Research Article

Log Pattern Mining for Distributed System Maintenance

Jia Chen , Peng Wang , Shiqing Du, and Wei Wang

School of Computer Science, Fudan University, Shanghai 200082, China

Correspondence should be addressed to Peng Wang; pengwang5@fudan.edu.cn

Received 7 October 2020; Revised 27 October 2020; Accepted 16 November 2020; Published 1 December 2020

Academic Editor: Jianxin Li

Copyright © 2020 Jia Chen et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Due to the complexity of the network structure, log analysis is usually necessary for the maintenance of network-based distributed systems since logs record rich information about the system behaviors. In recent years, numerous works have been proposed for log analysis; however, they ignore temporal relationships between logs. In this paper, we target on the problem of mining informative patterns from temporal log data. We propose an approach to discover sequential patterns from event sequences with temporal regularities. Discovered patterns are useful for engineers to understand the behaviors of a network-based distributed system. To solve the well-known problem of pattern explosion, we resort to the minimum description length (MDL) principle and take a step forward in summarizing the temporal relationships between adjacent events of a pattern. Experiments on real log datasets prove the efficiency and effectiveness of our method.

1. Introduction

With the increasing demand for computing power, many network-based distributed systems have emerged, such as the popular distributed storage system and HDFS. Distributed system utilizes multiple machine nodes to complete tasks based on the network. Since the network may be complex and each node in the network may report anomalies, to maintain the network-based distributed application, instead of monitoring each node of the network, experts usually analyze the node logs to evaluate the health of the system. Logs record the running status and significant events such as the starting and ending of a task of the system. Mining information from log sequences is a useful way to understand network-based distributed system behaviors.

Many researchers have proposed different log analysis approaches in recent years. Some try to sketch the operation process of the system [1–3], and some devote to functional tasks, such as anomaly detection and problem diagnosis [4, 5]. In this paper, we treat the timestamped log messages as an event sequence produced by a black-box system. We target on the problem of mining informative patterns from temporal log sequence data. Here, by informative, we mean a set of patterns that can summarize the log sequence well. The discovered patterns can help the operation engineers to have

a better understanding of the system behaviors and serve as an excellent source of information for online monitoring and anomaly detection. Our work is motivated by the well-known distributed file system HDFS. Many informative patterns are generated during system operation, corresponding to different system behaviors.

As the centerpiece of the HDFS, one important function of the NameNode is to determine the mapping of blocks of a file to DataNodes when a new file is written to the HDFS. For the NameNode, the write operation regularly generates a pattern as shown in Table 1. Normally, when a new data block is allocated, three copies of this block are created in different DataNodes, and after creating required replicas, NameNode updates the block map. This typical system behavior can be represented by a sequential pattern $P = \langle 132, 134, 134, 134 \rangle$.

Based on capturing this type of patterns, we can further implement several log analysis applications. For example, with the increasing amount of log data generated by the complex system, it is rather difficult for engineers to check the log messages one by one. Compressing the log data to a smaller size without losing the important information is of high demand. Since the pattern represents the semantic of the data block allocating behavior, we can reduce the number of logs greatly by encoding the original log sequence

TABLE 1: Log file example indicating a meaningful pattern.

Event type	Timestamp	Log message
132	2018-07-05 16:25:03,718	StateChange: BLOCK * allocate blk_1073741825_1001 for /hbase/tmp/hbase.version
134	2018-07-05 16:25:03,959	BlockStateChange: BLOCK * addStoredBlock: blockMap updated: 10.141.208.49:50010 is added to blk_1073741825_1001
134	2018-07-05 16:25:03,961	BlockStateChange: BLOCK * addStoredBlock: blockMap updated: 10.141.208.43:50010 is added to blk_1073741825_1001
134	2018-07-05 16:25:03,988	BlockStateChange: BLOCK * addStoredBlock: blockMap updated: 218.193.132.35:50010 is added to blk_1073741825_1001

with the discovered patterns. Besides, we can investigate the historical condition of the cluster by counting the number of newly added data blocks in the last week.

