Log Clustering based Problem Identification for Online Service Systems

Qingwei Lin[†], Hongyu Zhang[†], Jian-Guang Lou[†], Yu Zhang[§], Xuewei Chen[†]
[†]Microsoft Research, Beijing 100080, China
[§]Microsoft Corporation, Redmond, WA, USA
{qlin, honzhang, jlou, yugzhang, v-xuewc}@microsoft.com

ABSTRACT
Logs play an important role in the maintenance of large-scale online service systems. When an online service fails, engineers need to examine recorded logs to gain insights into the failure and identify the potential problems. Traditionally, engineers perform simple keyword search (such as "error" and "exception") of logs that may be associated with the failures. Such an approach is often time consuming and error prone. Through our collaboration with Microsoft service product teams, we propose LogCluster, an approach that clusters the logs to ease log-based problem identification. LogCluster also utilizes a knowledge base to check if the log sequences occurred before. Engineers only need to examine a small number of previously unseen, representative log sequences extracted from the clusters to identify a problem, thus significantly reducing the number of logs that should be examined, meanwhile improving the identification accuracy. Through experiments on two Hadoop-based applications and two large-scale Microsoft online service systems, we show that our approach is effective and outperforms the state-of-the-art work proposed by Shang et al. in ICSE 2013. We have successfully applied LogCluster to the maintenance of many actual Microsoft online service systems. In this paper, we also share our success stories and lessons learned.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging - *monitors, tracing*.

General Terms
Measurement, Reliability

Keywords
Logs, Problem Identification, Log Clustering, Diagnosis, Online Service System.

1. INTRODUCTION
Large-scale online service systems, such as those of Microsoft, Google, and Amazon, are getting increasingly large and complex - they often contain hundreds of distributed components and support a large number of concurrent users. Typically, engineers first test an online service system in a lab (testing) environment and then deploy the system in production (actual) environment. The lab environment often has a small, pseudo cloud setting with a limited amount of data, while the production environment has a large and complex cloud infrastructure supporting a huge amount of data. Because of the differences between the lab and production environments, online service systems often encounter unexpected problems, even they are well tested in the lab environment.

As debugging tools (e.g., an IDE debugger), all too often, are inapplicable in production settings, logging has become a principal way to record the key runtime information (e.g., states, events) of the online service systems into console logs for postmortem analysis. As an example, a fragment of logs produced by a Microsoft online service system is as follows:

```
2:26:00 PM   Connecting to SQL DB AM-DB-3202
2:26:00 PM   System.Data.SqlClient.SqlException: A network-related error occurred while establishing a connection to SQL Server
2:26:00 PM   Connecting to failover SQL DB #1 AM-DB-3203
2:26:00 PM   Load user profile successfully, start to get user document No. 33627349082
```

In general, when an online service fails, engineers need to examine the recorded logs to gain insight into the failure, identify the problems, and perform troubleshooting. Traditionally, engineers perform simple keyword search to obtain logs that indicate runtime failures. Examples of keywords include "fail", "kill", etc. However, such an approach is time-consuming and ineffective in production environment, especially for large-scale online service systems. Through our collaboration with Microsoft service product teams over the past four years, we have identified the following characteristics of the logs of online service systems:

- First, every day a vast number of logs are generated by the online service systems. As the service systems become increasingly complex, more and more logs are generated. For some large-scale systems that provide global services, the amount of daily log data could reach tens of TBs. A Microsoft service system even generates over 1PB of logs every day. As such, once a problem occurs, it is very time consuming to diagnose it through manual examination of the logs.
- Second, modern online systems often incorporate the "faileover" mechanism [5], which dynamically allocates jobs among computing nodes considering factors such as availability and performance. The systems could proactively kill a job and restart it elsewhere, which causes many "kill" and "fail" keywords in logs. Therefore, simple keyword search will lead to a large number of false positives, and hinder the identification of real problems.
- Third, there is a large number of recurrent issues reflected by the logs. For a traditional software system, if a bug is detected, it will be fixed and in most cases it will not appear in the new release. However, in a large-scale online service system, there

ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4205-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2889160.2889232

are many recurrent issues, which could lead to a lot of redundant effort in examining logs and diagnosing the previously known problems. The recurrent issues occur due to the following three reasons: a) When a service fails, a common practice is to restore the service as soon as possible by identifying a temporary workaround solution (such as restarting a server). Therefore, before the root cause is fixed, recurrent issues are expected; b) A large-scale online service usually contains a large number of components running in different computing environments. An issue occurs in one environment may appear in other environments; c) Many service failures are caused by environmental issues (such as machine down and network disconnection), which could occur from time to time.

- Fourth, log messages are highly diverse. Because of the complexity of an online service, the execution paths that lead to a same type of failure could be different. The frequent changes to service features and environments also increase the diversity of log messages. Furthermore, not all log messages are equal in their importance for problem identification - some log messages appear in both normal and failure scenarios, while some log messages only appear in failed scenarios and are more likely to be related to the failures. It is thus challenging for engineers to effectively identify and differentiate various service problems through examining a large number of highly diverse logs.

In recent years, some tools have been developed to help engineers identify a service problem through automated analysis of a large number of logs. For example, Shang et al. [18] proposed to group the related log messages into execution sequences. Engineers can compare the log sequences generated in production environment and the log sequences generated in lab environment. Their method not only significantly reduces the number of logs that should be verified, but also achieves much higher precision for identifying deployment problems than the traditional keyword search approach. However, the precision of their method is still rather low and could be further improved. Furthermore, their method does not consider the previous known problems, therefore may incur redundant effort in examining logs of recurrent issues. We will discuss more about the limitations of the ICSE'13 approach in Section 2.

In this paper, we propose LogCluster, a log clustering based problem identification approach that considers all the characteristics of the logs of online service systems. In our work, we assign weights to log messages and group the similar log sequences into clusters. We then extract a representative log sequence from each cluster. The operation of LogCluster can be divided into two phases: construction phase and production phase. In the construction phase, we use the log sequences collected from the testing environment, cluster them to construct an initial knowledge base. In the production phase, we analyze the log sequences collected from the actual production environment, cluster them and check if the clusters can be found in the knowledge base. In this way, developer only need to examine a small number of representative log sequences from the clusters that are previously unseen. Therefore, LogCluster further reduces the total number of logs need to be manually examined and improves the effectiveness of problem identification.

We have evaluated our method on two Hadoop applications and two Microsoft online service systems. The results show that the proposed method can effectively help engineers identify problems of online service systems. Our results also show that the proposed method outperforms the state-of-the-art method proposed by Shang et al. [18]. We have also successfully applied LogCluster to the maintenance of many actual Microsoft online service systems.

