Distributed Log Search based on Time Series Access and Service Relations

Tomoyuki Koyama and Takayuki Kushida

Abstract Distributed tracing helps administrators to analyze root causes of microservices under system failure. It enables tracking procedures by log messages. Distributed trace log searches require short response times. Therefore, this study proposes a log search method with fast response time to search queries. Log messages are stored on several nodes as blocks grouped by date/time and service-name. The search method focuses on time-series access patterns and service relations. It decreases the number of accessed log messages per query on search. Experiment results show that the proposed method is maximally 0.91 seconds faster than the parallel method 'all parallel'.

1 Introduction

Microservice architecture consists of several services, which communicate themselves on a network. When one microservice responds error to another microservice, a new error occurs in another microservice [1]. A step of error propagation is called cascading failure. Since the architecture consists of a single part of the application, the root cause of errors is identified by intercepting or tracing (e.g., stack trace) in the monolithic architecture [2]. On the other hand, the stack trace can not apply in the microservice architecture since the architecture is built on several programming languages, frameworks and platforms [3]. The monolithic architecture is superior to the microservice architecture in traceability for root cause analysis.

Tomoyuki Koyama
Graduate School of Computer Science, Tokyo University of Technology, Hachioji, Tokyo, Japan, e-mail: g21210247f@edu.teu.ac.jp

Takayuki Kushida
Graduate School of Computer Science, Tokyo University of Technology, Hachioji, Tokyo, Japan, e-mail: kushida@acm.org

Distributed tracing is widely used in a microservice architecture for root cause analysis [4, 5]. Each service in a microservice generates log messages, and when distributed tracing, each message has an identifier. The identifier is generated on the service, directly receives the user's request, and can be passed to other services during the communication process. When system administrators track user requests in root cause analysis, they can find log messages that match request identifiers. As the number of microservices increases, the total number of requests for service-to-service communication increases. In addition to increasing the number of communications, the total number of log messages increases. Therefore the response time is required short with large-scale logs in search.

The scatter-gather pattern is an approach for large-scale data processing. This pattern enables large-scale data processing by sharing tasks into distributed workloads [6]. There are two types of nodes in this pattern, root nodes and leaf nodes. Root node receives tasks from users and scatters tasks to leaf nodes. Leaf nodes receive tasks from root node and respond result to it.

Prerequisite

This section describes prerequisites and a use-case scenario. The target log for this study is the web access log that enables distributed tracing on microservices. **Code. 1** shows the log format for distributed tracing generated from Envoy built-in Istio. It means that one microservice sends an HTTP request to another microservice.

Code 1 Example log message for distributed tracing.

```
[2021-10-22T00:27:09.383Z] "GET /paper/0416f705-df88-4d5f-82e8-095d4bd89e37/
    download HTTP/1.1" 200 - via_upstream - "-" 0 736954 134 133 "-" "Python
    /3.9 aiohttp/3.7.4.post0" "11c0553b-e1cd-9044-b4ce-49576dcbae6c" "paper-
    app.paper:4000" "10.42.2.65:8000" inbound|8000|| 127.0.0.6:37351
    10.42.2.65:8000 10.42.2.64:44452 outbound_.4000_._.paper-app.paper.svc.
    cluster.local default
```

A use-case of the log is root cause analysis on microservice architecture. System administrators utilize the distributed tracing log to find the root cause of system failure. When trouble is reported from a user on a web service, the system administrator executes the following search queries.

- $status_code = 400$ & $service_name =$ front
- $request_id =$ 1c0553b-e1cd-9044-b4ce-49576dcbae6

Fig. 1 shows an example of microservices. A rectangle on the figure means a microservice (e.g., front-admin, front). A cylinder means Datastore (e.g., MongoDB, Minio). This paper defines a user who accesses web services as an End User and an administrator as Admin. A microservice connects to multiple microservices in order to respond to requests from users. For instance, when End User sends a request to 'front', 'front' sends several requests to 'author' service and 'paper' service. When

’paper’ service receives a HTTP request from ’front’ microservice, ’paper’ service generates an ingress log message as shown in **Code. 1**.

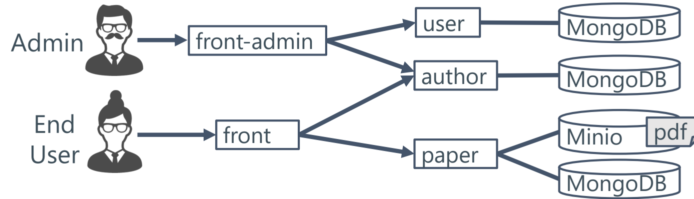


Fig. 1 Example of microservices for a paper publishing site.