Sequential pattern mining has become an important data mining problem for decades. Dozens of research efforts focus on the problem of finding sequential patterns effectively [6–8]. However, traditional mining approaches often generate a huge number of patterns that confuse users a lot. To solve the well-known problem of pattern explosion [9], we resort to the minimum description length (MDL) principle [10], which has been used in several previous works [9, 11, 12]. The MDL principle provides a good balance between the complexity of the result pattern set and its ability to represent the data.

Although our work is not the first work that discovers patterns by utilizing the MDL principle, we take a step forward in summarizing the temporal relationships among events of a pattern. Logs monitor repetitive behaviors corresponding to execution traces of several program statements. It is a common case that there exist significant temporal regularities in system logs. Consider an example in Figure 1, which shows simplified log messages for two behaviors of the NameNode. We use capitals to represent event types and lowercase characters to represent corresponding occurrences. There exist two patterns: $P_1 = ABC$, where B is generated after A every 2 or 5 seconds and C is generated after B every 3 seconds. $P_2 = DD$, where the interval between D is 1 or 2 seconds. Obviously, patterns summarizing the temporal relations between adjacent events indicate more information of the running status of the system.

However, none of these methods fully consider the temporal relationships. From the perspective of handling temporal relations between adjacent events of a pattern, existing approaches can be divided into two groups. GoKrimp [9] and SQS [12] punish gaps by allocating higher cost when encoding patterns with large gaps, which do not consider the regularity of interarrival times at all. CSC [11] considers patterns restricted to fixed intervals and does not allow any event type appearing more than once, which strongly limits the expressiveness of a pattern. The generated patterns of previous methods either mix together and result in high redundancy [9, 12] or are too simple to represent the true behaviors [11].

In this paper, we propose an approach DTS (discovering patterns from temporal log sequences) to remedy the defects. To the best of our knowledge, our work is the first work

proposed to encode the temporal regularity of patterns in an event sequence. The encoding scheme enables us to discover high-quality patterns with low redundancy.

The key contributions of our work include the following:

- (1) We formalize how to use histograms to describe the distribution of time intervals between adjacent events in a pattern. Moreover, we design an encoding scheme that compresses the original sequence as well as the temporal regularities with the mined patterns in a lossless way.
- (2) We introduce a heuristic algorithm DTS that discovers a set of informative patterns both effectively and efficiently.
- (3) Evaluation results on real datasets show that our methods are capable of discovering high-quality patterns with low redundancy.

The rest of this paper is organized as follows. In Section 2, we give the preliminary knowledge about our work, including the pattern semantic and the problem statement. The encoding scheme for temporal data is described in Section 3. In Section 4, we present the algorithm for mining patterns that best compress the log sequence. We evaluate the effectiveness and efficiency of our approaches in Section 5. We discuss related work in Section 6 and conclude this paper in Section 7.

2. Preliminaries

2.1. Log Sequence and Log Parsing. A log sequence is a sequence of log entry and timestamp pairs, denoted as $S_{\log} = \{(\log_1, t_1), (\log_2, t_2), \dots, (\log_n, t_n)\} (t_i \leq t_{i+1}, 1 \leq i \leq n)$, where $\log_i$ is the i -th log entry and t_i is the corresponding timestamp. The granularity of timestamps can be set to any level depending on the application.

Raw logs are usually unstructured, free-text messages. To analyze log data, a common preprocessing step is to parse unstructured log messages into structured representations [13, 14]. Concretely, log messages printed by the same statement are often highly similar to each other and vice versa [13]. Based on this observation, we can extract two parts of information from a log message: event type and parameters. Event type refers to the constant content shared by all logs generated by the same statement, while parameters are the values of variables that differ from log to log. In

FIGURE 1: Toy example of three possible encodings, DTS, SQS, and CSC, respectively.

this paper, we use Drain [14] to parse the original log messages due to its popularity and excellent performance. For each log message $\log_i$, we store its event type and parameters with a structured representation, denoted as a tuple $[e_i, \text{para}_1, \text{para}_2, \dots, \text{para}_p]$, where e_i refers to an instance of the event type of $\log_i$, para_j ($1 \leq j \leq p$) refers to the values of the parameters, and p is the number of parameters in $\log_i$. We assume that the event types of all logs in $S_{\log}$ come from a finite alphabet $\Sigma = \{E_1, E_2, \dots, E_l\}$. Therefore, the original log sequence is parsed into a temporal event sequence $S = \{(e_1, t_1), (e_2, t_2), \dots, (e_n, t_n)\}$. We use $\Delta(S)$ to denote the time spanned by S , i.e., $\Delta(S) = t_n - t_1$. For brevity, we omit the parameters when unnecessary.