The main contributions of this paper are as follows:

- We propose LogCluster, which facilitates problem identification by clustering similar log sequences and retrieving recurrent issues. Our approach outperforms the state-of-the-art method [18].
- We have successfully applied LogCluster to many Microsoft online service systems and confirmed the effectiveness of our approach.
- We share some success stories and lessons learned from the collaborations with multiple product teams across Microsoft over the past four years.

The remainder of this paper is organized as follows: We introduce background and motivation of our work in Section 2. Section 3 describes our approach. Section 4 presents our experimental design and results. In Section 5, we present the successful stories of our approach in industrial practice. In Section 6, we discuss our lessons learned. Section 7 surveys related work followed by Section 8 that concludes this paper.

2. BACKGROUND AND MOTIVATION

2.1 Logs and Log Parsing

Large-scale software systems often generate logs for troubleshooting. The log messages are usually semi-structured text strings, which are used to record events or states of interest. In general, when a job fails, engineers can examine recorded log files to gain insight about the failure, and locate the potential root causes. Logging is particularly important for large-scale online services running in a Big Data environment with multiple clusters of servers and data centers, where other software debugging techniques are difficult to be applied. For example, it is impractical to attach a debugger to an online service system.

Because of its importance, logging has been commonly used in practice. For example, Hadoop [9] prints job and task related logs to provide information about the inner working status of the platform. An empirical study also shows that logging is commonly used in Microsoft [4, 7], where engineers use several mechanisms such as ULS [22] to perform logging.

Figure 1 shows an example of logs generated by a task of a Hadoop service.

```
2015-09-29 10:38:40 INFO [ContainerLauncher #6]
org.apache.hadoop.mapreduce.v2.app.launcher.ContainerLauncherImp
l: Processing the event EventType:
CONTAINER_REMOTE_CLEANUP for container container_000006
taskAttempt attempt_m_000004_0

2015-09-29 10:38:43 INFO [ContainerLauncher #5]
org.apache.hadoop.mapreduce.v2.app.launcher.ContainerLauncherImp
l: Processing the event EventType:
CONTAINER_REMOTE_CLEANUP for container container_000008
taskAttempt attempt_m_000006_0

2015-09-29 10:38:43 INFO [ContainerLauncher #5]
org.apache.hadoop.mapreduce.v2.app.launcher.ContainerLauncherImp
l: KILLING attempt_m_000006_0

2015-09-29 10:38:43 INFO [AsyncDispatcher event handler]
org.apache.hadoop.mapreduce.v2.app.job.impl.TaskAttemptImpl:
attempt_m_000006_0 TaskAttempt Transitioned from
SUCCESS_CONTAINER_CLEANUP to SUCCEEDED
```

```
2015-09-29 10:38:43 INFO [AsyncDispatcher event handler] org.apache.hadoop.mapreduce.v2.app.job.impl.TaskImpl: Task succeeded with attempt attempt_m_000006_0

2015-09-29 10:38:43 INFO [AsyncDispatcher event handler] org.apache.hadoop.mapreduce.v2.app.job.impl.TaskImpl: task_m_000006 Task Transitioned from RUNNING to SUCCEEDED

2015-09-29 10:38:44 INFO [AsyncDispatcher event handler] org.apache.hadoop.mapreduce.v2.app.job.impl.TaskAttemptImpl: Diagnostics report from attempt_m_000006_0: Container killed by the ApplicationMaster.
```

Figure 1. An example of Hadoop logs

A semi-structured log message contains two types of information: a free-form constant string that is used to describe a system status; and parameters that record some important system attributes. To facilitate analysis, a common practice is to parse the log messages into constant strings and parameters [6, 11, 12, 24] and form abstract log messages. An abstract log message is often called a *log event*, which represents generic log messages printed by the same log-print statement in the source code. The log events can be linked through the same task ID and form a log sequence. Figure 2 gives an example of log sequence (E1, E1, E2, E3, E4, E5, E6) obtained through parsing the log messages shown in Figure 1.

```
E1: $DATE INFO [ContainerLauncher #$NUMBER] org.apache.hadoop.mapreduce.v2.app.launcher.ContainerLauncherImpl: Processing the event EventType: CONTAINER_REMOTE_CLEANUP for container $CONTAINERID taskAttempt $ATTEMPTID

E1: $DATE INFO [ContainerLauncher #$NUMBER] org.apache.hadoop.mapreduce.v2.app.launcher.ContainerLauncherImpl: Processing the event EventType: CONTAINER_REMOTE_CLEANUP for container $CONTAINERID taskAttempt $ATTEMPTID

E2: $DATE INFO [ContainerLauncher #$NUMBER] org.apache.hadoop.mapreduce.v2.app.launcher.ContainerLauncherImpl: KILLING $ATTEMPTID

E3: $DATE INFO [AsyncDispatcher event handler] org.apache.hadoop.mapreduce.v2.app.job.impl.TaskAttemptImpl: $ATTEMPTID TaskAttempt Transitioned from SUCCESS_CONTAINER_CLEANUP to SUCCEEDED

E4: $DATE INFO [AsyncDispatcher event handler] org.apache.hadoop.mapreduce.v2.app.job.impl.TaskImpl: Task succeeded with attempt $ATTEMPTID

E5: $DATE INFO [AsyncDispatcher event handler] org.apache.hadoop.mapreduce.v2.app.job.impl.TaskImpl: $TASKID Task Transitioned from RUNNING to SUCCEEDED

E6: $DATE INFO [AsyncDispatcher event handler] org.apache.hadoop.mapreduce.v2.app.job.impl.TaskAttemptImpl: Diagnostics report from $ATTEMPTID: Container killed by the ApplicationMaster.
```

Figure 2. An example of log parsing

2.2 Log-based Problem Identification

Although important, log-based problem identification is not easy. Traditionally, when a service failure occurs, engineers identify problems by searching for "erroneous" jobs in the generated logs. They perform simple keyword search (such as "kill", "fail", "error", and "exception") of logs that may be associated with the failure. Due to the increasing scale and complexity of online service systems, the number of generated logs could be quickly overwhelming. Clearly, it can be very time consuming for a human operator to diagnose system problems by manually examining a huge number of log messages.

Furthermore, modern online systems often incorporate the "failover" mechanism [5]. To ensure reliability, availability, and performance, the systems could dynamically allocate jobs among computing nodes by proactively killing a job and restart it elsewhere. Therefore there are many "kill" and "fail" keywords in logs. For example, the red lines in Figure 1 are log messages that contain the keyword "kill". However, these lines actually indicate normal system behavior. Therefore, simple keyword search will lead to a large number of false positives, and hinder the identification of real problems. Because of the inefficiency and ineffectiveness of the traditional approach, it is essential to have automated tools that can assist log-based problem identification.

