Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

High Speed Error Log Control Method in In-memory Cluster Computing Platform

Ryuichi Saito[1,a)] Shinichiro Haruyama[1,b)]

Received: September 5, 2019, Accepted: January 9, 2020

Abstract: Since 2010, in-memory cluster computing platform has been increasingly used in firms and research institutions to analyze large amounts of datasets within a short amount of time. In these methods, unexpected errors cause the load to exceed the assumption for computer infrastructures such as a monitoring system, owing to the execution of multithreading, assigning divided datasets to multiple nodes, and storing them in in-memory spaces. In this research, we propose a method that notifies administrators with only information needed to understand the situation in a short period by eliminating duplications of numerous application error logs for that period and clustering messages using an unsupervised learning k-means method with an in-memory cluster computing framework "Apache Spark." By implementing this method, we can demonstrate that it is possible to eliminate duplications of error messages by 93% on an average compared with conventional methods. Further, we can extract significant messages from the application error messages and notify the administrators in an average of 4.2 min from the time of occurrence of the error.

Keywords: error logs, TF-IDF, k-means, Distributed System, Spark

1. Introduction

Since 2010, companies have increasingly been analyzing large numbers of collected datasets statistically and utilizing them in order to implement behavioral targeting advertisements and biometoric recognition with fourth-generation mobile network technologies. Research institutions also utilize such analysis in tandem with high performance computer (HPC) to perform simulations for drug discovery or to predict climate changes. And these data often have the property of streaming data generated continuously. To bring these to fruition, the following functional programming model such as MapReduce [1] in 2004 is widely used. (1) Divide the dataset consisting of key-value pairs into a predetermined number and pass over to a worker node composed of multiple computers starting with. (2) Perform processes such as totaling on the worker node and save the intermediate results on the local disk of the computer (Map Task). (3) List the process results of (2) in the master node, and save it to the file as persistent data (Reduce Task) [2].

By adopting such methods, enterprises and research institutions can easily construct large-scale clusters on general-purpose computers and process large numbers of datasets. Further, engineers can easily implement a parallel/distributed information system as the framework has abstracted functions that originally require considerable effort in design such as parallel processing, fault tolerance, data distribution, and load balancing. Furthermore, Spark in 2014 achieved higher performance and latency by basing the MapReduce model and using the resilient distributed datasets (RDD) collection that share data in the memory across clusters [3]. During the execution of such cluster computing, the total number of threads is simply represented by the expression (1), as multithreading is performed using multiple nodes.

$$TotalThreadCount = NodeCountPerCluster * ThreadCountPerNode \quad (1)$$

NodeCountPerCluster indicates the number of nodes that make up the cluster and *ThreadCountPerNode* indicates the number of threads executed on one node.

While multiple threads can achieve high throughput by referring low-latency in-memory data, numerous error logs will be output as files owing to programming errors or input data that was not anticipated by the designer, and the following problems may occur.

- Notification to the administrator is delayed as the error log reading of the monitoring system stagnates.
- It becomes difficult for administrators to analyze error logs in a short time.
- Conventional incident management methods can't be applied because errors and incidents simply do not link one on one.

Accordingly, we poropose the methods that enables the information system administrater to understand the failure situation in a short time through summary of redundant error messages and notification in time according to the user's service level.

This method resolves the operational problems in the in-memory cluster computing in the following steps.

(1) Collect the error messages output by the worker node, format the collected message, and deduplicate these messages to obtain only the necessary messages. The monitoring sys-

[1] SDM, Keio University, Yokohama, Kanagawa 223–8526, Japan
[a)] ryuichi.saito@keio.jp
[b)] haruyama@sdm.keio.ac.jp

tem refers to these deduplicated messages, thereby reducing the load.

(2) For messages that cannot be deduplicated, convert the message string to a feature vector using the TF-IDF algorithm, and then perform clustering using an unsupervised learning k-means algorithm. Among the grouped messages, those nearest to the centroid of the cluster are defined as representative values and extracted.

(3) Send EMail notification to the administrator after adding up the messages dedpulicated in (1) and messages extracted in (2).

These methods can be implemented in a short time using in-memory computing Spark application and these implementation assumes general purpose use case.

For the evaluation, we deployed the Spark application that implemented the proposed methods in Amazon Elastic MapReduce and compared it with conventional methods. As a result, it was verified that it is possible to achieve the deduplication of 93.2% on average for the output error messages, and only representative messages including deduplicated messages can be notified to the administrator in an average of 4.2 min after the error messages are output to the files. Although this research is focused on application error log including applicaiton framwork error log on in-memory cluster computing environment, these method can be applied for various types of logs with pre-defined format such as OS log, database log and hardware log.

2. Relevant Study

2.1 Error Detection

Research has been conducted on methods to efficiently detect errors in large-scale systems in conjunction with American super computer projects. Iyer et al. [4] sequentially grouped error logs into time series, mutual correlation, and repeated patterns in a multiprocessor system environment of IBM in a statistical way in 1990, and proposed a method to isolate grouped error logs into temporary signs or ongoing signs, and then ongoing signs into single or multiple signs.

Zheng et al. [5] categorized hardware error events such as Cache Failure and DDR Register Failure in the supercomputer environment of Cray XT 4 and IBM Blue Gene/ L, and devised a method that extracts errors occurring at the same time and defines them as the same cause, after excluding the redundancy of similar errors that occurred in different places.

Gurumdimma et al. [6] created a matrix of message types and window times in the IBM Blue Gene/L environment in the same way as Zheng et al., and tried to improve the error detection efficiency in a large-scale parallel/distributed system by deriving an approximation between message sequences, based on the Jensen-Shannon divergence algorithm.

2.2 Streaming Data Process

Research on methods to process large-scale data generated continuously have been mainly conducted by the enterprises that actually have to process massive data. Toshniwal et al. [7] proposed Storm as a tool for aggregating and analyzing Tweets that are massively and continuously generated on Twitter platform. Storm achieves fault-tolerant streaming data process by expressing the data source as Spouts and process as Bolts and executing Spouts and Bolts in parallel on the Directed Acyclic Graph (DAG) topology across the worker nodes in the cluster.

Meanwhile, Xin et al. [8] also executed SQL queries on Spark's in-memory storage RDDs that were also designed on the DAG topology, and improved the latency of data extraction, which has been regarded as a trade-off of data distribution in the conventional MapReduce. This research facilitated data extraction/processing during the streaming data process.