2.2. Pattern Discovery with the MDL Principle. In this work, our goal is to turn the raw log sequence into a sequence of more informative patterns. Specifically, we aim to discover a set of informative patterns which can (1) represent event relationships and temporal regularities and (2) provide the best lossless compression of the event sequence. This is achieved based on the minimum description length (MDL) principle [10].

2.2.1. A Brief Introduction to MDL. We use MDL as a metric to balance data compression quality and complexity. Specifically, we apply the two-part MDL principle, which can be roughly described as follows. Given a set of models $\mathcal{M}$, the best model $M \in \mathcal{M}$ is the one that minimizes $L(S, M) = L(M) + L(S|M)$, where $L(M)$ is the description length of M and $L(S|M)$ is the description length of sequence S when encoded with M . The description length is computed at the bit level. In this paper, the model refers to a set of patterns $\mathcal{P}$. We will define the encoding scheme as to how to describe S with a set of patterns and how to compute the description length later. Thus, our problem of pattern discovery can be formulated as follows.

Problem Definition. Given a temporal log sequence S , find the set of patterns $\mathcal{P}$ that minimizes the description length $L(S, \mathcal{P}) = L(\mathcal{P}) + L(S|\mathcal{P})$.

2.2.2. Semantics of the Pattern Language. To use MDL, we first need to specify the pattern language, which determines the “vocabulary” of all possible patterns we can discover given S . Specifically, the key elements of a pattern P_i are defined as follows:

- (1) **Content:** the content of P_i is denoted as $E_1, E_2, \dots, E_m$, where $E_j \in \Sigma$ ($1 \leq j \leq m$), which can be used as a unique identifier of each pattern. The length of the episode of P_i is m .
- (2) **Occurrence (set):** an occurrence of pattern P_i in S is denoted as $O_{P_i} = \langle (e_1, t_1), (e_2, t_2), \dots, (e_m, t_m) \rangle$, a list of events e_j ordered by time t_j , where $(e_j, t_j) \in S$ ($1 \leq j \leq m$) with e_j denoting an instance of event type $E_j \in P_i$. The occurrence set of a pattern is denoted as $\mathcal{O}_{P_i}$. In this paper, we consider the left-most occurrences of P_i .
- (3) **Support:** the support of a pattern P_i is the number of its nonoverlapping occurrences, denoted as supp_i . Two occurrences of P_i are nonoverlapping if they do not share common events. We also do not allow overlapping among occurrences of different patterns.
- (4) **Time interval distribution:** given the occurrence set of P_i , we use a histogram h_i^k to describe the distributions of time intervals between adjacent events e_k and e_{k+1} ($1 \leq k \leq m-1$) in P_i . Specifically, the bin $h_i^k[b]$ counts the number of time intervals that equal b .

3. Data Encoding Scheme

In this section, we present our encoding scheme and the way we compute description lengths. It is worth noting that we are only interested in code length functions instead of actual encodings [10]. All logarithms in this paper are to base 2.

3.1. Encoding the Pattern Set. Under the MDL criterion, we need to first encode the model, which is the pattern set $\mathcal{P}$ in our paper. Here, we use $\mathcal{P}$ to store the elements of all patterns, which comprise the items in the header row of Figure 2. The event sequence S can then be compressed by replacing the occurrences of patterns with the corresponding codes. For each pattern, we need to encode its episode and histograms which describe the distribution of time intervals between adjacent events. Although most of the events in S are covered by patterns, some events can be left out. Each of such events is covered by a special pattern with only one item in its content (namely, the type E_i of the event), which is denoted as a *singleton pattern* which has no histogram.

