Hayabusa: Simple and Fast Full-Text Search Engine for Massive System Log Data

Hiroshi Abe
IIJ Innovation Institute, Japan
abe@iij.ad.jp
Japan Advanced Institute of Science and Technology
h-abe@jaist.ac.jp

Keiichi Shima
IIJ Innovation Institute, Japan
keiichi@iijlab.net

Yuji Sekiya
The University of Tokyo
sekiya@nc.u-tokyo.ac.jp

Daisuke Miyamoto
The University of Tokyo
daisu-mi@nc.u-tokyo.ac.jp

Tomohiro Ishihara
The University of Tokyo
sho@c.u-tokyo.ac.jp

Kazuya Okada
The University of Tokyo
okada@ecc.u-tokyo.ac.jp

ABSTRACT

In this study, we introduce a simple and high-speed search engine for large-scale system logs, called Hayabusa. Hayabusa uses SQLite, standard lightweight database software with GNU Parallel and general Linux commands, such that it can run efficiently without complex components. Network administrators can use Hayabusa to accumulate and store log information at high speeds and to search the logs quickly.

In our experiments, Hayabusa required only 8 seconds to convert 1.2 M log messages into a database file. Moreover, Hayabusa required only 5 seconds to search a keyword from 1.7 billion records. Hayabusa achieved high-performance search speed in a stand-alone environment without a complex distributed environment. Compared with the distributed environment, Spark, the proposed stand-alone Hayabusa was approximately 27 times faster.

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, 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 CFI'17, Fukuoka, Japan, June 14-16, 2017,* 7 pages.
https://doi.org/10.1145/3095786.3095788

CFI'17, June 14-16, 2017, Fukuoka, Japan

ACM ISBN 978-1-4503-5332-8/17/06…$15.00
https://doi.org/10.1145/3095786.3095788

1 INTRODUCTION

Network administrators are responsible for providing stable networks for users. These administrators conduct multiple analyses in their daily operations to efficiently detect system failures or network attacks. Consequently, they collect various system data, including raw packets, syslog messages, and sampled network flows generated by multiple devices on the network.

When a system failure occurs on a network, the administrators must detect the cause of the trouble as soon as possible. Of course, it is better when cyber attacks against the network can be detected before the attack leads to serious damage to the systems or user devices. However, these analyses must handle large amounts and various types of data. Further, data must be searched iteratively to find the small target data related to the issue they are tracking.

The search time required to do so depends considerably on the number of datasets and the data volume. Unfortunately, the number of devices that are potential sources of problems and the data size are steadily increasing, owing to the rapid deployment of multiple middleboxes, such as next-generation firewalls, sandboxes, and accelerators on networks. Moreover, the scale of monitored networks can be very large in some cases. In data centers, facilities often comprise tens of thousands of servers and network devices. Additionally, virtualization technologies consolidate software instances on a single physical device. These virtual devices also generate traffic and logs similar to physical devices. Administrators cannot avoid this trend and can rarely reduce the data size of the networks and servers. Therefore, a scalable and high-speed search engine for the log data in network operations is desirable.

Unfortunately, dedicated log analysis systems for individual devices or applications are unsuitable for log management. In some cases, operators use Hadoop-type [1] cluster-based data storage and search engines instead of expensive commercial products. Typically, however, these cluster-based systems lead to additional operational costs for operators. Furthermore, many parameter configurations and numerous tunings are needed to meet the required performance. Even when administrators merely seek a high-speed log search system for daily troubleshooting jobs, they end up managing a complex distributed system by themselves, and this complex system is itself prone to difficulties.

In this study, we propose a simple and fast system, called Hayabusa, to search for required information from a large amount and a wide variety of log messages. Typically, when processing a large amount of data, a large-scale distributed processing system is used, such as Hadoop. However, we propose a stand-alone parallel processing system based on SQL. The proposed system is a simple management system, rather than a complicated distributed system. Hayabusa provides a simple parallel processing mechanism suitable for searching log messages. Administrators need only issue a simple SQL sentence that is distributed to multiple CPU cores for parallel processing to search log messages, and the results are merged using the UNIX Pipe mechanism and UNIX commands.