Issue

The response time depends on the search query that the administrator sent in a log search on scatter-gather. This is because the search query differs from the pattern of accessing log messages on leaf nodes [7]. When the root node forwards the search query to leaf nodes by a search query, each leaf nodes finds log messages in a local volume. The placement of log messages on leaf nodes changes the search response time. Balanced log placement archives short search response time. When the log messages corresponding to a search request are unevenly distributed to several nodes, the search response time becomes slow. This is because the response time in scatter-gather takes the maximum value of the response time of the leaf node.

Fig. 2 shows an issue of log search with scatter-gather. The figure has System Administrator, Root Node, and Leaf Nodes(A, B, C, D). The rectangle inside each Leaf Node is the log message. It has an identifier as an integer number such as 10. System Administrator sends Search Request to Root Node in order to find log messages which have specified request-id (e.g., ID=10). Root Node sends Search Requests to Leaf Nodes by mapping table that resolve log into Leaf Node. For instance, Root Node accesses Leaf Node A and B in the figure. Leaf Nodes find log messages that match the search request on local storage.

The time to find log messages depends on the number of log messages on Leaf Nodes. As the number of log messages increase, the disk I/O increase on a leaf node. The pattern of unbalanced disk I/O on Leaf Nodes produces high search latency since the total search response time is equal to the maximum search response time of Leaf Nodes. For example, The search targets are three in Leaf Node B, on the other hand, zero in Leaf Node C. This is called *straggler problem* in scatter-gather. The placement of log messages is essential for fast log search. On the other hand, an actual data access pattern corresponding to search requests does not balance Leaf Nodes.

An access pattern of log messages is different by conditions such as ID or Parameters(e.g., service-name) in search query on distributed tracing. Therefore, some existing methods that utilize simple conditions clustering produce high latency on log search. This paper proposes low latency log search, which specializes in two types of query: request-id based filtering, service-name based filtering.

This paper is organized as follows. Section 2 explains existing works on distributed tracing and Information Retrieval. Section 3 presents the method to speed up a log search engine to specialize in Distributed tracing. Section 4 displays the implementation of the proposed method and describes the environment for experiments. Section 5 evaluates the results of the experiments. Section 6 discusses points of improvement and parameters on the investigation. The last section shows the contribution and the conclusion of this study.

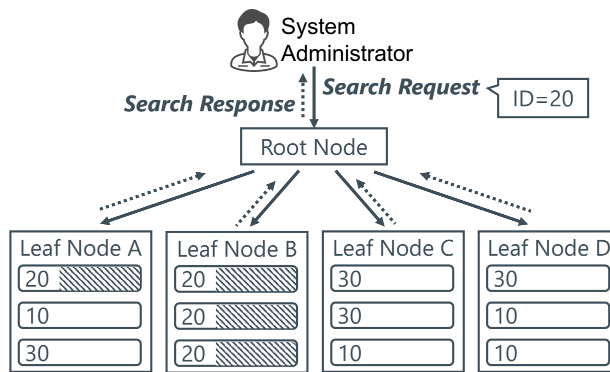


Fig. 2 Issue: access pattern and log placement.

2 Related Works

The scheduling approach to minimize job execution time finds a relation between replication factors and job execution time on Hadoop[8]. The factors in the study include the following parameters: available disk capacity, network throughput, data copying time. This approach decreases a job execution time by over 18 - 20 percent compared with existing methods. Shorting job execution time helps search large volumes of logs; nevertheless, its approach is not enough to make aware stored content. Thus, it has a possibility of decreasing job execution time by log access pattern.

CDRM proposes cost awareness dynamic replication management of storage clusters on a cloud [9]. The study focuses on relations between availability and the number of replicas. The study calculates minimum replicas that satisfy availability requirements from node capacity and blocking probability. The proposed method

improves data access latency. However, the technique could not figure out dependencies between each data. The study can improve access latency in distributed tracing since log messages have dependencies with other entries.

The heterogeneous database system produces high throughput database by SQL and NoSQL [10]. The proposal enables decreasing Disk I/O latency by Elasticsearch and MySQL. The characteristics of the proposed system (independence between transactions, single write) make it suitable for use with logs. Although, the study is not enough to be aware of the data access pattern. The method has an issue with access latency.