To help engineers perform log-based problem identification, Shang et al. [18] proposed to examine the differences between the log sequences in testing (lab) environment and the log sequences in production (actual) environment. Their approach first abstracts the execution logs, recovers the execution sequences, and then compares the sequences between the testing and actual deployments. Ideally, these two sets should be identical. However, due to platform configurations and workload differences, the underlying platform may execute the applications differently. The delta sets of execution sequences between these two sets could reflect the potential deployment failures. Their experiments on three Hadoop applications show that their approach not only significantly reduces the number of logs (by 86% - 97%) that should be verified, but also improves effectiveness in identifying deployment failures when compared to the traditional keyword search approach.

Although effective, the ICSE'13 approach has limitations too:

1) Although it can reduce effort in manual examination of log sequences, its precision is still rather low. According to their experiments, the precision values range from 10% to 38%, which clearly could be improved. Our analysis finds that the ICSE'13 approach simplifies the log sequences by only removing repetitions and permutations of the sequences. For example, both of the following two sequences: "E1, E2, E3, E3, E5, E6" and "E1, E3, E5, E2, E6" are reduced to the sequence "E1, E2, E3, E5, E6". In this way, similar logs are grouped together and manual examination effort can be reduced. Such a kind of grouping is rather simple as it does not consider the potential similarity between two log sequences when they are not repetition or permutation of each other. As described in Section 1, our experience with real logs of online service systems show that log sequences are highly diverse and log events are not equal in importance. We believe that the precision of the ICSE'13 approach could be further improved by incorporating a more advanced clustering technique.

2) It does not utilize the previous known failures. Currently, the ICSE'13 approach requires the engineers to examine all delta sequences that contain the failure-indicating keywords. As described in Section 1, our experience with real logs of online service systems shows that many of the failures are recurrent ones, whose mitigations/resolutions are already known to the engineers and whose corresponding logs need not to be examined again. Therefore, we could utilize the previous known failures to further reduce the number of log sequences that should be manually examined.

In this paper, we describe our proposed approach, which utilizes the characteristics of logs of online service systems to facilitate log-based problem identification. Our approach also addresses the limitations of the previous approaches (the keyword search approach and the ICSE'13 approach) and outperforms them.

3. THE PROPOSED APPROACH

3.1 Overview

The overall structure of LogCluster is shown in Figure 3. The operations of LogCluster can be divided into two phases: construction phase and production phase. In the construction phase, we use the log sequences collected from the testing environment. We convert the log sequences into vectors and cluster them. We then select a representative sequence from each cluster and store the selected sequences and the associated mitigation solutions in a knowledge base. In the production phase, we analyze the log sequences collected from the actual production environment. After log vectorization and clustering (which are the same as those in the construction phase), we extract a representative sequence from each cluster and check if it represents a previously examined cluster stored in the knowledge base. The engineers are only required to manually examine the representative log sequences that are previously unseen. The knowledge base is also updated with the new clusters. In this way, LogCluster can further reduce the effort required for log-based problem identification.

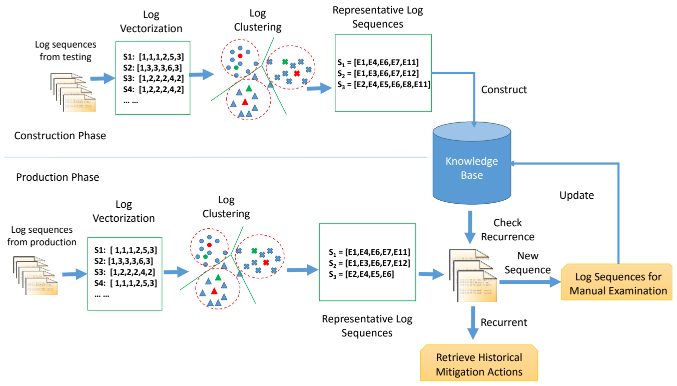

Figure 3. The overall structure of LogCluster

The major steps shown in Figure 3 are as follows. We describe them in detail in this section:

Log Vectorization: Before clustering log sequences, we first turn each log sequence into a vector. We believe that different log events have different importance in terms of problem identification. Therefore, we assign a weight to each event in a vector. More details are given in Section 3.2.

Log Clustering: We calculate the similarity value between two log sequences and apply the Agglomerative Hierarchical clustering technique to group the similar log sequences into clusters. More details are given in Section 3.3.

Extracting Representative Log Sequence: After log clustering, we get a number of clusters. We then select a representative log sequence from each cluster by choosing the centroid of the cluster. More details are given in Section 3.4.

Checking Recurrence: LogCluster checks if a representative log sequence appears before by querying a knowledge base. Only new log sequences are required to be manually examined. More details are given in Section 3.5.

3.2 Log Vectorization

For the free-form raw log messages, LogCluster first parses them into log events using the log abstraction technique described in [6]. It then produces log sequences by removing the duplicate events and linking the events with the same task ID. A log sequence contains multiple unique events. We believe that different log events have different discriminative power for problem identification. We propose two methods to weight the log events.

IDF-based Event Weighting: For each log event, we calculate its IDF (Inverse Document Frequency) values. IDF is commonly used as a term weighting technique in information retrieval [15]. We treat each event as a term and each log sequence as a document. Intuitively, if an event frequently appears in many log sequences, its discriminative power is lower than the event that only appears in a small number of log sequence. Formally, IDF is calculated as:

$$w_{idf}(t) = \log\left(\frac{N}{n_t}\right)$$

where N is the total number of log sequences, and n_t denotes the number of sequences where the event t appears. Using IDF, the events that occur very frequently have lower weights and the events that occur rarely have higher weights.

Contrast-based Event Weighting: When a new failure occurs, engineers often perform diagnosis by comparing the log sequences generated in production (actual) environment and the log sequences generated in lab (testing) environment. Intuitively, an event that appears in both lab and production environments has less discriminative power for problem identification than an event that only occurs in production environment. The events occur only in the production environment are more likely to reflect the failures, thus they can be weighted higher. In our approach, an event t is assigned with a weight of 1 if it only appears in production environment, otherwise, its weight is 0.

$$w_{con}(t) = \begin{cases} 1 & \text{if } t \text{ appears in } \Delta S \\ 0 & \text{otherwise} \end{cases}$$

where ΔS indicates the set of log events that only appear in production environment.

Vectorization: For a log event t, we combine its normalized IDF-based weight and contrast-based weight together as follows:

$$w(t) = 0.5 \times Norm(w_{idf}(t)) + 0.5 \times w_{con}(t)$$

where *Norm* is the normalization function, which normalizes the IDF-based weight to a value between 0 and 1. We choose the commonly used *Sigmoid* function [30] as the normalization function.

