Whether to use the proposed Distributed Hayabusa or Hayabusa2 differs depending on the processing requirements of the user. If the user needs state-of-the-art processing performance, the user can choose Distributed Hayabusa. If storage capacity is essential, Hayabusa2 can be used. We have released Hayabusa2 as open-source software on GitHub [12], and we are continuing development. The performance evaluation of Hayabusa2 is a future task.

7 CONCLUSION AND FUTURE WORK

In this paper, we designed and implemented the distributed processing version of the Hayabusa system. The evaluation results indicate that our implementation reduces the search processing from 249 s on one processing host to a minimum of about 7 s.

Distributed Hayabusa performs a full scan of 14.4 billion records of syslog data in 6 s and delivers high performance as a full-text searchable log search engine. Distributed Hayabusa will be a useful tool for network administrators managing multi-vendor devices. It enables a large number of logs to be used to perform troubleshooting and incident response, which may shorten response times. Moreover, Distributed Hayabusa has a simple system design, which substantially lowers system management costs, allowing network administrators to spend time on other important tasks.

Acknowledgement

This work was supported by JST CREST Grant Number JPMJCR1783, Japan.

REFERENCES

[1] 1994. Interop Tokyo. http://www.interop.jp/.
[2] 1994. ShowNet. http://www.interop.jp/2017/shownet/.
[3] 2003. Splunk. https://www.splunk.com/.
[4] 2006. Amazon Web Service. https://aws.amazon.com/.
[5] 2009. UDP Samplicator. https://github.com/sleinen/samplicator.
[6] 2010. Elasticsearch. https://www.elastic.co/products/elasticsearch.
[7] 2010. SQLite. https://www.sqlite.org/.
[8] 2011. Apache Hadoop. http://hadoop.apache.org/.
[9] 2013. InfluxDB. https://www.influxdata.com/time-series-platform/influxdb/.
[10] 2013. An inside look at google bigquery. (2013). https://cloud.google.com/files/BigQueryTechnicalWP.pdf
[11] 2014. VMware vRealize Log Insight. https://www.vmware.com/products/vrealize-log-insight.html.
[12] 2018. Hayabusa2. https://github.com/hirolovesbeer/hayabusa2.
[13] Hiroshi Abe, Keiichi Shima, Yuji Sekiya, Daisuke Miyamoto, Tomohiro Ishihara, and Kazuya Okada. 2017. Hayabusa: Simple and Fast Full-Text Search Engine for Massive System Log Data. In *Proceedings of the 12th International Conference on Future Internet Technologies (CFI'17)*. ACM, New York, NY, USA, Article 2, 7 pages. https://doi.org/10.1145/3095786.3095788
[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[15] George L. 2011. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data* (1st ed.). O'Reilly Media, Inc.
[16] Pieter Hintjens. 2011. 0MQ - The Guide. http://zguide.zeromq.org/page:all
[17] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*. 330–339. http://www.vldb2010.org/accept.htm
[18] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972
[19] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb. 2011), 42–47. https://doi.org/10.5281/zenodo.16303
[20] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2523616.2523633
[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301
[22] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113