This paper is organized as follows. Section 2 describe the related works of this research. Section 3 introduces proposed system of Hayabusa. Section 4 and 5 describe Hayabusa's evaluation result and discussion. Finally, Section 6 presents our conclusions and future works.

2 RELATED WORK

The MapReduce[8] algorithm and Hadoop ecosystems (such as Apache Spark[2]) are usually used as full-text search engines to analyze system log messages. The big Hadoop and Spark clusters provide users with fast processing speeds, insofar as the number of hosts is increasing. However, managing Hadoop and Spark clusters is very difficult for system administrators, because these approaches are designed with complicated software. Administrators who merely want simple search and count functions will find that Hadoop ecosystems are too complex to use and manage.

Furthermore, distributed systems have storage problems[9]. HDFS[10] is the storage component of the Hadoop ecosystem, and it is highly reliable for data preservation, insofar as it adopts a file replication system for multiple storage nodes. Elasticsearch[4] is another distributed data store. Elasticsearch can store data easily using the REST API, and several hosts replicate the stored data. HDFS and Elasticsearch achieve high reliability in terms of data preservation by making multiple copies, but this mechanism leads to slow data storage.

UNIX commands such as the Grep and Awk commands are often used for data search and analysis. However, if administrators attempt to execute high-speed data processing, they need in-depth knowledge of the data structure and the system environment (such as memory, number of cores, and system cache files). Administrators usually use the UNIX commands as a single process for simple usage. However, the single processing flow is sequential, and this does not usually achieve fast processing speeds.

3 PROPOSED SYSTEM

3.1 Architecture of Hayabusa

In this section, we show the design of the proposed system. The system goals are to store a large amount of data and retrieve the necessary data at high speeds using a simple approach.

Figure 1 shows Hayabusa's architecture. The architecture of Hayabusa is very simple. Hayabusa comprises two core parts: the StoreEngine, and the SearchEngine. The StoreEngine reads a system log file generated with external programs such as rsyslogd, and converts the log file into a database (DB) file. The converted DB files are placed in a structured directory suitable for retrieval. The SearchEngine runs multiple commands in parallel and aggregates the executed results using the UNIX Pipe mechanism. The operation works as a parallel processing mechanism without complex distributed components.

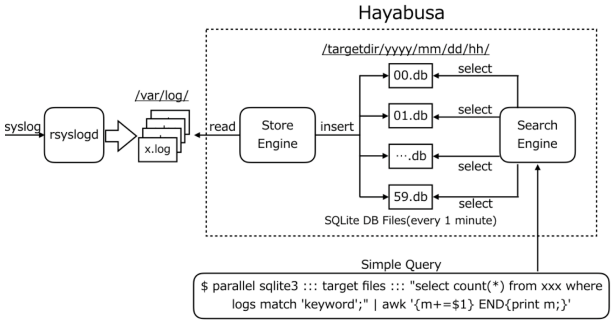

Figure 1: Architecture of Hayabusa

3.2 Implementation of Hayabusa

In our implementation, we selected SQLite[6], which offers Full-Text Search (FTS) capability as Hayabusa's core function. The reason for selecting SQLite is that we can achieve high-performance full-text searches in a stand-alone environment without a complex distributed environment. SQLite was used exclusively for the full-text search engine. That is, we are not using SQLite as a usual relational database in this proposal.

The SearchEngine runs multiple SQL commands in parallel using GNU Parallel[11] and aggregates the executed results using the UNIX Pipe mechanism.

3.3 Design of the directory structure

If a user or a program requests data from a specified period, it is necessary to pass the start time and the end time in the command argument to the search application in the case of SQL. The more users that add other conditions to pass to the SQL command execution, the slower the search time will be. In the case of log files, a user or a program must find matching lines whose recorded times are between the start time and the end time. With the proposed method, the system stores data files in directory paths that are based on the log times. A search program can simply specify the search target time using that directory path.