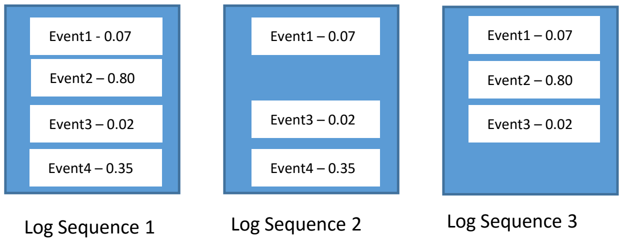

Figure 4. Log sequences represented as vectors

After calculating the weight for each event, we can represent a log sequence as a vector of weight in an N-dimensional space, where N is the number of unique events. For example, suppose there are 4 different events appearing in the three log sequences as shown in Figure 4. The log sequences can be represented as three 4-dimensional vectors: [0.07, 0.80, 0.02, 0.35], [0.07, 0.0, 0.02, 0.35], [0.07, 0.80, 0.02, 0.0].

3.3 Log Clustering

Having obtained the vector-representation of log sequences, we compute the cosine similarity between any two N-dimensional vectors S_i and S_j as follows:

$$Similarity(S_i, S_j) = \frac{S_i \cdot S_j}{\parallel S_i \parallel \parallel S_j \parallel} = \frac{\sum_{k=1}^{n} S_iE_k \times S_jE_k}{\sqrt{\sum_{k=1}^{n}(S_iE_k)^2} \times \sqrt{\sum_{k=1}^{n}(S_jE_k)^2}}$$

where S_iE_k stands for the k^{th} event in the j^{th} sequence vector.

Having computed the similarity between two log sequences, we perform log clustering using the Agglomerative Hierarchical clustering technique [8]. At the beginning of the agglomerative hierarchical clustering, each log sequence belongs to its own cluster. Then, the closest pair of clusters is selected and merged. To decide which pair of clusters should be merged, the distance metric between the clusters should be defined. In our approach, we adopt the maximum distance of all element pairs between two clusters as the cluster distance metric. In other words, the cluster distance metric depends on the maximum distance between the log sequences in each cluster. We adopt a distance threshold θ as a stopping criterion for the clustering process. The value of θ is set empirically (in our experiments, we set it to 0.5. We will discuss the impact of θ on the effectiveness of LogCluster in Section 4). Once the maximum distance between a pair of clusters is above the distance threshold, the clustering process for this pair is stopped. For example, in Figure 5, two clusters (Cluster 1 and Cluster 2) are produced. In this way, similar log sequences are grouped into the same cluster.

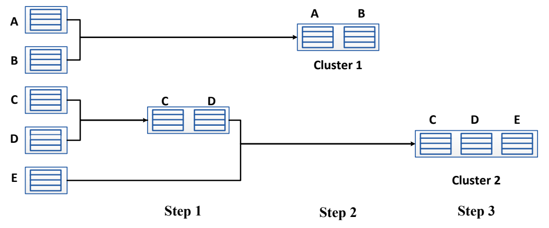

Figure 5. Illustration of the Agglomerative Hierarchical Clustering process

3.4 Extracting Representative Log Sequence

After log clustering, we get a number of clusters. For each cluster, we select a representative log sequence by choosing the centroid of the cluster. To do so, we compute the score of each log sequence *i* in a cluster based on its average distance to other log sequences in the same cluster:

$$Score(i) = \frac{1}{n-1}\sum_{j=1}^{n}(1 - Similarity(S_i, S_j))$$

where *n* is the number of log sequences in the cluster. From each cluster, we select the log sequence with the minimal score as the representative sequence of the cluster. The selected sequences are the candidates for manual examination.

3.5 Checking Recurrence

In practice, many failures that occur to an online service system are recurrent failures, which appeared in the past. For the recurrent failures, their mitigations/resolutions are already known to the engineers. Therefore, the corresponding log clusters need not to be examined again. LogCluster checks if a log cluster is a recurrent one by querying a knowledge base, which stores the historical log clusters (represented by the representative log sequences) obtained from the past executions. To check the recurrence of a cluster, the representative log sequence of the cluster is selected and the same cosine similarity measure in Section 3.3 is used to determine the similarity between the new cluster and all the clusters stored in the knowledge base (the distance threshold is also set to θ). If a cluster is deemed recurrent and corresponds to a known failure, the associated mitigation actions are retrieved from the knowledge base and returned to engineers. Only new representative log sequences (no matching sequence in knowledge base is found) are returned to engineers for manual examination. The knowledge base is also updated with the new sequences. In this way, LogCluster reduces the number of log sequences to be examined.

4. EXPERIMENTS

In this section, we describe our experiments for evaluating LogCluster.

4.1 Setup

In our experiments, we use the same two Hadoop-based Big Data applications that were used in [18]:

- WordCount: an application that is released with Hadoop as an example of MapReduce programming. The WordCount application analyzes the input files and counts the number of occurrences of each word in the input files.
- PageRank: a program that is used by a search engine for ranking Web pages.

During the execution of these two applications, the underlying Hadoop platform generate logs. We first run the applications in the lab environment without injecting any failures. In order to simulate the service failures in production environment, we manually inject the following deployment failures:

- Machine Down: we turn off one server when the applications are running to simulate the machine failure.
- Network Disconnection: we disconnect one server from the network to simulate the network connection failure.
- Disk Full: we manually fill up one server's hard disk when the applications are running to simulate the disk full failure.

In addition, to further evaluate LogClutser on industrial systems, we chose two Microsoft online service systems (their actual names are anonymized due to confidentiality):

- Microsoft Service X, which is a large-scale online service system serving millions of users globally. Designed with a 3-tier architecture, Service X runs on a large number of machines, each of which continuously generates a huge number of logs. Service X works in a load balance mode, it accepts end user requests, and dispatches them to different front ends according to load balancing strategies. There are many components in application tier, each in charge of a dedicated functionality. Most components include a failover mechanism in order to tolerate failures. Most of the user requests involve multiple components on multiple servers. Each component records its own logs and all the logs are automatically uploaded to a distributed HDFS-like data storage.
- Microsoft Service Y, which is also a large-scale online service system providing 7*24 basis, continuous service for tens of millions of end users worldwide. There is a SLA (service level agreement) for the service, such as 99.99% availability for the entire user purchased period. If SLA is violated, penalty could apply. The entire service is divided into multiple subsystems, each subsystem has its full set of front end, application tier, and database tier. Service Y produces a huge number of logs during runtime.

For the two Microsoft service systems, we obtain the logs from the lab (testing) environment. We also collect half an hour of real logs of each system from the production (actual) environment, including

3.3 million raw log messages from Service X and 10 million raw log messages from Service Y. We then use these logs to evaluate LogCluster.

All experiments were performed on a cluster of PC (a cluster with 46 cores across five machines). Each PC has Intel(R) Core(TM) i7-3770 CPU and 16GB RAM.