AptStore builds dynamic data management system for decreasing storage costs and improving Disk I/O throughput on Hadoop [11]. It enables the calculation of the probability of data accessing counts. The approach helps to match past access patterns and feature accessing design. However, the practice of data accessing is little in distributed tracing. The way has search throughput on distributed search.

The technique improves communication performance by data placement on InfiniBand [12]. The study decreases a memory registration overhead by reducing the page size of the file system. However, the works are not enough to be aware of data placement and data access patterns. Therefore, It has problems with search performance improvement.

3 Proposed Method

This study proposes the log search method for distributed tracing. The method fast responds to search queries as follows: request-id based filtering, service-name based filtering, date/time-based filtering. This study focuses on the relation between log access patterns and these parameters in search queries. The proposed approach determines the placement of log files in distributed nodes based on service-to-service relation graph and invocation order.

Fig. 3 shows an overview of the proposed log search method. The figure consists of two phases as follows: *Store Phase*, *Search Phase*. Store Phase denotes procedures from the generated log to the log stored in Leaf Nodes and Search Phase denotes procedures to find log messages that match a filter condition in query. The figure has three types of nodes: Microservices, Root Node, and Leaf Nodes. Microservices provide several users with a web service over the internet. Root Node processes log messages, scans service relations, and forwards log messages to Leaf Nodes based on the placement rules. Leaf Nodes store log messages as log blocks and respond to search requests on log search. The flow of logs generated by the microservice and stored in Leaf Nodes is described below.

1. Microservices forward log messages to Root Node as log files.
2. Root Node converts log files into blocks.
3. Service Scanner discovers microservices.
4. Service Scanner generates Service Relations from discovered microservices.

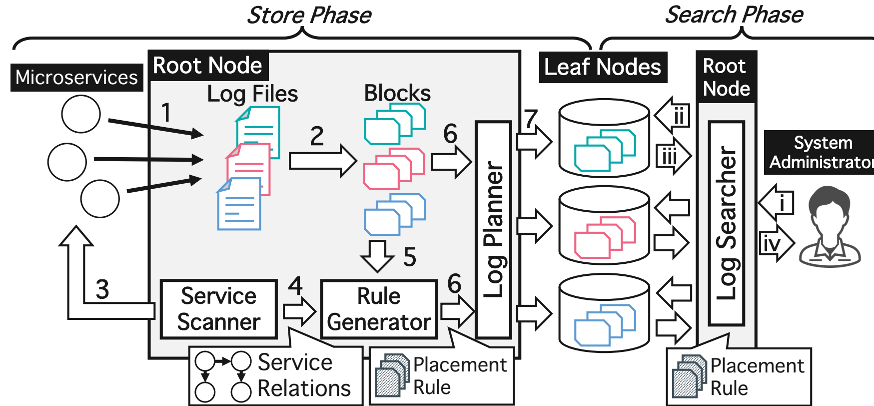


Fig. 3 The proposal overview.

5. Rule Generator loads blocks and service relations.
6. Rule Generator writes the placement rule.
7. Log Planner forwards blocks to Leaf Nodes based on the placement rule.

The method works the followed flow when the System Administrator executes a search query with filtering conditions.

i. System Administrator sends a search query to Log Searcher on Root Node.
ii. Log Searcher scatters search requests to Leaf Nodes.
iii. Leaf Nodes finds log blocks and sends a search response to Log Searcher.
iv. Log Searcher gathers search responses and sends the search result to System Administrator.

3.1 Rule Generator

This section describes generating the log placement rule. The rule aims to decrease access blocks on log search for reducing search response time since the number of access blocks is equal to search response time. The Rule Generator gets two inputs such as **Service Relations**, **Blocks** and outputs the **Placement Rule**.

Service Relations: The Service Relations has a graph structure and represents relations in microservices. When one microservice α invokes another microservice β, microservice α has a service relation to microservice β. The relation displays as $\alpha \rightarrow \beta$. The proposed method utilizes a service-to-service relation within microservices for blocks placement. The way to get service relation is service mesh. A service mesh supports a service discovery that enables to find microservices and their relations [13]. Istio is used for service discovery in this study [1].