Armbrust et al. [9] developed relational database queries and high-performance SQL code generation engines in Spark's streaming data process using high-level API structured streaming, and expanded the range of applications to business applications of the streaming data process executed on the DAG topology.

2.3 Message Categorization

Research on message categorization can be considered from two stages: information retrieval and clustering of retrieved data. Regarding information retrieval, TF-IDF is widely used as an algorithm for evaluating the importance of words in corpus documents, but in data streams, there are problems that it is not possible to define knowledge of static document collections and available memory size is smaller than the size of data stream. Erra et al. [10] proposed Approximate TF-IDF method based on Sort-based Frequent Items algorithm [11] which finds an approximate solution from preset memory size in continuous data stream to solve these problems.

k-means is prevalent as clustering algorithm for message categorization. Meng et al. [12] developed machine learning library: MLlib on Apache Spark which includes k-means method to cluster large-scale data. MLib on Spark could be used integrated with other Spark APIs to operate large-scale data such as Spark SQL.

Svyatkovskiy et al. [13] provided analysis workflow to detect policy diffusion of legislative proposals in the United States on Spark ML. On this research, they proposed k-means as a method to categorize each vectors which was converted using TF-IDF method from the raw text for the purpose of estimating the similar diffusion topics.

3. Problems of Current Methods

3.1 Problems of Error Detection

As the problem of current error detection, the trade-off relationship between immediacy and message extraction that administrator can understand is thought as a theme to be solved. The error message in the information system is immediately notified to administrators via the monitoring tool after the file output. Administrators scrutinize the notified messages and isolate the incidents in a short period of time whether they should ignore messages owing to assess these messages don't affect service continuity or require any urgent actions.

Meanwhile, in the case of a system that clustered by a plurality of nodes and executed by multithreading per node, a large number of error messages are issued when an error of the same cause such as incorrect input data occurs. In that case, the administrator that receives a large amount of messages through the monitoring

tool need to spend much time and workload to isolate the incident. Also, the approach of extracting messages that administrator can understand from a large number of error messages based on statistical rules such as Jensen-Shannon divergence algorithm does not conform to error detection required urgency because it is necessary to analyze error messages for a fixed time. Therefore, there is a need for a solution that analyzes a large number of error messages at short time intervals and notifies the administrator immediately.

3.2 Problems of Streaming Data Process

Regarding operation of streaming data process on the parallel/distributed system, aggregation of logs that are output on each node is considered as a major issues. Clustered system consists of a plurality of nodes, so output logs on each node need to aggregate to single data storage for analysis. At same time, output logs on streaming data process need to be delivered through same kind of streaming data process tool in order to suppress retency of log processing. Once output logs have to be persisted in local disk on each node for troubleshooting, these logs are also collected to single data storage through data collection tool immediately. Although there are log collection tools like FluentD [14] or Kafka [15], log collection platform connected these tools each other asynchronously need to be designed.

3.3 Problems of Message Categorization

There are two problems of message categorization in error detection. First is the number of clusters set in advance that categorize error messages according to the type of message. The cluster number depends on number of triggers for message output because administrator that responds to failures have to reach their cause. These cluster number need to define based on some rules or machine learning methods.

Second is the message to notify administrator that is extracted from clustered messages. Clustered messages have different properties depending on the distance from the centroid of cluster. Messages need to be extracted randomly or based on some rules from each cluster according to actual use case.

4. Proposed Method

4.1 Proposal

The proposed method solves the following problems regarding in-memory cluster computing.

(1) A large amount of error messages are output to local disk at application failure in a short time, so monitoring tool can't read these messages.
(2) It is difficult for administrators to isolate failures from a large amount of error messages in a short period.
(3) Architecture design in applicaiton log management of in-memory cluster computing system isn't clarified generally.

This proposed method will de-duplicate the error messages output to storage to the extent that they don't load the monitoring tool, and summarize the deduplicated messages and notify the administrator in a short time to analyze failures. At the same time, this proposal present system design to manage applicaiton log in in-memory cluster computing system.

4.2 System Design

The proposed method collects several messages that are simultaneously output to files from multiple nodes in a cluster computing environment that executes the process by referring to in-memory data, and extracts and notifies the representative error messages after deduplicating the redundant error messages and clustering messages.

To deduplicate error messages, a label is given to the error message string and an aggregate query is executed based on the label key information. Further, the clustering of error messages is performed using the k-means algorithm after converting messages into a feature vector using the TF-IDF algorithm in units of windows (tabulation frames), and the message located at the Euclidean distance nearest to the centroid of the cluster is defined as the representative value, and then extracted and notification is sent via EMail. The relationship between functions in the proposed method and in the external system is summarized in the sequence diagram in **Fig. 1**.

Log Collector collects error meseages from each node on cluster and sends their messages to *Log Processor*. *Log Processor* pools streaming error messages on pub/sub message queue. *Log Deduplicator* subscribs error messages from *Log Processor* and deduplicate their messages. Deduplicated messages are stored on Amazon S3 as object storage. *Log Marshaller* picks out dudu-plicated messages from Amazon S3 storage and summarize their messages as representative values throuth TF-IDF and k-means clustering method. Summarized messages instantly notify administrator by EMail using Amazon SNS service. The above is a series of flows of this proposal.

Only *Log Collector* runs on the application server that outputs logs in a production environment. All functions except for *Log Collector* run on another servers. Since *Log Collector* is FluentD developed on the assumption that it is executed concurrently with the application server, this system has very little impact on the resources of the collocated application that outputs logs. Although this architecture is designed on the premise of a cloud platform and will be verified on a cloud platform, it is possible to applly for on-premise platform used similar open source architectures.

4.3 Message Deduplication

4.3.1 Labeling and Data Collection

Labels are given to error messages that are converted and output according to a certain rule by *Log Collector* as shown in **Table 1** when the error messages are collected to deduplicate error messages using aggregate queries.

Here, a delimiter was used when labeling a blank character string in the message sequence. Data are collected using FluentD as *Log Collector*, and then streamed to the Kafka message queue of *Log Processor* in JSON format, after assigning labels to multiple log lines using FluentD's Multiline Parser Plugin.