4.2 Research Questions

To evaluate our approach, we design experiments to address the following research questions:

RQ1: ***How much effort reduction does LogCluster achieve?***
When a service failure occurs, engineers need to examine the generated logs for troubleshooting. The number of logs to be examined is thus an important indicator of the effort required for engineers. To evaluate LogCluster in terms of effort reduction, we use the log data generated by running the two Hadoop applications (WordCount and PageRank) and count the number of log sequences that are required to be manually examined. We also evaluate LogCluster using the real log data generated by the two Microsoft online service systems.

To evaluate the effort required by traditional keyword search based approach, we count the number of log messages that contain the keywords (including "kill", "fail", "error", and "exception"). We also compare LogCluster with the state-of-the-art approach proposed in ICSE'13 [18].

RQ2: ***How accurate is LogCluster in identifying problems?***
Besides effort reduction, we also measure the accuracy of our approach in problem identification. We count the number of true positives (the number of examined log sequences that are indeed associated with actual failures) and false positives (the number of examined log sequences that are not associated with actual failures), and then calculate the precision values achieved by LogCluster, the ICSE'13 approach, and the traditional approach, respectively.

RQ3: ***The impact of the distance threshold***
As described in Section 3.3, LogCluster uses a hierarchical clustering algorithm to cluster the log sequences. Here a distance threshold θ is used to determine the number of clusters – a lower threshold leads to a larger number of clusters, which will in turn affect the total number of log sequences to be examined. Furthermore, the distance threshold θ is also used in Section 3.5 to retrieve recurrent sequences. In this RQ, we evaluate the impact of the distance threshold on the accuracy of LogCluster.

4.3 EXPERIMENTAL DESIGN AND RESULTS

This section presents our experimental design and results by addressing the research questions.

RQ1: ***How much effort reduction does LogCluster achieve?***
For the Hadoop applications (WordCount and PageRank), we first inject three consecutive machine failures and count the number of log sequences need to be examined at each round. Table 1 shows the effort reduction achieved by LogCluster. We can see that using LogCluster, the number of log sequences that should be examined is significantly reduced. For example, when the first machine failure is injected when running WordCount, only 8 of them require manual examination. While using the ICSE'13 approach, we need to examine 29 log sequences. Using traditional keyword search, we need to examine 335 raw log messages. When the second machine failure is injected, using LogCluster we only need to examine 5 log sequences, while using the ICSE'13 approach and the traditional keyword search, we need to examine 40 log sequences and 818 raw log messages, respectively. When the third machine failure is injected, using LogCluster we only need to examine 3 log sequences, while using the ICSE'13 approach and the traditional keyword search, we need to examine 25 log sequences and 392 raw log messages, respectively. The results show that using LogCluster, the number of log sequences to be examined decreases when there are recurrent failures. These results confirm the effectiveness of the proposed approach as described in Section 3. The same results are observed when running the PageRank application (for the 3rd failure, LogCluster detects that the log sequences appear before by querying the knowledge base. Therefore, no manual examination is needed).

We also evaluate LogCluster when there are multiple types of failures. For the Hadoop applications (WordCount and PageRank), we inject three failures of different types (Machine Down, Network Disconnection, and Disk Full) during runtime, and observe how LogCluster performs. Table 2 shows the results. When there are multiple failures, LogCluster still significantly reduces the number of log sequences that should be examined. For example, for WordCount, when the Network Disconnection failure is injected, using LogCluster we only need to examine 6 log sequences. While using the ICSE'13 approach and the traditional keyword search, we need to examine 20 log sequences and 8437 raw log messages, respectively.

Table 1. The effort reduction under three consecutive machine failures[1]

	1st Failure			2nd Failure			3rd Failure		
	Keyword Search	ICSE'13	LogCluster	Keyword Search	ICSE'13	LogCluster	Keyword Search	ICSE'13	LogCluster
WordCount	335 (16.7%)	29 (44.8%)	8 (100.0%)	818 (30.0%)	40 (65.0%)	5 (40.0%)	392 (10.0%)	25 (36.0%)	3 (66.7%)
PageRank	361 (1.1%)	18 (5.6%)	2 (50.0%)	372 (0.3%)	24 (4.2%)	2 (50.0%)	272 (0.4%)	21 (4.8%)	0 (N/A)

Table 2. The effort reduction under multiple types of failures[1]

	Machine Down			Network Disconnection			Disk Full		
	Keyword Search	ICSE'13	LogCluster	Keyword Search	ICSE'13	LogCluster	Keyword Search	ICSE'13	LogCluster
WordCount	335 (16.7%)	29 (44.8%)	8 (100.0%)	8437 (97.0%)	20 (55.0%)	6 (66.7%)	388 (24.5%)	26 (50%)	6 (100.0%)
PageRank	361 (1.1%)	18 (5.6%)	2 (50.0%)	10250 (14.0%)	23 (17.4%)	6 (66.7%)	395 (0.8%)	24 (4.2%)	4 (25.0%)

[1] Measured in terms of #log sequences to be examined. Numbers in brackets indicate the precision values (i.e., the percentage of examined log sequences that are associated with the actual failures).

Table 3. Effort reduction for Microsoft Online Service Systems

	Raw Log Messages	Keyword Search	ICSE'13	LogCluster
Service X	3.3 million	278,430 (0.01%)	522 (0.77%)	7 (42.86%)
Service Y	10.0 million	200,119 (0.08%)	2433 (2.84%)	40 (55.00%)

To further evaluate LogCluster on industrial systems, we use the real log messages (13.3 million in total) generated by Microsoft Service X and Y systems. Table 3 shows the results. LogCluster achieves significant effort reduction on real-world log data. For example, for Service X, using LogCluster we only need to examine 7 log sequences. While using the ICSE'13 approach and the traditional keyword search, we need to examine 522 log sequences and 278,430 raw log messages, respectively.

RQ2: ***How accurate is LogCluster in identifying problems?***
Tables 1 - 3 also show the precision results achieved by LogCluster, which are the percentages of examined log sequences that are indeed associated with actual failures. In general, LogCluster can achieve much higher precision values than the ICSE'13 and keyword search approaches. For example, for the WordCount application, using LogCluster, 100.0% (8 out of 8) examined log sequences are indeed related to the actual Machine Down failure, while the ICSE'13 and the keyword search approaches achieve the precision value of 44.8% (13 out of 29) and 16.7% (56 out of 335), respectively. For the Network Disconnection failure to WordCount, the precision value (66.7%) achieved by LogCluster is lower than that achieved by the keyword search approach (97.0%). However, the number of log sequences required for manual examination is much smaller (6 vs. 8437). Similarly, although for some systems, the absolute precision values are low (e.g., 42.86% for Service X), the number of log sequences need to be examined is much smaller than the total number of raw messages. Therefore, LogCluster is still considered effective in these scenarios. Figure 6 shows the average of all precision results (as listed in Tables 1 - 3) achieved by all the three approaches. Clearly, LogCluster achieves the best overall accuracy.