As with description length computation, first we discuss how to calculate $L(P_i)$. We use $\log(|\Sigma|)$ bits to describe the content of each event type. Therefore, episode α_{P_i} of length m

Pattern	Support	Code (P)	Histogram	Code ($h_i^k[b]$)
$P_1 = \langle ABC \rangle$	20	$P(P_1)$		$C(h_1^1[2])$ $C(h_1^1[5])$ $C(h_1^2[3])$
$P_2 = \langle DD \rangle$	60	$P(P_2)$		$C(h_2^1[1])$ $C(h_1^2[2])$
$L(P_1) = 3\log(10) + C P_1 + C(h_1^1[2]) + \log(\Delta(S)) + \log(20)$ $+ C(h_1^1[5]) + \log(\Delta(S)) + \log(20)$ $+ C(h_1^2[3]) + \log(\Delta(S)) + \log(20)$ $L(P_2) = 2\log(10) + C P_2 + C(h_2^1[1]) + \log(\Delta(S)) + \log(60)$ $+ C(h_1^2[2]) + \log(\Delta(S)) + \log(60)$				
$L(P) = L(P_1) + L(P_2)$				

FIGURE 2: Toy example of a pattern set $\mathcal{P}$ and the description length of pattern P_1 .

needs $m \times \log(|\Sigma|)$ bits to describe. We use a unique code, denoted as $\mathcal{C}(P_i)$, to encode each pattern. Intuitively, frequent patterns should have shorter description lengths. Hence, the length of the binary code depends on the pattern's frequency f_i , denoted as $f_i = (\text{supp}_i/F)$, where $F = \sum_{P_i \in \mathcal{P}} \text{supp}_i$ is the sum of all patterns' support. Here, we consider optimal prefix codes, where the code length is $|\mathcal{C}(P_i)| = -\log(f_i)$.

As with the histogram h_i^k , let the number of nonempty bins in h_i^k be cnt_{bins} . Since the bin width is set to the time unit of S , the more regular each value of the time interval between adjacent events in P_i is, the less diverse these values are and the smaller cnt_{bins} is. Consider the value of the ratio $r_{h_k} = (\text{cnt}_{\text{bins}}/\text{supp}_i)$. A smaller value of r_{h_k} indicates better temporal regularity between e_k and e_{k+1} . This observation allows us to design an encoding scheme that compresses the temporal regularity of pattern P_i 's occurrences.

To be specific, the value of a nonempty bin $h_i^k[b_j]$ is denoted as $\text{cnt}_{b_j} = |\{O_{P_i} | t_{k+1} - t_k = b_j\}|$, where b_j is the difference between the timestamps of two adjacent events and cnt_{b_j} is the number of intervals in $h_i^k[b_j]$. We use $\log(\Delta(S))$ number of bits to encode b_j and $\log(\text{supp}_i)$ number of bits to encode cnt_{b_j} . We also need extra bits to identify each nonempty bin, whose length is dependent on the size of $h_i^k[b_j]$. Concretely, the more intervals that equal b_j , the shorter the code length, which is $|\mathcal{C}(h_i^k[b_j])| = -\log(\text{cnt}_{b_j}/\text{supp}_i)$. Thus, the information of a histogram h_i^k can be described using $L(h_i^k) = \sum_{j=0}^{\text{cnt}_{\text{bins}}} (\log(\Delta(S)) + \log(\text{supp}_i) + |\mathcal{C}(h_i^k[b_j])|)$ bits. To sum up, the description length of a pattern in $\mathcal{P}$ is

$$L(P_i) = m \times \log(|\Sigma|) + |\mathcal{C}(P_i)| + \sum_{k=1}^{m-1} L(h_i^k). \quad (1)$$

Figure 2 gives a concrete example on how the description length of a pattern is computed. For all patterns in $\mathcal{P}$, the total description length is

$$L(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} L(P_i). \quad (2)$$

3.2. Encoding the Event Sequence. Given the encoded pattern set $\mathcal{P}$, we can encode the event sequence S by replacing the occurrences of patterns in $\mathcal{P}$ with their codes. Since S is a temporal sequence, for each occurrence of a pattern, we need to cover all events in it as well as their timestamps. In our encoding scheme, we replace an occurrence with the code of the corresponding pattern P_i and the timestamp of its first event. The description length of a timestamp is $\log(\Delta(S))$. Moreover, we use the codes of histogram bins to specify the time intervals between adjacent events. We individually encode each bin $h_i^k[b_j]$ in h_i^k . The resulting encoding is $(\mathcal{C}(P_i), \Delta(S))\mathcal{C}(h_i^1[b_{j_1}]), \dots, \mathcal{C}(h_i^{m-1}[b_{j_{m-1}}])$. For the remaining events that are not covered by any pattern, we simply use singleton patterns E_i to encode them, i.e., $(\mathcal{C}(E_i), \Delta(S))$.