4.3.2 Message Deduplication

For the deduplication of messages, we use Spark Streaming to acquire error messages in JSON format from Kafka message queue of *Log Processor* in a 2-minute cycle and perform aggregation and deduplication on *Log Deduplicator*. As shown in Algorithm 1, error messages from the Kafka message queue are ob-

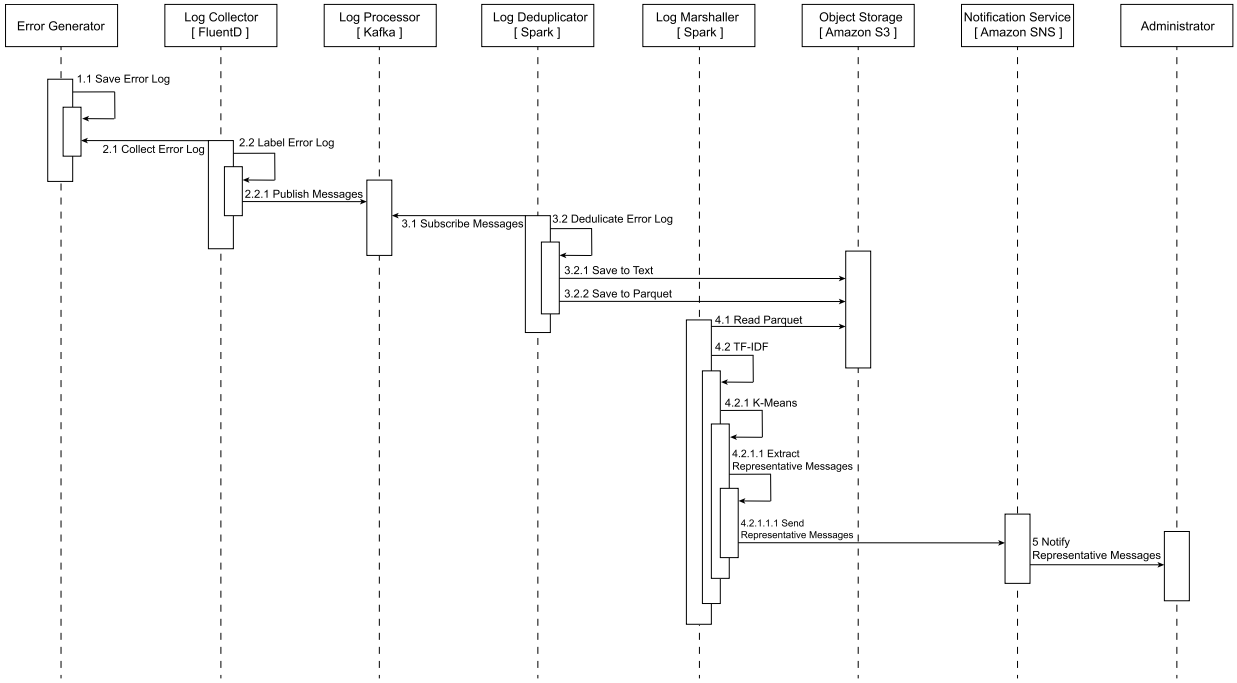

Fig. 1 Sequence diagram.

Table 1 Message label.

Label	Message Sequence
Date Time	18/03/09 14:58:52
Level	ERROR
Thread ID	`Streaming$:`
Message Body	EXP-E000001 “Tweet word is too long”
Stack Trace 1	jp.ac.keio.sdm.ConcurrentLogControl.WordLengthException:
Stack Trace 2	`at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:51)`
Stack Trace 3	`at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:27)`
Stack Trace 4	`at scala.collection.Iterator$$anon$11.next(Iterator.scala:409)`
Stack Trace 5	`at scala.collection.Iterator$$anon$11.next(Iterator.scala:409)`
Stack Trace 6	`at org.apache.spark.util.collection.ExternalSorter.insertAll(ExternalSorter.scala:194)`
Stack Trace 7	`at org.apache.spark.shuffle.sort.SortShuffleWriter.write(SortShuffleWriter.scala:63)`
Stack Trace 8	`at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:79)`
Stack Trace 9	`at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask.scala:47)`
Stack Trace 10	`at org.apache.spark.scheduler.Task.run(Task.scala:86)`
Stack Trace 11	`at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:274)`
Stack Trace 12	`at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)`
Stack Trace 13	`at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)`
Stack Trace 14	`at java.lang.Thread.run(Thread.java:745)`

tained as streaming data in step 2, and then error messages are aggregated/dedpulicated using Spark SQL after converting the streaming data to Spark Dataframe for each RDD function after step 3. Input data is interpreted as structured data consisting of rows and columns on Spark Dataframe. The error messages deduplicated in step 8 are output to Amazon S3 storage in Parquet format.

On the contrary, all error messages before deduplication are also output to Amazon S3 storage at the same time into text file format for archiving. Because messages before deduplication are needed as an applicataion trail.

4.4 Representative Value Extraction

Execute *Log Marshaller* as a Spark application to read the Parquet file from Amazon S3 storage and convert it to Spark Dataframe. Next, perform extraction process according to the number of cases as follows, and notify the results to the administrator.

(1) If duplicated error messages are greater than two items, extract all deduplicated messages
(2) In case of an error message without duplication, perform clustering and extract only representative messages

4.4.1 Extraction of Duplicate Error Messages

Extract errors greater than two items using the filter method of Spark Dataframe. Then, combine the Dataframe string of error message that was first output to the file with the extracted message using Spark SQL, and extract the original message.

4.4.2 Extraction of Error Messages without Duplication

Perform clustering after converting the message character strings excluding the error messages extracted in Section 4.4.1 to the feature vector, and extract only the error message at Euclidean distance nearest to the centroid of cluster.