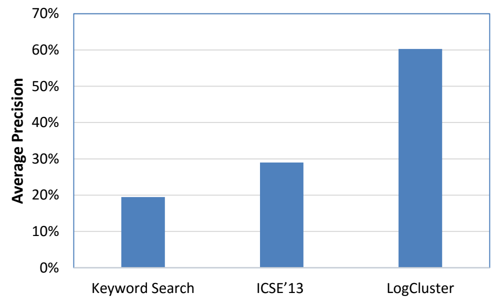

Figure 6. The average precision values

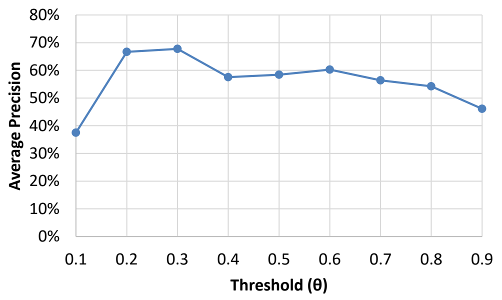

Figure 7. The impact of distance threshold θ

RQ3: ***The impact of the distance threshold***
As discussed in Sections 3.3 and 3.5, LogCluster uses a parameter θ as the distance threshold. We evaluate the impact of the distance threshold on the accuracy of LogCluster. Figure 7 shows the average precision values of all the experiments (as listed in Tables 1 - 3) achieved by different θ values. Generally, the accuracy of LogCluster is relatively stable when the θ value is between 0.2 and 0.8. The experimental results show that LogCluster is insensitive to the distance threshold.

We also use NMI (normalized mutual information) [15], which is one of the commonly used metrics to evaluate the quality of clustering. NMI is a number between 0 and 1. The higher the better. We manually examine the clusters and compute the NMI values. Table 4 shows the results when θ is 0.5. The NMI values are all above 80%, indicating good clustering quality.

Table 4. The Evaluation of Clustering Quality

	WordCount	PageRank	Service X	Service Y
NMI	90.42%	87.45%	83.48%	81.99%

4.4 Threats to Validity

We identify the following threats to validity:

- **Subject selection bias**: In our experiment, we only use four systems as experimental subjects. However, these four systems include both representative Hadoop projects as well as real-world industrial systems. In future, we will evaluate LogCluster on more projects on a variety of Cloud computing platforms such as Dryad [10].
- **Bugs in testing environment**: In our approach, we assume that all the bugs revealed by service testing are fixed. Therefore, we consider the log sequences obtained from the lab environment the "correct" ones and use them to compare with the log sequences obtained in actual environment. Our approach cannot detect erroneous log sequences from the lab environment.
- **Performance failures:** Our approach is effective in identifying functional or deployment failures. As we do not consider the temporal order of the events, our approach cannot identify performance related failures. We refer interested readers to our previous work [2] on log-based performance diagnosis.

5. SUCCESS STORY

Since 2013, LogCluster has been successfully applied to many projects in Microsoft. As an example, LogCluster has been used by Microsoft Service A team as a part of their log analysis engine. Service A is a globally deployed online service, serving millions of end users in 7*24 basis. The goal of the log analysis engine is to monitor the execution of Service A and to ensure its user perceived availability. Before adopting LogCluster, Service A team mainly used the Active Monitoring tool (also known as Synthetic Monitoring) [17] to monitor the health status of Service A. The Active Monitoring tool predefines and mimics end user requests, periodically sends these synthetic requests to the online service, and compares the content of the response with predefine correct results. Although useful, Active Monitoring fails to detect many problems because it is based on simulated user requests. LogCluster was used by the Service A team to complement Active Monitoring as it can recover actual user requests by mining execution logs. After integrating LogCluster, the Service A team is able to detect more problems and further shorten the mean time to recover the service. For example, in July 2014, due to a certain configuration fault, the component C of Service A kept calling a global topology server, which maintains the latest topology status of the overall system and provides critical information to many system functions. The component C called the topology sever in an unexpected high rate of speed and caused the server to be overloaded. As a consequence, many user requests that depend on the topology server failed. Using LogCluster, the Service A team quickly identified and fixed the problem. Furthermore, they also found a similar problem in another deployment of Service A. Using LogCluster, the service team successfully recognized the known failure and retrieved the corresponding mitigation solution.

LogCluster is also integrated into Product G, which is a product for root cause analysis of service issues. Using LogCluster, Product G builds clusters of similar log sequences mined from execution logs. Each identified cluster is assigned an anomaly score based on several criteria such as size of the cluster, age of cluster, and user provided feedback. When a service is experiencing a live site issue, engineers in the service team use Product G to examine the highest ranked anomalies that occurred around the same time as the service failure. Since many service failures manifest themselves as anomalous patterns in logs, engineers are able to quickly understand the details of the failures. This allows for more efficient root cause analysis, which in turn leads to improvement in key metrics like "Mean Time to Mitigate" and "Mean Time to Fix".

Another successful application of LogCluster is in the Product L, which is a distributed log analytic tool that can processes several TB log data every day. LogCluster is an integral component of Product L. Once Product L detects a service failure, it will automatically collect the typical log sequences and send them to service engineers for troubleshooting. Since its initial launch in July 2014, Product L has already helped identify many service problems, which have all been confirmed and fixed. For example, Product L detected a service problem that was due to a misconfiguration about "default max document size". Many users failed to upload their documents to Service B. This problem was not found in the testing environment but happened in the actual deployment of the service (with a large number of users). Product L successfully helped engineers diagnose this problem.

LogCluster is also applied to Microsoft Service C, which is a hub that hosts over four hundred different services. It maintains data from multiple Microsoft teams. Given the complexity of the Service C system, the collected logs vary from one service to the other. The diversity of logs generated by various products and teams brings much challenge to our LogCluster approach. LogCluster was integrated into the Service C team's log analysis pipeline. In the first month, the team successfully analyzed logs generated by 50 different services, without modifying any parameters. The log analysis engine was shown to be effective in assisting engineers with incident identification and diagnosis.

LogCluster is now used by many Microsoft teams and has received many encouraging feedbacks. For example:

"...the analysis pipeline is reliably running...The collaboration project was reviewed and got very positive feedback: a. from Mid-April to Mid-May (for one month), top 10 detected clusters 100% accurately detected real service issues; b. with accumulated knowledge, repeated issues could be fixed quicker/easier..." --- a senior program manager from Service A.