As a concrete example, we encode the sequence S in Figure 1 using the pattern set in Figure 2. Here, S is encoded by replacing the original events and timestamps in each occurrence of each pattern P_i with the corresponding codes. For example, $\{(a, 0)(b, 2)(c, 5)\}$ is an occurrence of P_1 , and it is replaced by $(\mathcal{C}(P_1), \Delta(S))\mathcal{C}(h_1^1[2])\mathcal{C}(h_1^2[3])$ in the encoded sequence.

With this encoding scheme, the description length of sequence S encoded by $\mathcal{P}$ is calculated as follows:

$$L(S|\mathcal{P}) = \sum_{P_i \in \mathcal{P}} \left(|\mathcal{C}(P_i)| + \log(\Delta(S)) + \sum_{k=1}^{m-1} |\mathcal{C}(h_i^k[b])| \right). \quad (3)$$

4. DTS: Discovering Patterns from Temporal Sequences

In this section, we introduce our DTS algorithm for pattern discovery. The problem of discovering a set of patterns $\mathcal{P}$ that best compresses S based on MDL has been proven to be NP-hard [9]. Therefore, we resort to discovering informative patterns heuristically instead of pursuing the optimal result. We assume sequence S is encoded by singleton patterns at the very beginning. Our DTS algorithm starts from an empty pattern set $\mathcal{P}$ and updates $\mathcal{P}$ iteratively until we cannot achieve no more compression.

4.1. Overall Process of DTS. We now present an overview of our DTS algorithm. A naïve strategy is to update the pattern set $\mathcal{P}$ by inserting the best pattern (the one that can achieve the largest gain in compression) in the current sequence S . However, for system logs, it is common that the execution of a task generates several different branches. For example, Figure 3 shows a simplified process of an HDFS client writing files. Each time the client finishes writing a data block, it will call `addBlock()` to allocate a new block. Each time the DataNode receives a new block, it will report to the NameNode, thus generating a sequence $\langle XYYY \rangle$. Also, the

FIGURE 3: A simplified process of an HDFS client writing files.

NameNode sometimes calls $fsync()$ to persist the cached data, thus triggering the log event M . Therefore, another sequence $\langle XMYYY \rangle$ can also be generated during the writing process.

The common part of these two sequences, i.e., $\langle XYYY \rangle$, can be easily discovered and inserted into the set $\mathcal{P}$. Nonetheless, it is much harder to recognize M together with $\langle XYYY \rangle$. In practical maintenance applications, we hope to identify the behavioral patterns more completely. This cannot be achieved using the aforementioned naïve strategy.

Faced with this problem, we divide the update operations of $\mathcal{P}$ into two types: the insertion operation and the refinement operation. Their definitions are as follows.

Definition 1 (insertion operation). Insert the newly discovered candidate pattern P into $\mathcal{P}$. The pattern set is updated to $\mathcal{P} \cup P$.

Definition 2 (refinement operation). Given a pattern $P = \langle E_1 E_2, \dots, E_m \rangle$ ($P \in \mathcal{P}$), we refine it by adding events of type E at position j ($j \in [0, \dots, m]$) of P . The result is a new pattern P_r with length $m + 1$. Specifically,

- (1) If $j = 0$, $P_r = EE_1 E_2, \dots, E_m$
- (2) If $j \in [1, \dots, m - 1]$, $P_r = E_1, \dots, E_j EE_{j+1}, \dots, E_m$
- (3) If $j = m$, $P_r = E_1 E_2, \dots, E_m E$

The refinement operation is denoted as $R(P)_j^E$, where E is called as the refining event. The pattern set is updated to $\{\mathcal{P} \setminus P\} \cup \{P_m, P_r\}$, where P_m denotes the remaining original pattern. If there are remaining occurrences of the original P , the operation is called a partial refinement; otherwise, it is called a full refinement, and then P_m is null. Figure 4 presents an example of the refinement operation.