[1] https://istio.io/

Blocks: The Rule Generator takes the Blocks as input. The Blocks are made from Log Files based on date/time and service-name to reduce the number of access blocks on search. For example, the service-name that the file name contains is "front," and the date/time are "2021-10-22T00:27:09.382Z" in **Code. 1**. A block length takes a fixed file size based on the date/time range, such as 12 hours. The steps of making blocks are as follows: 1) clustering log files by service-name; 2) concatenating log files per service-name and splitting concatenated files as a fixed size block.

Placement Rule: The Rule Generator outputs the Placement Rule for Leaf Nodes. The rule manages the pairs of blocks and Leaf Nodes. When the Root Node finds the leaf nodes that have log blocks matched search conditions, the Root Node utilizes the Placement Rule. The format for the rule has pairs of key-value structures. The key takes a block name, and the value takes node-name, service-name, and date/time. The steps of making the Placement Rule are as follows. The number of Leaf Nodes is defined as N. Each Leaf Nodes is assigned a serial number from 1 to N. B is the set of blocks as follows: $b \in B$. b is one of the blocks in B and is split by service-name and date/time.

1. Sort all blocks based on service-name and date/time in a log message.
2. Allocate blocks from the sorted block list to the leaf nodes while the number of iterators i increases. Select a block from the list. The leaf node for storing blocks is determined by modulo arithmetic. The serial number i that corresponds to a block is formulated as $mod(i,N)$.

Fig. 4 shows an example of log placement where $N = 4$. The figure has three services as Service A, Service B, and Service C. A block in the figure has service-name and date/time range. For example, the top left block means that service-name is 'Service A', and date/time range is from 1 to 3.

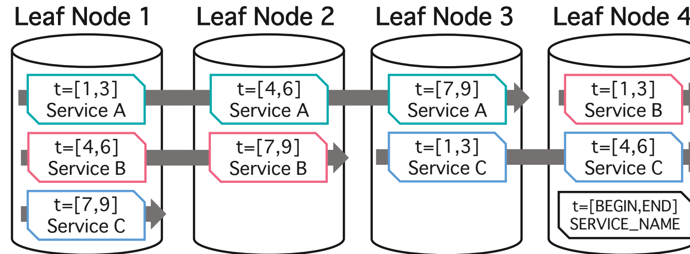


Fig. 4 Proposed log placement on 4 leaf nodes.

3.2 Log Searcher

This section describes the search method to decrease the number of accessed logs on log search. **Fig. 6** shows the method of proposed log search. The figure displays log blocks per service on microservices. The color of the block indicates whether to scan on the log search. The x-axis indicates elapsed time. The services such as Service A consist of a web service. The dotted lines indicate search target date/time range (e.g. from November 9, 2021, to December 7, 2021). When a user accesses Service A by Web Browser, Service A accesses Service B and accesses Service C to build a response. The following section describes the search procedure.

1. Find log messages in the block of Service A that locates the top of service relations when the system administrator finds log messages in log blocks parallelly.
2. Find the first message that matches the search condition in the search result of Service A.
3. Get the date/time from the first log message.
4. Find the log block that matches the first date/time in 'Service B that is the next service from Service A.
5. Repeat steps until no next service.

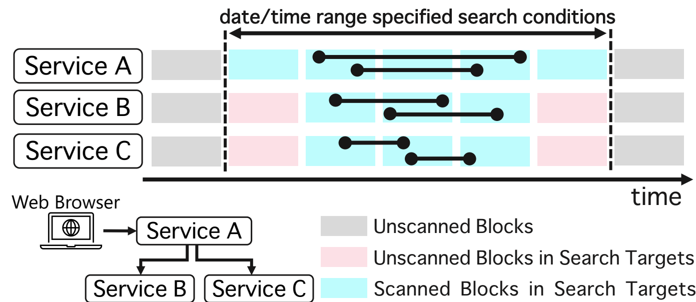


Fig. 5 Proposed log search.

Fig. 6 compares the steps of block access in log search between the parallel method 'All Parallel Search' and the proposed method 'Proposed Search'. The proposed method finds only green blocks. The parallel method finds green blocks and red blocks. A number in blocks mean access order in search. The number of access blocks is related to the search response time. The proposed method reduces the search response time. The characteristic of this study reduces the number of access blocks on log search based on time-series access and service relations. Reducing access blocks enables to reduce the search response time since the time is related to disk I/O.