More specifically, the directory path comprises the associated time strings of the target log messages as shown below.

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

Advantage: By specifying a search time-range in *"the directory path + yyyy + mm + dd + hh + min.db"*, the search program can select the search time systematically.

Disadvantage: Insofar as the StoreEngine generates a DB file every minute, the number of DB files to be managed increases. Moreover, when the SearchEngine executes a search operation, it is impossible to find the data in the DB file that is currently being

```
1  import os.path
2  import sqlite3
3
4  db_file = 'test.db'
5  log_file = '1m.log'
6
7  if not os.path.exists(db_file):
8      conn = sqlite3.connect(db_file)
9      conn.execute("CREATE VIRTUAL TABLE SYSLOG
           USING FTS3(LOGS)");
10     conn.close()
11
12 conn = sqlite3.connect(db_file)
13
14 with open(log_file) as fh:
15     lines = [[line] for line in fh]
16     conn.executemany('INSERT INTO SYSLOG VALUES( ?
           )', lines)
17     conn.commit()
```

Figure 2: Insert code

used for data insertion. The search operation can only be done for data recorded more than one minute beforehand.

3.4 Storing data at high speed

To store data at high speeds, the StoreEngine reads the log data from the file and inserts the data by using a transaction mechanism in SQLite. Because the StoreEngine has been designed to create a data file every minute, it needs to read one minute of log messages to generate a single DB file.

The actual insertion code is shown in Figure 2. This code performs the following operations.

(1) Creation of an SQLite (FTS format) DB
(2) On-memory processing of the log messages using the Python list-comprehension syntax
(3) Data insertion using the SQL transaction mechanism

First, the StoreEngine generates a DB file for SQLite in the form of Full-Text Search 3 (FTS3). Then, the StoreEngine reads a log file and stores each line in the `lines` array variable. The content of the lines array variable is loaded into the memory. The values are written to the DB file in FTS3 format by calling the `executemany` database transaction method. The DB file is locked by the transaction when the data is being inserted, and no other process can read that DB file.

Advantage: The StoreEngine can read log files and insert a DB file quickly by holding the file in memory. This is because a data file is created every minute, and modern server equipment has an enough memory to process one-minute data.

Disadvantage: Other processes cannot access the DB file until the transaction completes. Accordingly, the SearchEngine cannot search for messages stored in a DB file currently being used for data insertion.

```
$ parallel sqlite3 ::: target files ::: "select
    count(*) from xxx where logs match 'keyword';"
     | awk '{m+=$1} END{print m;}'

Parallel Processing : parallel + sqlite command
Aggregator : pipe(|) + shell script(awk)
```

Figure 3: Simple Parallel Processing

3.5 Retrieving data at high speeds

The SearchEngine provides a simple parallel-processing function to find data at high speeds. This simple function is based on parallel SQLite command execution using GNU Parallel and the UNIX Pipe, with the Awk command as an aggregator.

In the SearchEngine, a parallel processing code is expressed by the combination of GNU Parallel and SQLite commands. The same number of processes is generated as the number of files passed as a command line argument to the GNU Parallel software. An aggregate code receives the parallel processing results through the UNIX Pipe and aggregates the data with the Awk command.

Advantage: Insofar as many processes are executed simultaneously, it is possible to core-scale them when they are executed on a machine equipped with many core CPUs. If access to the SQLite DB file is fast, it is feasible to collect the results within a small I/O wait time.

Disadvantage: The load time for the disk I/O is high when the disk speed is slow, because the SQLite command that accesses the DB files is executed in parallel.

4 EVALUATION

In this section, we show the evaluation results for the proposed system. We conducted benchmarks to reveal storage and search performance with an original large-scale dataset.