4.4.2.1 Feature Extraction

Convert the error message string to feature vector using the TF-

Algorithm 1 Message Deduplication algorithm

```
1: function MAIN
2:     ds ← readKafka
3:     ds.foreachRDD(JsonRDD => {
4:     df ← readJsonRDD
5:     if df.count > 0 then
6:         result ← df.groupBy(
               level
               ,thread_id
               ,stack_trace_1
               ,stack_trace_2
               ,stack_trace_3
               ,stack_trace_4
               ,stack_trace_5
               ,stack_trace_6
               ,stack_trace_7
               ,stack_trace_8
               ,stack_trace_9
               ,stack_trace_10
               ,stack_trace_11
               ,stack_trace_12
               ,stack_trace_13).agg(min(Message))
7:     end if
8:     result.write(S3FileDirectory)
9: end function
```

IDF library of SparkML. The number of feature dimensionality is set to 10,000 against the default value of 262,144 in Spark's TF-IDF library. This setting is based on the assumption that only messages that could not be deduplicated are subject to clustering, and that many character strings are repeatedly output in the log characteristics. The expressions are shown in Eqs. (2) and (3).

$$IDF(t,d) = \log \frac{|D|+1}{DF(t,D)+1} \tag{2}$$

Document Frequency (*DF*) represents the number of documents d including words t. The DF value increases as words with less information such as "the" and "of" frequently appear in corpus D. Dividing DF from the total number of documents $|D|$ in the corpus in *inverse document frequency* (*IDF*) is interpreted as having special information in certain documents for words that do not appear frequently in the corpus. The value increases as long as the word is interpreted as special.

$$TFIDF(t,d,D) = TF(t,d) * IDF(t,D) \tag{3}$$

Term Frequency (*TF*) represents the number of word t appearing in document d. $TFIDF(t,d,D)$ is obtained by multiplication of TF and IDF, and the importance of a word appearing in a document in the corpus can be indicated by a feature.

4.4.2.2 Clustering Method

Clustering is performed for error messages converted to the feature vector of TF-IDF using Spark ML unsupervised learning k-means method library. The equation for determining the centroid of each cluster is shown below.

$$C^{(i)} := \arg\min_{j} \|x^{(i)} - \mu_j\|^2 \tag{4}$$

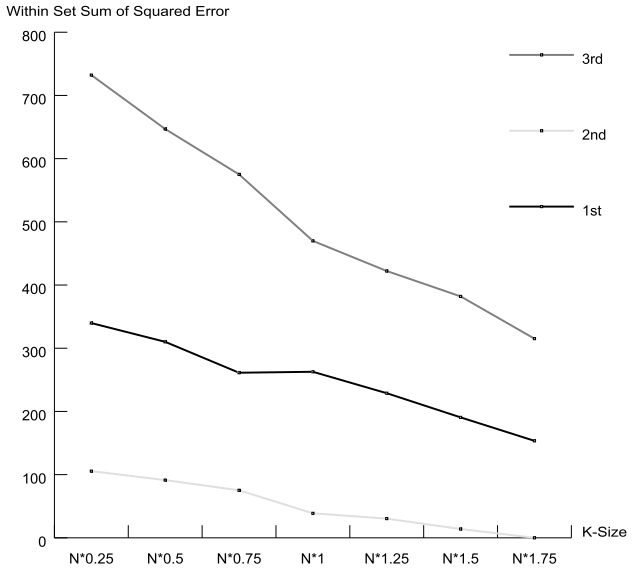

Fig. 2 Within set sum of squared error.

$\|x^{(i)} - \mu_j\|$ represents the Euclidean distance between each data point $x(i)$ and the arbitrarily allotted cluster centroid μ_j, and the process is repeated until the cluster centroid $C^{(i)}$ with the shortest distance is allocated to each data point. The Spark ML library performs the initialization process to optimize the allocation of cluster centroid μ_j and uses the k-means|| algorithm to speed up the initialization process when processing large-scale data, as the centroid of the first cluster depends on the arbitrary allocation in this algorithm [16]. The clustering image is shown in **Fig. 3**. Vertical and horizontal axis in the Figure mean a measure of the distance between each data point. Each error message converted into a feature vector is plotted as a data point and grouped based on the Euclidean distance from the centroid, which is the number of the predetermined cluster size k.

4.4.2.3 K Size

In evaluating the k size, the Within Set Sum of Squared Errors (WSSSE) was computed using input data. WSSSE is the sum of the square error of the set of input vectors and the centroid of the cluster to which it belongs. The value of WSSSE has the nature decreasing as the value of k increases. The number of clusters where the decrease in WSSSE converges is the optimal k. Convergence point is expressed as "elbow".

The k size was designed by multiplying the number of deduplicated messages in Section 4.3.2 by a factor. Because the number of output error messages changes dynamically according to the failure status. It was assumed that the number of deduplicated messages was proportional to the volume of output messages.

Figure 2 shows the results of calculating WSSSE using three different batch data that were deduplicated. The transition was shown by multiplying the number of each message by a factor, but a convergence point enough to be seen as an "elbow" could not be found. On the other hand, in the range of "N * 0.75" to "N * 1" of the 1st evaluation, it seems that it is more convergent than other ranges, Therefore, in this method, the value multiplying the number of deduplicated messages by 0.75 is defined as the dynamic k size.

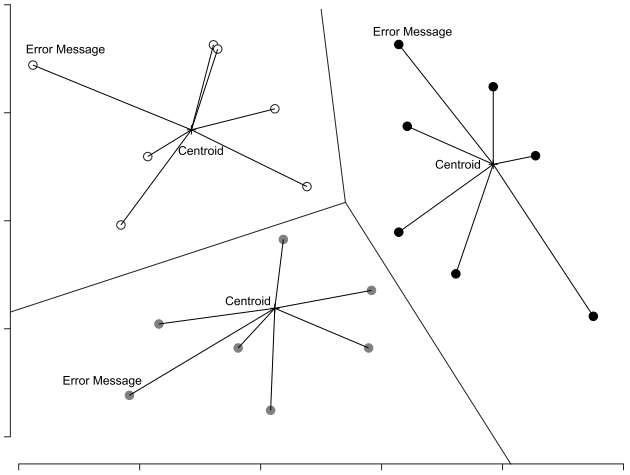

Fig. 3 Message clustering.

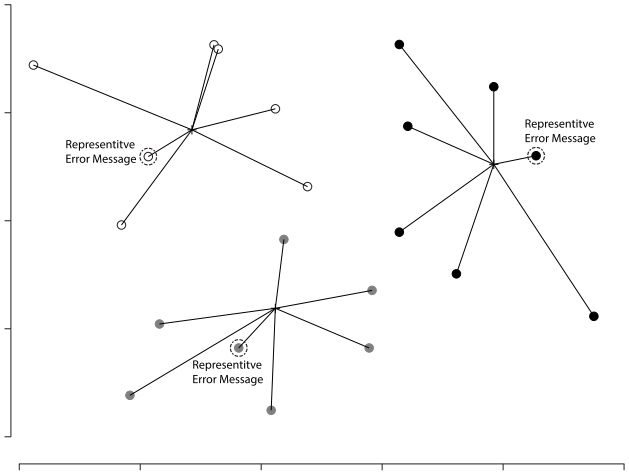

Fig. 4 Message extraction.