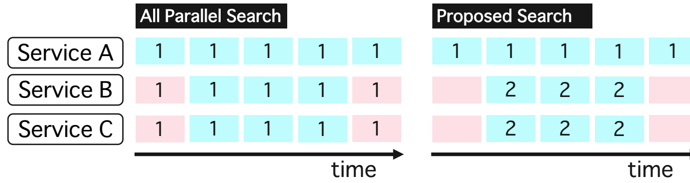


Fig. 6 Flow of accessing blocks.

4 Implementation

Fig. 7 shows a system architecture. The figure has two types of nodes: Root Node and Leaf Node. The Root Node generates Log Files and creates blocks from log files and forwards blocks to Leaf Nodes based on Placement Rule. The Placement Rule is implemented as JSON format as shown **Code. 2**. This rule takes block name as key, '20.author_block_.log.004' and takes block metadata as value. **Code. 4** shows the block for 'author' service from 'begin_datetime' to 'end_datetime' is stored on 'koyama-log1'. The Leaf Nodes receive blocks from the Root Node and stores blocks in local storage. The effect of network delay is small since the traffic of node communication is less than network bandwidth.

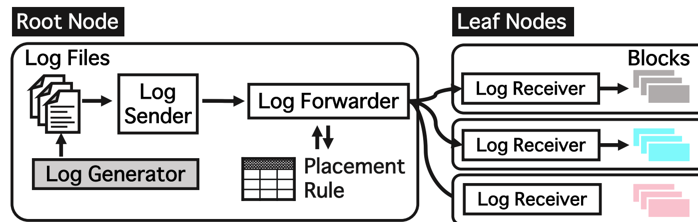


Fig. 7 The system architecture.

Code 2 Example of Placement Rule

```
{"20.author_block.log.004": {
    "begin_datetime": "2023-07-24T16:10:09",
    "end_datetime": "2023-12-28T18:42:21",
    "nodename": "koyama-log1",
    "servicename": "author"
}}
```

5 Experimental Results

Fig. 8 shows an evaluation environment. The environment works on BareMetal Hardware with VMware ESXi 7.0 Hypervisor. Root Node runs on a single Virtual Machine. Leaf Nodes works on 13 Virtual Machines. Two types of nodes have homogeneous hardware resources (CPU: 1[Core], RAM: 1[GB], Storage: 30[GB]). Log Generator expands log messages from 1,600 to 8,065,000. The log message reproduces the behavior of the microservices in **Fig. 1** (i.e. 'front', 'paper' and 'author'). **Fig. 9** shows the evaluation method. The figure has two types of nodes: Root Node and Leaf Node. User sends a Search Query to the Root Node and receives the Search Result from the Root Node. The Root Node sends search requests to the Leaf Nodes (i.e. Leaf Node1, Leaf Node2) and receives search responses. The evaluation measures the search response time from the Search Requests sent till the Search Responses received.

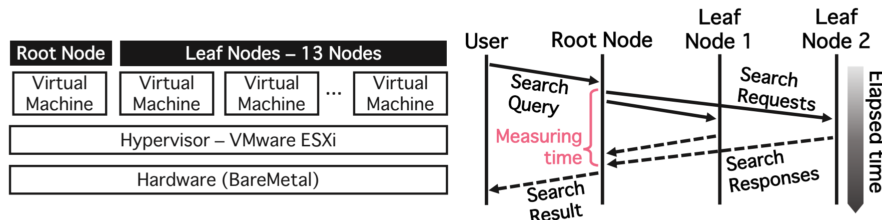


Fig. 8 Experiment environments.

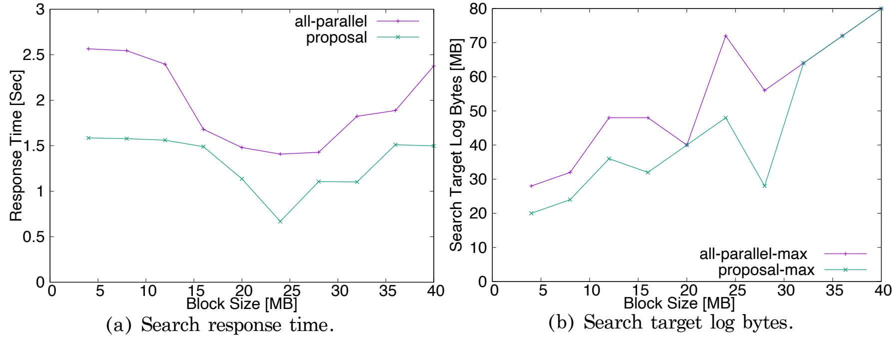
Fig. 9 The way to measure the response time.