"...The engine was able to identify 30 anomalous patterns in service logs. Of these 29 were legitimate failures of the service which is a very high precision... It was able to quickly discover both large scale outages as well as small anomalies in services that led to customer impacting failures..." --- a principle software manager from Product G.

"...Since the launching of the analysis system, a good number of hidden issues were successfully identified and corresponding bugs were filed and fixed in the past week, which were unable to detect with other existing systems. For example, in one case, 282,904 user sessions were impacted by a config bug that direct to a wrong URL. The issue was there for more than 10 days undetected, until our analysis engine was launched and mined it out..." --- a senior developer from Service B.

6. DISCUSSIONS AND LESSONS LEARNED

6.1 Log Severity Levels

Our study finds that, Microsoft developers, like developers for open source software, use verbosity levels (such as Verbose and Medium) to control the number of printed logs. They also label the severity level of logs (such as Warning, Debug, Error, and Critical). However, our experience shows that the log severity levels can only facilitate problem diagnosis to a certain extent. This is because developers of different components often have different views about the severity of a problem. A typical online service system consists of a large number of distributed components. A failure that is considered critical to one component (such as network failure) may not significantly affect the overall system because of the fault-tolerant designs. Therefore, a log with a high severity level (such as Exception, Error, and Critical) may not reflect an actual system failure. Similarly, developers of a component may not have a complete understanding of the implications of a program status for the entire system. Therefore, a log with a low severity level (such as INFO) may actually contain important information about a system failure. As an example, we examined logs generated by Microsoft Product K over a period of 6 months. We found that only a small percentage (<10%) of high severe logs are related to the actual system failures, and many (>30%) failures are associated with logs that have low severity levels. Our proposed LogCluster does not rely on log severity levels. It is based on abstraction and clustering of log sequences, therefore avoiding the limitations of using log severity levels.

6.2 Permutations of a Log Sequence

Our approach, like the ICSE'13 work, does not consider the permutations of events in a log sequence. For example, we consider the following two sequences the same: "E1, E2, E3, E5, E6" and "E1, E3, E5, E2, E6". This is because many tasks of an online service are multi-threading, which causes interleaving logs even for the same user request. Furthermore, a typical online service system consists of many distributed servers. The logs generated by each server are later consolidated and stored at a HDFS-like central place. However, due to the clock drift problem [21], the timestamps of events produced by different servers may loss synchronization, causing many different permutations of events for the same execution sequence. Therefore, in our work we do not consider the permutations of a log sequence.

6.3 Deployment Failures of Online Service Systems

Our experience shows that when an online service system is initially launched, many failures are related to functional features. Many new log clusters obtained by LogCluster correspond to new features. When the service system becomes stable, deployment failures account for a large percentage of failures of online service systems. The deployment failures are often caused by environmental issues, such as issues in network connection, DNS, configuration, hardware, etc. In production environment, the deployment of an online service system is typically performed in an incremental manner. The deployment topology is divided into multiple *farms*. The system is firstly deployed in a small number of farms and then gradually moved to other farms. At each deployment step, the scales of system and data are increased. If LogCluster detects a new cluster of log sequences in a new farm, it is likely that the new farm encountered a deployment issue. Furthermore, a deployment issue occurs in one farm could happen in other farms as well. Using LogCluster, developers can quickly detect the recurrent deployment failures and find mitigation solutions from the knowledge base, thus reducing diagnosis and maintenance effort.

In ideal cases, engineers can identify the root cause of the incident and fix it quickly. However, in most cases, engineers are unable to identify and fix root causes within a short time. Thus, in order to recover the service as soon as possible, a common practice is to restore the service by identifying a temporary workaround solution (such as restarting a server) to restore the service. Then after service restoration, identifying and fixing the underlying root cause for the incident can be conducted via offline postmortem analysis.

6.4 Log Event IDs

Our experience shows that log parsing accounts for a large portion of computation time of LogCluster. During log parsing, we process the raw log messages, parse them, and convert them into log events. The log events can be regarded as the generic log messages printed by the same log-printing statement in the source code. Some of the Microsoft products we worked on provide directly the log event IDs - each log message contains an event ID, a log level, and log contents. In this way, much time and computing resource are saved during log analysis. We consider it a good practice to directly add a log event ID to each log-printing statement in source code. It is also possible to develop a tool to automatically scan the logging statements and generate a unique ID for each log message, before the source code is submitted to the version control repository.

6.5 Distributed Computing

For the Microsoft online services we worked on, the log data is usually at very large scale (TeraBytes or PetaBytes every day). The large amount of log data demands much computing resource. To reduce the computation time, in practice our analysis algorithm is deployed in an internal distributed computing environment, with tens to hundreds servers. Furthermore, we select algorithms that are more suitable for a distributed computing environment. For example, we have tried several commonly-used clustering algorithms such as K-Means, K-Medoids, DBSCAN, and hierarchical clustering. Finally, we select the hierarchical clustering algorithm because it works well in a distributed environment.

7. RELATED WORK

Logging is widely used for diagnosing failures of software-intensive systems because its simplicity and effectiveness. Analyzing logs for problem diagnosis has been an active research area [12, 13, 16, 24, 25, 26]. These work retrieve useful information from logs (such as events, variable values, and locations of logging statements), and adopt data mining and machine learning techniques to analyze the logs for problem detection and diagnosis. For example, Lou et al. [12] mine invariants (constant linear relationships) from console logs. A service anomaly is detected if a new log message breaks certain invariants during the system execution. Xu et al. [24] preprocess the logs and detect anomalies using principal component analysis (PCA). The log-based anomaly detection algorithms can check whether a service is abnormal, but can hardly obtain the insights into the abnormal task. LogEnhancer [27] aims to enhance the recorded contents in existing logging statements by automatically identifying and inserting critical variable values into them. The work of [6] records the runtime properties of each request in a multi-tier Web server, and applies statistical learning techniques to identify the causes of failures. Unlike the above-mentioned work, our work facilitates problem identification for online service systems by clustering similar logs.

Some log-based diagnosis work is also based on the similarity among log sequences. For example, Dickenson et al. [1] collected execution traces and used classification techniques to categorize the collected traces based on some string distance metrics. Then, an analyst can examine the traces of each category to determine whether or not the category represents an anomaly. Yuan et al. [26] proposed a supervised classification algorithm to categorize system traces based on the similarity to the traces of the known problems. Mirgorodskiy et al. [14] used string distance metrics to categorize function-level traces, and to identify outlier traces or anomalies that substantially differ from the others. Ding et al. [3, 4] designed a framework to correlate logs, system issues, and corresponding simple mitigation solutions when similar logs appear. In our work, we consider weights of different events and apply hierarchical clustering to cluster similar log sequences. We also compare the newly obtained log sequences with those of known failures.