Each time a measurement was performed in this experiment, the Linux file cache was cleared (i.e., the buff/cache item, which can be confirmed with the "free" command). More specifically, we executed the following command each time before starting a measurement.

```
# echo 3 > /proc/sys/vm/drop_caches
```

4.1 Description of the environment

In the evaluation, we ran all benchmarks on a server: Intel Xeon E5-2670-v3 (12 cores, 2.3 GHz) x 2, with DDR4 384 GB memory and an 800 GB Intel SSD. The operating system of the server was CentOS 7.1, with Linux Kernel 3.10. We used an actual dataset for the benchmarks. The dataset included syslog data collected from the Interop Tokyo ShowNet[5] 2016, which is a Japanese large-scale demonstration network comprised of over 400 devices. The number of log messages received from the equipment in the Interop Tokyo ShowNet 2016 comprised 4.35M lines, collected over two weeks.

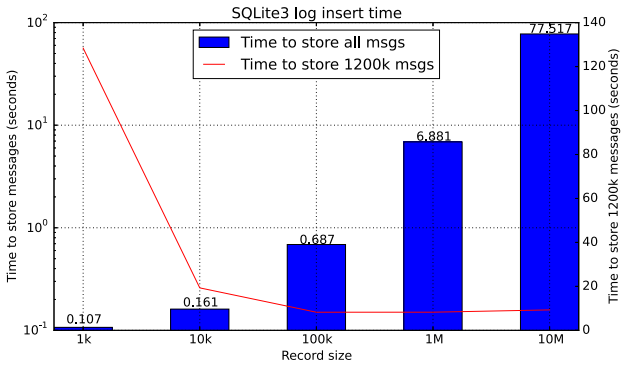

Figure 4: SQLite insert time

4.2 Storage performance

Insofar as the message-generation rate for the ShowNet of Interop Tokyo 2016 reached 20,000 messages per second, we adopted this as the target value for the experiment. The total number of messages stored in one minute was 1200k (20k x 60 s) messages. Therefore, to receive all of the syslog messages generated by ShowNet, Hayabusa must be able to store 1200k messages per minute. Consequently, the goal of the StoreEngine was to accumulate the data from 1200k messages per minute. Thus, the StoreEngine satisfies the desired performance if the log message size increases to more than 1200k messages per minute. Given this, we retrieved benchmark data from an amount greater than 1200k as a preliminary experiment for the target and performance measure (100k, 1M, the amount of data 10M) carried out in this study.

Figure 4 shows the average time spent inserting 1k, 10k, 100k, 1M, and 10M records. As the number of messages increased, the time spent inserting data increased linearly. Figure 4 also shows the time required to store 1200k messages. In the cases of 1M and 10M records, the time spent was approximately 8 s. This is faster than the desired 1200k messages per minute. The StoreEngine thus satisfied the desired performance.

4.3 Retrieval performance

To perform high-speed data retrieval, we utilized GNU Parallel in this proposal to achieve a simple parallel-processing mechanism. The parallel-processing mechanism was implemented with a combination of GNU Parallel, SQLite, UNIX Pipe, and the Awk program. GNU Parallel executes the same number of processes as the number of files passed as a command line argument, and it launches an SQLite process for each file. We adopted the following two benchmarks.

(1) Single search process for a single DB file with various record sizes
(2) Multiple search processes with GNU Parallel for multiple DB files of various record sizes

As shown in Figure 5, we can confirm what kind of commands will be executed by the GNU Parallel command by passing the `dry-run` option. The results from SQLite commands executed in parallel are aggregated by the Awk command, as shown in Figure 6. Although the series of SQLite commands are a part of the parallel processing process executed by GNU Parallel, the Awk command, which is aggregate processing, will not be performed in parallel.

```
$ parallel --dry-run sqlite3 ::: /path/1k-[0-9].db
    ::: "select count(*) from syslog where logs
    match 'noc';"
sqlite3 /path/1k-1.db select\ count\(\*\)\ from\
    syslog\ where\ logs\ match\ \'noc\'\;
sqlite3 /path/1k-2.db select\ count\(\*\)\ from\
    syslog\ where\ logs\ match\ \'noc\'\;
sqlite3 /path/1k-3.db select\ count\(\*\)\ from\
    syslog\ where\ logs\ match\ \'noc\'\;
...
```