Algorithm 2 Representative Extraction algorithm

```
1:  procedure MAIN
2:      centroids ← model.clusterCenters
3:      distance ← df.withColumn("square_distance")
            , CalDistance(col("prediction")
            , col(features))).distinct()
4:      nearestdistance ← distance
            .groupBy("prediction")
            .agg(min("square_distance"))
5:      for i ← 0, ksize − 1 do
6:          if  nearestdistance.groupBy("prediction").count()
          .filter(finalData("prediction") = i) ≠ 0 then
7:              messages ← nearestdistance
                .filter(nearestdistance("prediction") = i)
                .select("messages").first()
8:          end if
9:      end for
10:     function CalDistance(cluster, datapoint)
11:         return Vectors.sqdist(centroids(cluster), datapoint)
12:     end function
13: end procedure
```

4.4.2.4 Extraction of Representative Value

The Euclidean distance is obtained from each clustered error message and centroid of cluster, and the feature vector of the shortest error message is extracted as a representative value. The original error message is extracted corresponding to the feature vector of the extracted error message from Spark Dataframe.

As shown in Algorithm 2, the centroid of each cluster and the Euclidean distance between each message are calculated by the CalDistance method of step 3, and retained as a Dataframe string. In step 4, Spark SQL is used to determine the message with the shortest distance for each cluster, and the message body with the shortest distance for each cluster is acquired from Spark Dataframe in step 7.

In the machine learning use cases using the conventional k-means algorithm, it was common to interpret the data point with the longest distance from the centroid of cluster such as anomaly detection in network traffic as an outlier [17]. However, here we adopt a method of interpreting data with the shortest distance from the centroid of a cluster as a representative value. An image of acquiring representative value is shown in **Fig. 4**.

On the other hand, it is also possible that important messages need to be extracted are not extracted in this representative value extraction phase. Therefore, the message read by the monitoring system needs to be set the message after deduplication in Section 4.3 and the message before clustering in Section 4.4. By having the monitoring system read the message after deduplication and the messages before clustering, all kinds of messages can be finally notified to the administrator through the monitoring system instead of taking operation time.

4.4.3 EMail Notification

Inquire the error message extracted in Sections 4.4.1 and 4.4.2, call Amazon Simple Notification Service from *Log Marshaller* with the error message as an argument and notify it to the EMail address of the administrator. These messages are formatted in advance for recipient visibility in this message merge process.

4.5 Expected Latency

Estimated breakdown time throughout the workflow is shown in **Fig. 5**. *Log Collector* and *Log Processor* respectively takes only several seconds to process log messages in sequential and be influenced by computional and network overhead. *Log Deduplicator* is a streaming process that acquires data from Kafka as *Log Processor*. Therefore, *Log Deduplicator* is required to get stream data on *Log Processor* at certain intervals and is designed to invoke every 2 minutes with Spark's micro batch. *Log Marshaller* is a batch process that acquires data as files from Amazon S3. So, *Log Marshaller* is started every 2 minutes by Linux crontab scheduler. Since start interval time of *Log Deduplicator* and *Log Marshaller* depends on administrator's requirement, we set to 2 minutes as provisional minimum value for batch process. *Notification Service* takes only several seconds to process message strings in sequential and be influenced by computional and network overhead. The entire workflow is designed with a latency of just over 4 minutes from error occurrence to notification to the administrator.

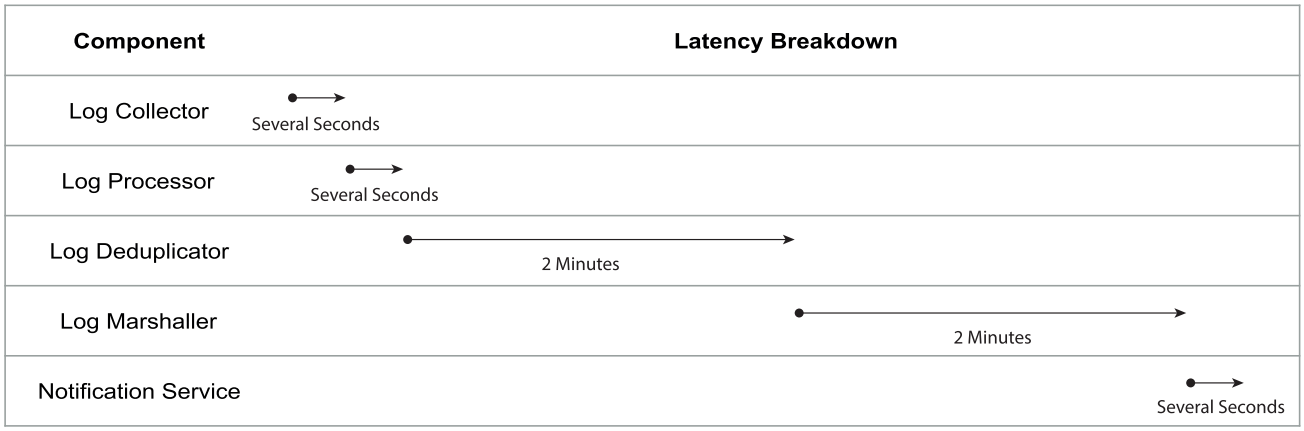

Fig. 5 Expected latency.