Comparison of search response time and search target log bytes while block size increases: **Fig. 10** compares the response time in (a) and the maximum search target log bytes in (b) while the block size changes. The figure compares the parallel search method (*all-parallel*) to the proposed search method (*proposal*). The parallel search method finds all log blocks that match the date/time range. The proposed search method finds log blocks that match the date/time range. The search query has date/time-based filtering and keyword-based filtering as shown in **Code. 3**. The length of a log message is around 400 [Bytes]. The response time is the median of 10 measurements, which is measured by the time module in Python.

Code 3 The search query for evaluation: filtering block size and date/time range.

```
s_bs=8, s_dt_begin="2021-12-13T10:21:50", s_dt_end="2023-02-13T10:21:50"
```

Fig. 10 (a) compares the search response time while block size increases. The x-axis is block size (unit: MB), and the y-axis is search response time (unit: seconds). The block size takes following values: $\{4, 8, 12, 16, 20, 24, 28, 32, 36, 40\}$. The response time of the proposed search method is 0.91[sec] faster than the parallel search method maximally, where the block size is 4[MB]. The response time depends on the block size. As the block size increases from 4 [MB] to 24 [MB],

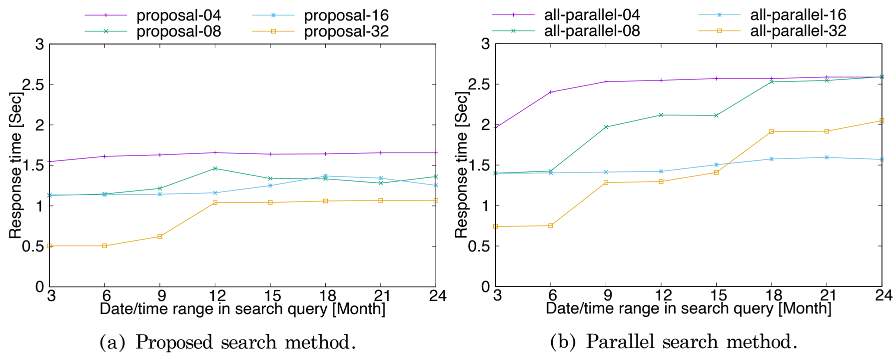


(a) Search response time. (b) Search target log bytes.

Fig. 10 Comparison of the block size.

the response time reduces. Response time gets shorter by reducing the total number of opened files. The existing study shows the same result [14]. As the block size increases from 24 [MB] to 40 [MB], the response time increases. The increase of response time results from increasing access target blocks per leaf node. As the block size decreases, the number of allocated blocks per leaf node decreases. On the other hand, increasing the block size increases the difference in the number of stored logs per leaf node compared to small block size. Thus, the difference of accessed blocks counts makes the response time increase.

Fig. 10 (b) compares the maximum search target log bytes while block size increases. The x-axis is block size (unit: MB), and the y-axis is the maximum total search target log bytes in Leaf Nodes (unit: MB). The block size takes followed values: {4, 8, 12, 16, 20, 24, 28, 32, 36, 40}. As the block size increases, the maximum search target log bytes increases. The increasing response time is caused by the maximum search target log bytes increasing between 24[MB] and 40[MB] in **Fig. 10 (a)**.



(a) Proposed search method. (b) Parallel search method.

Fig. 11 Comparison of the log placement method.

Comparison of search response time while the number of target log bytes increases: **Fig. 11** compares the number of search target log bytes between (a) Proposed search method and (b) Parallel search method. The figure aims to recognize the effectiveness between the number of search target blocks and the search response time. The x-axis is the date/time range (unit: Month) which is specified in search query. It takes followed values: $\{3,6,9,12,15,18,21,24\}$. For example, $x = 3$ means to find log messages within 3 months. The y-axis is response time. The legends indicate block size (unit: MB) and take followed values: $\{4,8,16,32\}$. The response time is the median of 50 measurements, which is measured by the time module in Python. **Fig. 11 (a)** shows that the response time remains constant as the date/time range increases. The proposed method keeps the number of access blocks constant on search when the date/time range is extended in the search query. The search response time decreases as the block size increases from 4[MB] to 32[MB]. A large block size reduces Disk I/O overhead on a Linux file system. In order to decrease file system overhead, GFS which is created by Google adopts 64 [MB] as a block size [15]. **Fig. 11 (b)** indicates that the response time increases in the parallel search method with the date/time range increased. The response time depends on the number of accessing blocks on search. The number of accessing blocks increases on the parallel method as the search query's date/time range expands. **Fig. 11 (a)** and **(b)** indicate that the proposed search method faster than the parallel method in search response time. The reason of differ response time is that the proposed method is less than the parallel method in the number of accessing blocks on log search.