Figure 5: Dry-run results with GNU Parallel

```
$ parallel sqlite3 ::: /path1k-[0-9].db ::: "
    select count(*) from syslog where logs match '
    noc';" | awk '{m+=$1} END{print m;}'

Parallel Processing : parallel sqlite3 ::: /path1k
    -[0-9].db ::: "select count(*) from syslog
    where logs match 'noc';"
Aggregator : | awk '{m+=$1} END{print m;}'
```

Figure 6: Counting code example

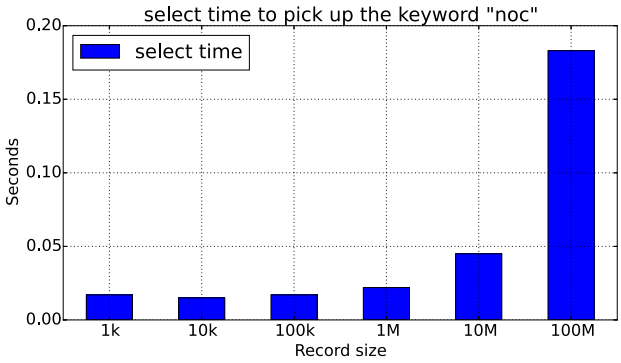

Figure 7: SQLite select time

4.3.1 Benchmark results (varying record sizes). Figure 7 shows the results of the total search time for one SQLite DB file with different record sizes (1k, 10k 100k, 1M, and 10M). We searched for the word "noc" in the files, using an FTS with each benchmark. According to the results, even for the DB file with 100M records, the search process required less than 0.19 s with the FTS function. Therefore, we confirm that the FTS function on SQLite works well for a single DB file.

4.3.2 Benchmark result (each file numbers). In the previous experiment, we benchmarked a single DB file search time. Next, we measured the search time with multiple files (1, 10, 100, and 1000

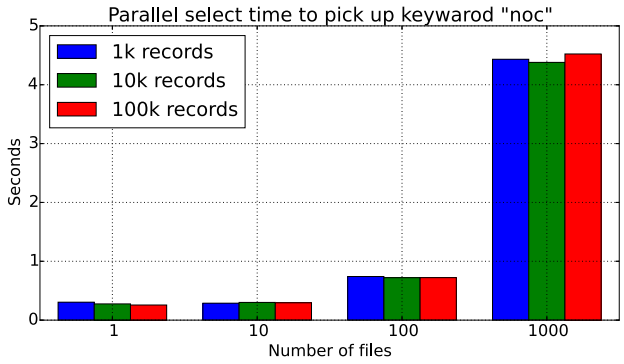

Figure 8: Parallel SQLite execution time

```
from pyspark.sql import SQLContext
from pyspark import SparkContext

sc = SparkContext(appName = "test")

sqlContext = SQLContext(sc)
lines = sc.textFile("/path/to/100k-*.log")

print(lines.filter(lambda s: 'noc' in s).count())
```

Figure 9: PySpark code

files), with various record sizes (1k, 10k, and 100k). We conducted this evaluation using GNU Parallel.

Figure 8 shows the benchmark results. In the graph, the x-axis shows the number of files, and the y-axis shows the time in seconds. Even as the number of records increased, SQLite maintained the same speed for FTS.

4.4 Comparison with Apache Spark

To compare the proposed method to another method, we measured the search execution time using Apache Spark. We executed Python code using the `spark-submit` command on a stand-alone Spark environment. The Python code is shown in Figure 9.

As shown in Figure 10, Spark and Hayabusa were respectively executed by changing the number of files (1, 10, 100, and 1000). Furthermore, we varied the number of records stored (or lines of messages, with Spark). When the record (or log) size was 1k and 10k, the time required to search for a word was almost the same for both Spark and Hayabusa.

When there were fewer than 100 files, Hayabusa was faster than Spark. However, in the case of 1000 files with 1k and 10k records, Spark was faster than Hayabusa. Nevertheless, in cases with large files (100k records), Hayabusa was approximately four times faster than Spark.