No.	Message Sequence	Classification
1	[yyyy-MM-dd HH:mm:ss.SSS ERROR ac.keio.sdm.ConcurrentLogControl.Streaming$ EXP-E000001 "Tweet Message is too long" jp.ac.keio.sdm.ConcurrentLogControl.NumericException: at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:30)]	Application Exception
2	[yyyy-MM-dd HH:mm:ss.SSS ERROR ac.keio.sdm.ConcurrentLogControl.Streaming$ EXP-E000002 "Tweet Message contains Japanese words" jp.ac.keio.sdm.ConcurrentLogControl.UnicodeBlockException: at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:40)]	Application Exception
3	[yyyy-MM-dd HH:mm:ss.SSS ERROR ac.keio.sdm.ConcurrentLogControl.Streaming$ EXP-E000003 "Tweet Message contains Hash Tag" jp.ac.keio.sdm.ConcurrentLogControl.WordLengthException: at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:50)]	Application Exception
4	[yyyy-MM-dd HH:mm:ss.SSS ERROR ac.keio.sdm.ConcurrentLogControl.Streaming$ EXP-E000004 "Tweet Message contains Numeric" jp.ac.keio.sdm.ConcurrentLogControl.NumericException: at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:60)]	Application Exception
5	[yyyy-MM-dd HH:mm:ss.SSS ERROR org.apache.spark.executor.Executor Exception in task *Any Message String* java.lang.StringIndexOutOfBoundsException: String index out of range: *Any Message String* at java.lang.String.charAt(String.java:646)]	Runtime Exception
6	[yyyy-MM-dd HH:mm:ss.SSS ERROR org.apache.spark.executor.Executor Exception in task *Any Message String* java.lang.NumberFormatException: For input string: "*Any Message String*" at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)]	Runtime Exception
7	[yyyy-MM-dd HH:mm:ss.SSS ERROR apache.spark.streaming.scheduler.JobScheduler Error running job streaming job *Any Message String*	Framework Error
8	[yyyy-MM-dd HH:mm:ss.SSS WARN org.apache.spark.scheduler.TaskSetManager Lost task *Any Message String*	Framework Error

Fig. 6 Expected extraction result.

5. Verification and Evaluation

5.1 Verification Method

5.1.1 Verification Data

For the evaluation, the Spark application which implemented the proposed methods was deployed to Amazon Elastic MapReduce. In addition, a Spark application *Error Generator* is created to implement logic to output large number of error files in a short time, and the following items are verified.

(1) Error message deduplication rate
(2) Accuracy of representative value extraction
(3) Latency from when an error occurs until it is notified to the administrator

"Error message deduplication rate" verifies effect to eliminate redudancy of large number of error messages in units of windows based on rules. "Accuracy of representative value extraction" tests accuracy of retrieving only the required messages based on machine learning method. "Latency from when an error occurs until it is notified to the administrator" verifies whether this tools output is notified to end user in short preriod. These items were designed from the administrator's perspective who isolate failures in a short period of time.

For *Error Generator*, Spark Streaming is used to receive the streaming data of tweets in Japanese from Twitter, and the following check is conducted for the tweet string, and an exception is thrown explicitly. The type of exception that occurs depends on the string of tweets, and there are also tweets that pass all the checks.

(1) Check exception based on the presence/absence of hash tag
(2) Check exception due to Japanese character code
(3) String length check exception
(4) Check exception due to presence/absence of number
(5) Runtime exception due to invalid index
(6) Runtime exception due to invalid data type conversion

Error messages generated by *Error Generator* is transferred to *Log Processor* in JSON format using FluentD as *Log Collector* that is installed at each node and temporarily retained in Kafka's messaging queue.

5.1.2 Verification Method

Log Deduplicator uses Spark Streaming to acquire error messages from *Log Processor* at 2-minute intervals, performs deduplication, and sends the results to Amazon S3 storage in Parquet format. By doing so, data before deduplication are also simultaneously output to Amazon S3 storage, and the deduplication rate is derived by comparing with the results after deduplication.

Log Marshaller extracts deduplicated results from Amazon S3

storage and extracts error messages. In addition to extracting three or more deduplicated error messages, error messages that were not subject to deduplication are clustered and only representative values are extracted. The extracted error message are notified via EMail to the administrator from *Log Marshaller* via Amazon SNS, and then the accuracy of representative value extraction is confirmed by matching the notification contents with the contents output from the *Log Deduplicator*.

The latency from the time when the errors occur to the time when the administrator is notified is derived by subtracting the time stamp of the file output to the local disk of *Error Generator* from the time stamp of the EMail sent to the administrator.

5.2 Expected Value

Figure 6 shows the expected extracted result. No.1 to No.4 Application Exceptions and No.5 and No.6 Runtime Exceptions are expected to be deduplicated or extracted as representative values.

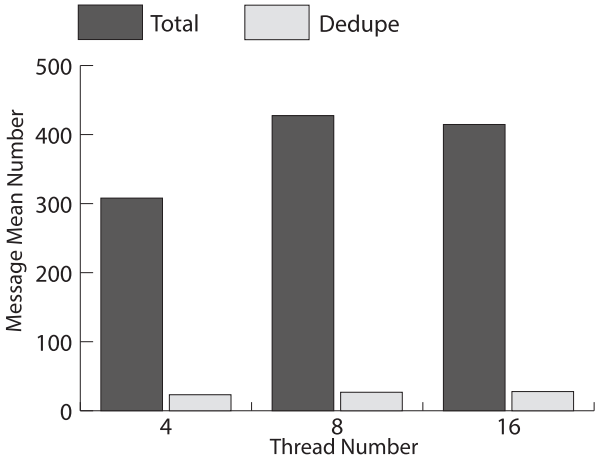

Fig. 7 Deduplication result.

In addition, No.7 and No.8 Application Framwork Errors when jobs fail are deduplicated or extracted as representative values. The number and type of error messages to be extracted depends on the nature of each tweet message string that happen to be acquired from Twitter used as input data. However, these messages output to the file once needs to be notified to be deduplicated or extracted as a representative value to administrator.

5.3 Verification Results

5.3.1 Deduplication Rate of Error Messages

In the experiment, we changed the number of threads of *Error Generator* to 4, 8, 16 and observed the results of deduplication. The results are shown in **Fig. 7**.

The graph is the average value of the result of executing *Log Deduplicator* five times for each thread number. Increasing the thread number from 4 to 8 will increase the number of output error messages by 38.8%, but if the thread number increases from 8 to 16, the “Total” number of output error messages remains unchanged. This is due to the CPU performance limit of the *Error Generator* node that generates the error. Meanwhile, expected results are shown for the deduplication rate “Dedupe,” which is 92.5% for the thread number 4, 93.7% for the thread number 8, and 93.3% for the thread number 16. These results indicate that the method adequately reduces load on the monitoring system that reads error messages.

5.3.2 Accuracy of representative value extraction

In the experiment, we changed the number of threads of *Error Generator* to 4, 8, 16 and observed the results of deduplication. The results are shown in **Fig. 8**.