The pattern set $\mathcal{P}$ is updated iteratively by either inserting a new pattern to $\mathcal{P}$ or refining a pattern that is already in $\mathcal{P}$. It is common that an event covered by a candidate pattern P is also a refining event for another pattern $P' \in \mathcal{P}$. We refer to such a case as an event conflict. In each iteration, if there is no event conflict, we can greedily add the best candidate pattern to $\mathcal{P}$. However, if there exists an event conflict, since our encoding scheme does not allow overlap among different patterns, it is necessary to decide

which update operation is better. Here, we choose the operation that decreases the description length $L(\mathcal{P}) + L(S|\mathcal{P})$ more. After choosing an update operation, we encode the events covered by the new patterns with the corresponding codes. The pattern set $\mathcal{P}$ is updated iteratively until we cannot decrease the description length any more.

The overall process of DTS is divided into two stages: the update stage driven by the event conflict and the update stage containing only refinement. We maintain two data structures to support the two update operations: a candidate pattern set $\mathbb{CP}$ for insertion and a candidate refinement map $\mathbb{CR}$ for refinement. $\mathbb{CP}$ stores the candidate patterns in the current sequence dynamically, while $\mathbb{CR}$ stores the possible refinements for each pattern in $\mathcal{P}$. As shown in Figure 5, at the first stage, DTS iteratively compares the compression gain of the candidate pattern and the refinement result, selecting the better one to update $\mathcal{P}$. When $\mathbb{CP}$ becomes empty, namely, we cannot get a valid candidate pattern any more, DTS continues with the second stage, iteratively updating $\mathcal{P}$ with the best refinement in $\mathbb{CR}$ until $\mathbb{CR}$ is empty. We now move on to the technical details of our DTS algorithm.

4.2. Candidate Pattern Set $\mathbb{CP}$. Concerning the insertion operation, we hope the pattern P added to $\mathcal{P}$ will best decrease the description length. To this end, we define the compression gain of a pattern P as follows:

$$\text{gain}(P) = L(S, \mathcal{P}) - L(S, \mathcal{P} \cup P). \quad (4)$$

A naïve way of selecting the best candidate pattern is to enumerate all possible patterns exhaustively. However, this is time-consuming due to the exponential search space. In practice, instead of searching from scratch in each iteration, we maintain a candidate pattern set $\mathbb{CP}$ that stores promising candidate patterns. We initialize $\mathbb{CP}$ with Algorithm 1. For each event type $E \in \Sigma$, best growth $(\alpha, S, \mathcal{E}, P)$ generates a pattern starting from a singleton pattern $P = \langle E \rangle$ by appending any event type E' to the end of α greedily. Since it is computationally intensive to find the optimal occurrence set having the fewest histogram bin, we resolve to a simple approach that finds the leftmost occurrences of P . If the newly generated pattern yields greater compression gain than the previous one, we move on to test the next pattern created by a new growth. The process stops until we cannot get a new pattern better than the previous one. We select the last pattern P_{last} as the best pattern starting with E and add it to $\mathbb{CP}$.

4.3. Candidate Refinement Map $\mathbb{CR}$. Considering the refinement operation, we maintain a candidate refinement map $\mathbb{CR}$ to store the best refinements for each pattern $P \in \mathcal{P}$. Each refinement follows the format $R(P)_j^E = [\text{refined} - \text{position } j, \text{event} - \text{type } E, \text{remaining} - \text{pattern } P_m, \text{refined} - \text{pattern } P_r]$. For example, $[1, F, D D, DF D]$ indicates that a pattern DD is partially refined by event F at position 1, generating a new pattern $DF D$ with some occurrences of the original pattern DD

FIGURE 4: An example of the refinement operation. (a) Refining an occurrence. (b) Refining a pattern.

FIGURE 5: Overall process of DTS.

remaining. On the contrary, if D is fully refined by F , the refinement result will be written as $[1, F, \text{null}, DF D]$. The refinements in $\mathbb{CR}$ are indexed by the refining event type. In other words, the refinement results with the same event type are grouped together.