4.5 Comparison of the distributed processing environment

We performed a comparison between the multiple-host Spark environment and our proposed stand-alone Hayabusa. In the evaluation, we used three servers with the same configuration: Intel Xeon E3-1231-v3 (4 cores, 3.4 GHz, 8 MB cache), with DDR3 32 GB memory and a 400 GB Intel SSD 910 (PCIe 2.0). The operating system for the servers was CentOS 7.1, with Linux Kernel 3.10.

The number of files was a pattern of 1, 10, 100, and 1000 files. The DB record size and the logs were both 100k. The Spark environment was built using CDH (Cloudera's open-source software distribution)[3].

Spark was over ten times faster than Hadoop's MapReduce[12]. However, as shown in Figure 11, Spark was slower than the proposed stand-alone Hayabusa environment. With 1000 files, the stand-alone Hayabusa was approximately 27 times faster than the distributed Spark environment.

5 DISCUSSION

5.1 Data structure suitable for retrieval

In this proposal, we designed a time-range-based file-splitting approach. Therefore, it is possible to improve its search speed by treating the file names as a time condition and by omitting the SQL-based time condition required by a traditional approach. There is a drawback to doing so, however, insofar that the number of files will increase. Nevertheless, because we can assume parallel processing using GNU Parallel on suitable file paths, the efficiency of search operations will eventually improve.

5.2 Speed of SQLite

The FTS function in SQLite realizes high-speed searches using B-Tree word management. The FTS table itself is a large index, and it provides for faster operations than the string-matching process executed by a standard SQLite `LIKE` operation. If the formats of the log messages are known and structured, we might be able to design a database scheme suitable for specific messages, thus improving the search performance. However, given that a wide variety of log messages cannot be structured like syslog, full-text searching is appropriate.

Full-text searches with SQLite operate at very high speeds, compared to full-text searching with other programs. This is due to the manner in while files are handled with SQLite. SQLite uses the mmap system call to process DB files in memory. In other programs, searching is slower than SQLite, because they use the read system call for file content.

5.3 Parallelism of GNU Parallel

Apache Spark executes processes efficiently using a scheduler in a distributed environment. However, GNU Parallel can run multiple processes at the same time efficiently in a stand-alone environment. In this study, Hayabusa adopts stand-alone parallel SQLite execution, and this was demonstrated to be faster than the distributed Spark environment. This parallel execution is a powerful environment for log analysis, because it is very simple.