698 error messages that are actually obtained as the output are organized into 10 messages and extracted. Category D (Duplication) denotes the data deduplicated in Section 4.4.1, and cate-

Message Sequence	Classification
[2019-12-08 13:30:07.195 ERROR ac.keio.sdm.ConcurrentLogControl.Streaming$ EXP-E000002 "Tweet Message contains Japanese words" jp.ac.keio.sdm.ConcurrentLogControl.UnicodeBlockException: at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:40)]	D
[2019-12-08 13:30:07.195 ERROR ac.keio.sdm.ConcurrentLogControl.Streaming$ EXP-E000003 "Tweet Message contains Hash Tag" jp.ac.keio.sdm.ConcurrentLogControl.WordLengthException: at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:50)]	D
[2019-12-08 13:30:07.195 ERROR ac.keio.sdm.ConcurrentLogControl.Streaming$ EXP-E000004 "Tweet Message contains Numeric" jp.ac.keio.sdm.ConcurrentLogControl.NumericException: at jp.ac.keio.sdm.ConcurrentLogControl.Streaming$$anonfun$4.apply(Streaming.scala:60)]	D
[2019-12-08 13:30:07.195 ERROR org.apache.spark.executor.Executor Exception in task 2.0 in stage 1.0 (TID 3) java.lang.StringIndexOutOfBoundsException: String index out of range: 10 at java.lang.String.charAt(String.java:646)]	D
[2019-12-08 13:30:07.195 WARN org.apache.spark.scheduler.TaskSetManager Lost task 2.0 in stage 1.0 (TID 3, localhost): java.lang.StringIndexOutOfBoundsException: String index out of range: 10 at java.lang.String.charAt(String.java:646) at scala.collection.immutable.StringOps$.apply$extension(StringOps.scala:37)]	D
[2019-12-08 13:30:07.195 WARN org.apache.spark.scheduler.TaskSetManager Lost task 0.0 in stage 1.0 (TID 1, localhost): java.lang.NumberFormatException: For input string: "@kapitanmoo" at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65) at java.lang.Integer.parseInt(Integer.java:580)]	D
[2019-12-08 13:30:07.195 ERROR org.apache.spark.executor.Executor Exception in task 2.0 in stage 1.0 (TID 3) java.lang.NumberFormatException: For input string: "川上さんがかわいい。かわいすぎて心臓が持たない。墓掘ってくる。" at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)]	R
[2019-12-08 13:30:07.195 ERROR apache.spark.streaming.scheduler.JobScheduler Error running job streaming job 1575811640000 ms.0 org.apache.spark.SparkException: Job aborted due to stage failure: Task 2 in stage 1.0 failed 1 times, most recent failure: Lost task 2.0 in stage 1.0 (TID 3, localhost): java.lang.StringIndexOutOfBoundsException: String index out of range: 10 at java.lang.String.charAt(String.java:646)]	R
[2019-12-08 13:30:07.195 WARN org.apache.spark.scheduler.TaskSetManager Lost task 4.0 in stage 2.0 (TID 12, localhost): java.lang.NumberFormatException: For input string: "3年前の今日焼酎水割りをぶっかけで水没したコントローラー 分解して乾燥させたけど、不動品になっちゃったけどまたなぜか動き出したんですよ" at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)]	R
[2019-12-08 13:30:07.195 WARN org.apache.spark.scheduler.TaskSetManager Lost task 1.0 in stage 2.0 (TID 8, localhost): java.lang.NumberFormatException: For input string: "指板のバーズアイメイプルしゅごい Fender" at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)]	R

Fig. 8 Extraction result.

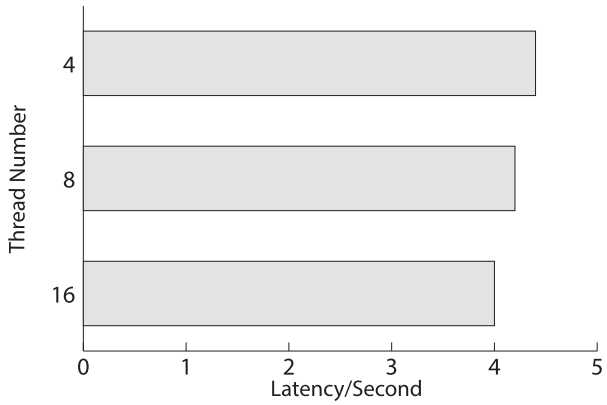

Fig. 9 Notification latency.

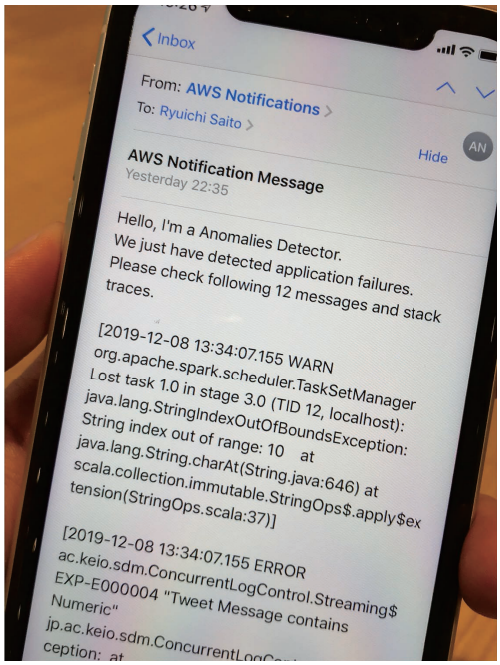

Fig. 10 Notification image.

gory R (Representative) denotes the data not deduplicated in Section 4.4.1 but extracted as representative values after clustering. Figure 8 presents the result of executing eight threads; however, the data that were output in three or more messages were simply deduplicated in the error message with extraction category D. Alternatively, category R error messages are not duplicated more than twice and nonstandard error messages in an application are ignored as noise. Instead, they are error messages explicitly generated in large quantities and extracted as representative values.

Comparing Fig. 8 with Fig. 6, it can be seen that seven of the eight messages in Fig. 6 are extracted. The message No.1 in Fig. 6 was not actually output to the file since an error related to the string length did not happen. Owing to these reasons, it can be judged as a valid result.

5.3.3 Notification Time (Latency)

The error generated by executing *Error Generator* in Tokyo node is sent to American Oregon Region of Amazon Web Services, deduplication/abstraction processes are executed, and the results are received by EMail in a smartphone in Japan. **Figure 9** shows the latency from the occurrence of an error to the receipt of the summary of errors by EMail.