6 Discussion and Conclusions

The proposed method sets fixed block size. Thus, the number of log messages per block is homogeneous. While the data size of the block grows, the data size grows sequentially. The file size which can be read and written simultaneously depends on Disk I/O performance per node. Thus, block size has to be calculated from Disk I/O performance. One of the methods is *iostat* command which gets I/O performance.

Default length of block size is 16[KB] on existing database systems such as MySQL(InnoDB) and PostgreSQL. This is because page cache size of file system uses multiple of 4[KB]. For example, considering the case where a 16[KB] block contains the format as shown in **Code. 1**. When the length of a log message is 400[KB] as shown **Code. 1**, a block contains 40 messages ($16,000/400 = 40$).

As block size increases, the response time is short. On the other hand, increasing the block size makes the placement problem. As the block size decreases, the number of log messages allocated on leaf nodes increases due to decreasing free space in the leaf node's local disk. Block allocation algorithm requires not only the block size but also free disk space on leaf nodes.

The issue is the slow response time for distributed tracing on the log search system. This study aims to reduce the search response time of two types of queries: filtering status-code, request-id). This study proposes a fast log search method for dis-

tributed tracing. The method makes the log blocks based on date/time and service-name in log messages. The log blocks are stored on several distributed nodes and retrieved on search. Experiment results show that the response time of the proposed method is maximally 0.91 seconds faster than the method of all parallel. This study contributes to reducing search response time on log search.

References

1. F. Montesi and J. Weber, "From the decorator pattern to circuit breakers in microservices," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1733–1735.
2. S. Mallanna and M. Devika, "Distributed request tracing using zipkin and spring boot sleuth," *International Journal of Computer Applications*, vol. 975, p. 8887.
3. C.-Y. Fan and S.-P. Ma, "Migrating monolithic mobile application to microservice architecture: An experiment report," in *2017 ieee international conference on ai & mobile services (aims)*. IEEE, 2017, pp. 109–112.
4. A. Bento, J. Correia, R. Filipe, F. Araujo, and J. Cardoso, "Automated analysis of distributed tracing: Challenges and research directions," *Journal of Grid Computing*, vol. 19, no. 1, pp. 1–15, 2021.
5. M. Santana, A. Sampaio Jr, M. Andrade, and N. S. Rosa, "Transparent tracing of microservice-based applications," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1252–1259.
6. C. Alvarez, Z. He, G. Alonso, and A. Singla, "Specializing the network for scatter-gather workloads," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 267–280.
7. A. Dan, S. Y. Philip, and J.-Y. Chung, "Characterization of database access pattern for analytic prediction of buffer hit probability," *The VLDB Journal*, vol. 4, no. 1, pp. 127–154, 1995.
8. H. E. Ciritoglu, L. Batista de Almeida, E. Cunha de Almeida, T. S. Buda, J. Murphy, and C. Thorpe, "Investigation of replication factor for performance enhancement in the hadoop distributed file system," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 135–140.
9. Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng, "Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster," in *2010 IEEE international conference on cluster computing*. IEEE, 2010, pp. 188–196.
10. U. Taware and N. Shaikh, "Heterogeneous database system for faster data querying using elasticsearch," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–4.
11. K. Krish, A. Khasymski, A. R. Butt, S. Tiwari, and M. Bhandarkar, "Aptstore: Dynamic storage management for hadoop," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1. IEEE, 2013, pp. 33–41.
12. R. Rex, F. Mietke, W. Rehm, C. Raisch, and H.-N. Nguyen, "Improving communication performance on infiniband by using efficient data placement strategies," in *2006 IEEE International Conference on Cluster Computing*. IEEE, 2006, pp. 1–7.
13. W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225.
14. T. L. S. R. Krishna, T. Ragunathan, and S. K. Battula, "Performance evaluation of read and write operations in hadoop distributed file system," in *2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming*. IEEE, 2014, pp. 110–113.
15. S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.