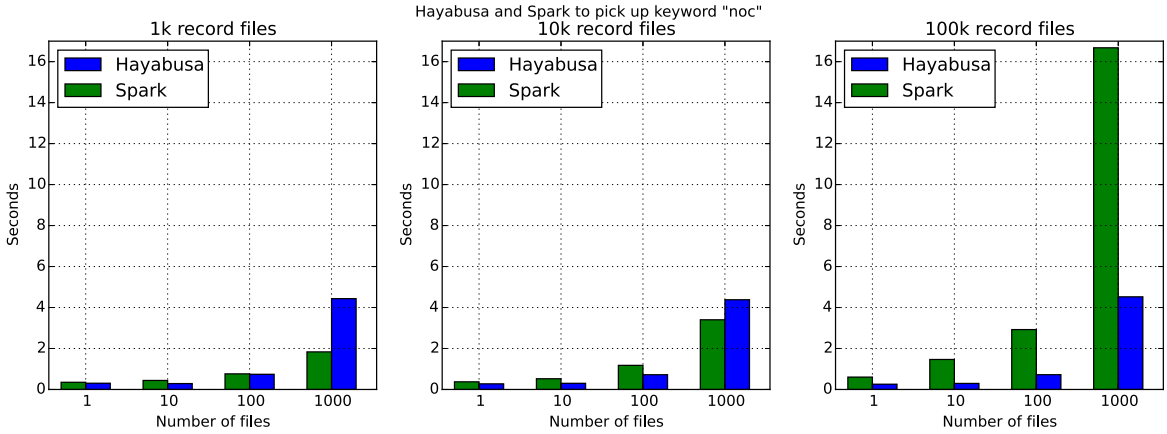

Figure 10: Hayabusa and Spark time comparison

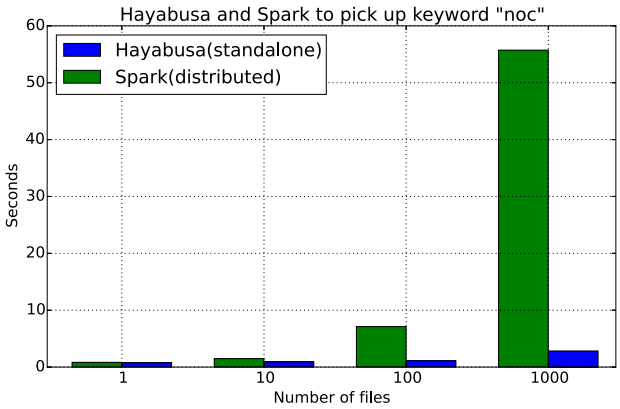

Figure 11: Comparison of the distributed Spark environment and the proposed stand-alone Hayabusa

Apache Spark is usually integrated with the HDFS file system, providing for an abstract storage access layer. Accessing the HDFS using the abstract layer is slow, however, and Spark cannot map the distributed files directory to memory, as is the case with the mmap system call.

5.4 System management costs

Managing the Hadoop ecosystem is difficult because it requires managing a large distributed system. Distributed systems have many complex management programs, in order to achieve high availability such that it continues to work even when some parts of the system are inoperative. Moreover, the programs or applications for data analysis run on a compound infrastructure. If problems occur in these programs or applications, it is especially difficult for administrators to find the location of the problem. By contrast, Hayabusa works on a stand-alone server, thus minimizing management costs.

5.5 Simple execution

Hayabusa's execution command is simply one-liner code, as shown in Figure 6. In our experiments, we used the `spark-submit` command provided by the Spark framework. Compared to the Spark code shown in Figure 9, Hayabusa's code is much simpler.

6 CONCLUSION AND FUTURE WORK

Hayabusa achieved good performance when accumulating and retrieving data. This is because Hayabusa is based on a time-range file-splitting approach. The StoreEngine required less than seven seconds to read and insert the DB file, which is sufficiently fast for our target message-generation rate (of 1200k messages per minute) calculated from the real operation of the Interop Tokyo 2016. The SearchEngine realized fast full-text searches, without being affected by the number of records that are registered there and the number of DB files. Hayabusa also searched keywords 27 times faster than the distributed Spark environment. Our proposed Hayabusa was implemented with simple and strong parallel processing using GNU Parallel. We believe Hayabusa is an efficient system that can resolve system and network problems quickly. Apache Spark provides a text-searching function, but only with a cache mechanism for searching repeatedly in memory. Furthermore, Spark contains multiple libraries for machine-learning calculations. Yet, the most important action for troubleshooting is fast log storage and searching. Hayabusa thus offers these crucial functions for administrators, without the complexity of Spark.

Currently, Hayabusa only works on a single server, thus limiting its storage capability. There are some distributed storage products available, such as HDFS or GlusterFS[7], and if Hayabusa is integrated with these, it can offer large-scale storage. Indeed, we shall consider these options for distributed storage in future research.

REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Apache Spark. http://spark.apache.org/.
[3] Cloudera's open source software distribution. https://www.cloudera.com/products/open-source/apache-hadoop/key-cdh-components.html.

[4] Elasticsearch. https://www.elastic.co/products/elasticsearch.
[5] ShowNet. http://www.interop.jp/2016/shownet/.
[6] SQLite. https://www.sqlite.org/.
[7] Simple application of glusterfs: Distributed file system for academics. *International Journal of Computer Science and Information Technologies*, 6(3), 2015.
[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
[9] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.
[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
[11] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.
[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.