The graph shows the average values of the results of executing in each of the thread numbers for five times. The results show that the EMail notification was sent 4.0 to 4.4 min after the error occurrence. In the experiment, the average latency of 4.0 to 4.4 minutes is a reasonable result, because *Log Deduplicator* and *Log Marshaller* are executed in 2-minutes cycles as seen in Fig. 5. Latency becomes lower as the thread number increases from 4 to 8 and from 8 to 16, but it depends on the periodic execution timing of the Log Deduplicator and Log Marshaller jobs, and the correlation between the thread number and latency cannot be perceived. Moreover, the image of the actual notification results is shown in **Fig. 10**. The details of error message are displayed following the guidelines. It is also possible to send notification messages via Short Message Service (SMS) in a manner similar to EMail messages.

6. Conclusion and Upcoming Challenges

In this research, we proposed and verified methods to collect streaming error messages in the in-memory cluster computing environment and perform deduplication after labeling as a preprocess. Clustering is performed for error messages that could not be deduplicated after converting a message string to a feature, extracting a representative value from that, and notifying the summarized content to the administrator in a certain amount of time. The following knowledge was obtained through verification.

(1) As long as the log format is predetermined for error messages that are output in large quantities in an in-memory cluster computing environment, over 90% of the messages can be summarized and managed through the deduplication process.
(2) Representative value can be estimated by extracting error messages in the shortest distance from the central axis in each error message plotted by k-means clustering for managing unexpected types and numbers of error messages.
(3) Streaming process using in-memory cluster computing is effective in system design for managing error messages in in-memory cluster computing environment.

The challenges in the future can be examined by dividing them into clustering algorithms and verification data. Regarding clustering algorithms, this research uses the k-means algorithm of unsupervised learning of the Spark ML library; however, it is presumed that the accuracy in the extraction of representative value can be improved by using supervised learning, creating models based on training data in advance, and deploying the created data on Spark cluster. It is also possible to improve the clustering accuracy by reducing the dimensionality of the feature values extracted by TF-IDF.

Regarding the verification data, the exception that was generated continuously in a pseudo manner were logged and used in this research; however, error messages in an in-memory cluster computing environment operated by actual companies and research institutions are more diverse. Although it can be considered that the effect of deduplication processing does not change significantly if the error message is output in the format decided at the time of logging beforehand, improvement is required by tun-

ing the clustering algorithm in the production environment as it depends on the type of the error message and size of the k-means cluster during production.

Acknowledgments This research was conducted with aid from the Keio University Doctrate Student Grant-in-Aid Program in 2017 and 2018.

References

[1] Dean, J. and Chemawat, S.: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI (2004).
[2] Coulouris, G., Dollimore, J., Kindberg, T. and Blair, G.: *Distributed Systems: Concepts and Design*, 5th Edition, Pearson (2012).
[3] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I.: *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, Research Council of Canada (2011).
[4] Iyer, R.K., Young, L.T. and Iyer, P.V.K.: *Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data*, IEEE (1990).
[5] Zheng, Z., Lan, Z., Park, B.H. and Geist, A.: *System Log Preprocessing to Improve Failure Prediction*, pp.572–577, IEEE (2009).
[6] Gurumdimma, N., Jhumka, A., Liakata, M., Chuah, E. and Browne, J.: Towards Detecting Patterns in Failure Logs of Large-Scale Distributed Systems, *IEEE International Parallel and Distributed Processing Symposium Workshops*, pp.1052–1061 (2015).
[7] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S. and Ryaboy, D.: Storm @Twitter, *SIGMOD'14*, pp.147–156 (2014).
[8] Xin, R.S., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S. and Stoica, I.: Shark: SQL and Rich Analytics at Scale, *SIGMOD'13*, pp.13–24 (2013).
[9] Armbrust, M., Das, T., Torres, J., Yavuz, B., Zhu, S., Xin, R., Ghodsi, A., Stoica, I. and Zaharia, M.: Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark, *SIGMOD'18*, pp.601–613 (2018).
[10] Erra, U., Senatore, S., Minnella, F. and Caggianese, G.: Approximate TF-IDF based on topic extraction from massive message stream using the GPU, *Information Sciences*, Vol.292, pp.143–161 (2015).
[11] Cormode, G. and Hadjieleftheriou, M.: Finding the frequent items in streams of data, *Comm. ACM*, Vol.52, pp.97–105 (2009).
[12] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M.J., Zadeh, R., Zaharia, M. and Talwalkary, A.: MLlib: Machine Learning in Apache Spark, *Journal of Machine Learning Research*, Vol.17, pp.1–7 (2016).
[13] Svyatkovskiy, A., Imai, K., Kroeger, M. and Shiraito, Y.: Large-scale text processing pipeline with Apache Spark, *IEEE International Conference on Big Data*, pp.3928–3935 (2016).
[14] Cloud Native Computing Foundation: A WEB Page, FluentD (online), available from ⟨https://www.fluentd.org⟩ (accessed 2019-04-28).
[15] Kreps, J., Narkhede, N. and Rao, J.: Kafka: A Distributed Messaging System for Log Processing, *NetDB'11*, Athens, Greece (2011).
[16] Bahmani, B., Moseley, B., Vattani, A., Kumar, R. and Vassilvitskii, S.: Scalable KMeans++, *Proc. VLDB Endowment*, Vol.5, No.7 (2012).
[17] Ryza, S., Laserson, U., Owen, S. and Wills, J.: *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*, O'Reilly Media (2015).

Ryuichi Saito was born in 1974. He is a Ph.D. student at Graduate School of System Design and Management, Keio University. He was engaged in software architecture design until 2018 and is currently engaged in Business Development with technology domain companies at SoftBank Corporation. He is a member of the IPSJ. His research interest is Data Analytics Platforms Design and Systems Engineering.

Shinichiro Haruyama obtained a Master's degree in Electrical Engineering and Computer Science at University of California, Berkeley, USA and a Ph.D. in Computer Science at University of Texas, Austin, USA. He worked for AT&T Bell Laboratories (1991–) and Lucent Technology Bell Laboratories (1996–) as researcher in USA. After that, he moved to Sony Computer Science Laboratories as researcher (1998–2002), served as visiting professor at Department of Information and Computer Science, Faculty of Science and Technology, Keio University (2002–2008) in Japan. Currently, he served as professor at Graduate School of System Design and Management, Keio University (2008–).

(Editor in Charge: *Saneyasu Yamaguchi*)