While most of research has focused on the usage of logs for problem diagnosis, recently much work has been conducted to understand the log messages and logging practices. For example, Yuan et al. [28], Shang et al. [19], and Fu et al. [7] reported empirical studies on logging practice in open source and industrial software. Zhu et al. [29] proposed a "learning to log" framework, which aims to provide informative guidance on logging. Additionally, Shang et al. [18] used a sequence of logs to provide context information when examining a log message. To facilitate the understanding of log messages, Shang et al. [20] further proposed to associate the development knowledge stored in various software repositories (e.g., code commits and issues reports) with the log messages. In our work, the obtained log clusters and representative log sequences could also help engineers understand different categories of log messages.

8. CONCLUSIONS

Online service systems generate a huge number of logs every day. It is challenging for engineers to identify a service problem by manually examining the logs. In this paper, we propose LogCluster, an approach that clusters the logs to ease log-based problem identification. LogCluster also utilizes a knowledge base to reduce the redundant effort incurred by previously examined log sequences. Through experiments on two representative Hadoop-based apps and two Microsoft online service systems, we show that our approach is effective and outperforms the state-of-the-art work proposed in ICSE 2013 [18]. We have also described the successful applications of LogCluster to the maintenance of actual Microsoft online service systems, as well as the lessons learned.

In the future, we will integrate LogCluster into an intelligent and generic Log Analytics engine. We will also investigate effective log-based fault localization and debugging tools, such as those described in [23].

Acknowledgement

We thank the intern students Can Zhang and Bowen Deng for the helpful discussions and the initial experiments. We thank our product team partners for their collaboration and suggestions on the applications of LogCluster.

9. REFERENCES

[1] W. Dickinson, D. Leon, and A. Podgurski, Finding Failures by Cluster Analysis of Execution Profiles. In *Proc. of the 23rd International Conference on Software Engineering (ICSE 2001)*, May 2001. pp. 339 - 348.

[2] R. Ding, H. Zhou, J. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, T. Xie. *Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis*. In *Proc. of the 2015 USENIX Annual Technical Conference (USENIX ATC '15),* Santa Clara, CA, USA. pp. 139-150, July 2015.

[3] R. Ding, Q. Fu, J. Lou, Q. Lin, D. Zhang, J. Shen, and T. Xie, Healing online service systems via mining historical issue repositories. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, September 2012, 318-321.

[4] R. Ding, Q. Fu, J. Lou, Q. Lin, D. Zhang, and T. Xie, Mining historical issue repositories to heal large-scale online service systems. In *Proc. 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, Atlanta, GA, USA, pp. 311–322.

[5] Failover mechanism, https://en.wikipedia.org/wiki/Failover

[6] Q. Fu, J. Lou, Y. Wang, and J. Li, Execution anomaly detection in distributed systems through unstructured log analysis. In *Proc. of the 9th IEEE International Conference on Data Mining (ICDM 2009)* Miami, Florida, USA, December 2009. pp. 149-158.

[7] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Proc. of the 36th International Conference on Software Engineering (ICSE 2014),* June 2014, pp. 24-33.

[8] J. C. Gower, G. J. S. Ross, "Minimum spanning trees and single linkage cluster analysis*", Journal of the Royal Statistical Society, Series C* 18 (1): 54–64, 1969.

[9] Hadoop. http://hadoop.apache.org/core.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", In *Proc. of EuroSys 2007*, Mar 2007.

[11] Z. M. Jiang, A. E. Hassa, P. Flora, and G. Hamann, "Abstracting Execution Logs to Execution Events for Enterprise Applications", in *Proc. of the 8th International Conference on Quality Software (QSIC 2008)*, pp.181-186, 2008.

[12] J. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, Mining invariants from console logs for system problem detection. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC'10),* Boston, MA, USA, June 2010.

[13] D. Lo, H. Cheng, J. Han, S. C. Khoo, and C. Sun, Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proc. of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2009)*, Paris, France, June 2009, pp. 557-566.

[14] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, "Problem Diagnosis in Large-Scale Computing Environments", In *Proceedings of the ACM/IEEE SC 2006 Conference*, Nov. 2006.

[15] C. D. Manning, P. Raghavan and H. Schütze. *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[16] T. Reidemeister, M. Jiang, and P. Ward, Mining unstructured log files for recurrent fault diagnosis. In *Proc. of the 12th IFIP/IEEE International Symposium on Integrated Network Management,* Dublin, Ireland, May 2011, pp. 377-384.

[17] Active Monitoring, available at: https://en.wikipedia.org/wiki/Synthetic_monitoring

[18] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, P. Martin, "Assisting developers of Big Data Analytics Applications when deploying on Hadoop clouds," in *Proc. of the 35th International Conference on Software Engineering (ICSE 2013),* pp.402-411, May 2013.

[19] W. Shang, M. Nagappan, and A. E. Hassan. Studying the relationship between logging characteristics and the code quality of platform software. *Empirical Software Engineering*, 20(1), Feb 2015, pp. 1-27.

[20] W. Shang; M. Nagappan, A. E. Hassan, Z. M. Jiang, Understanding Log Lines Using Development Knowledge. In *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME 2014),* Sept 2014, pp.21-30.

[21] A. Tanenbaum and van S. Maarten, *Distributed Systems : Principles and Paradigms*, Prentice Hall, 2002.

[22] ULS (Unified Logging Service), available at: http://weblogs.asp.net/erobillard/sharepoint-trace-logs-and-the-unified-logging-service-uls

[23] R. Wu, H. Zhang, S. C. Cheung, and S. Kim, CrashLocator: locating crashing faults based on crash stacks. In *Proc. of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014),* pp. 204-214, July 2014.

[24] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, Big Sky, Montana, USA, October 2009, pp. 117-132.

[25] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, Experience mining Google's production console logs. *Proceedings of SLAML,* Vancouver, Canada, October 2010.

[26] C. Yuan, N. Lao, J.R. Wen, J. Li, Z. Zhang, Y.M. Wang, and W. Y. Ma, "Automated Known Problem Diagnosis with Event Traces", In *Proceeding of EuroSys 2006*, April 2006.

[27] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, Improving software diagnosability via log enhancement. In *ACM Transactions on Computer Systems,* 30(1), Feb. 2012.

[28] D. Yuan, S. Park, and Y. Zhou, Characterizing logging practices in open-source software. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012),* Zurich, Switzerland, June 2012, pp. 102-112.

[29] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, D. Zhang, Learning to Log: Helping Developers Make Informed Logging Decisions, In *Proc. of the 37th International Conference on Software Engineering (ICSE 2015)*, Florence, Italy, May 2015.

[30] Sigmoid function, available at: https://en.wikipedia.org/wiki/Sigmoid_function.