When the pattern set $\mathcal{P}$ is updated by a new pattern P , which is either selected from $\mathbb{CP}$ or generated by a refinement operation, we decide whether there exists a good refinement for each position by Algorithm 2. For each position j , we only store the refinement that best decreases the encoded description length. The compression gain is defined in the following equation:

$$\text{gain}(R(P)_j^{E'}) = L(S, \mathcal{P}) - L(S, (\mathcal{P} \setminus P) \cup \{P_m, P_r\}). \quad (5)$$

The way we obtain a refinement $\text{REFINE}(P, S, j, E)$ is as follows. The content of the newly refined pattern P_r can be easily obtained by Definition 2. The occurrences of P_r can be obtained from the original occurrence set of P . For each occurrence $O_P = \langle (e_1, t_1), (e_2, t_2), \dots, (e_m, t_m) \rangle \in \mathcal{O}_P$, we find the leftmost event instance (e_i, t_i) in S that meets the time restriction and refine O_P with it.

4.4. The Complete DTS Algorithm. In this section, we present the complete DTS algorithm for heuristic discovery of the pattern set $\mathcal{P}$. As is shown in Algorithm 3, we update $\mathcal{P}$

iteratively until we cannot decrease the description length of S any more. At the beginning, we initialize the relevant data structures. Lines 4–17 correspond to the update stage driven by the event conflict. During this stage, DTS first gets the best candidate pattern P^* in $\mathbb{CP}$. Then, DTS calls the function `BestConflictRefinement` ($P, \mathbb{CR}$) to check if there exist one or more refinements in $\mathbb{CR}$ that conflict with P^* and retrieve the best one $R^*(P')^E$. DTS decides which one is better to update the pattern set $\mathcal{P}$ by comparing the compression gains of P^* and $R^*(P')^E$ with the following equation:

$$\text{cmp}(P^*, R^*(P')^E) = \frac{\text{gain}(P^*)}{|P^*| * |\mathcal{O}_P|} - \frac{\text{gain}(R^*(P')^E)}{|\mathcal{O}_{P'}|}. \quad (6)$$

Note that we compare the compression gains of the two operations at the event level in order to avoid bias towards patterns with longer episodes or greater frequency. A positive value of $\text{cmp}(P^*, R^*(P')^E)$ indicates that P has greater compression gain. Otherwise, $R^*(P')^E$ is better. If P^* is better or there is no conflicting refinement, $\mathcal{P}$ will be updated to $\mathcal{P} \cup P^*$; otherwise, it will be updated to $\{\mathcal{P} \setminus P'\} \cup \{P_m, P_r\}$.

When the function `BestCandidatePattern` ($\mathbb{CP}$) returns null which indicates that there is no candidate pattern in the current sequence any more, DTS moves on to the next stage. Lines 18–21 correspond to the update stage containing only refinement. Here, DTS updates $\mathcal{P}$ with the best refinement in $\mathbb{CR}$ iteratively.

To illustrate, consider the example in Figure 6, in which the best current candidate pattern is $P_1 = EF$, and there exist two conflicting refinements in $\mathbb{CR}$, where $R^*(D D)^F = [1, F, D D, DF D]$ is selected using `BESTCANDIDATEREFINEMENT` (EF). If $R^*(D D)^F$ has greater compression gain, pattern set $\mathcal{P}$ will be updated to $\mathcal{P}'$; otherwise, pattern $P_1 = EF$ will be added to $\mathcal{P}$, resulting in $\mathcal{P}''$.

The data structures $\mathbb{CP}$ and $\mathbb{CR}$ are maintained dynamically. After each update, we delete invalid elements in them and adjust the patterns that are influenced by the update.

Input: singleton event set Σ and sequence S
Output: candidate pattern set $\mathbb{CP}$

```

(1) function INITIALIZE ( $\mathbb{CP}$ ) ( $\Sigma, S$ )
(2)    $\mathbb{CP} \leftarrow \emptyset$ 
(3)   for  $E \in \Sigma$  do
(4)     create a singleton pattern  $P \leftarrow \langle E \rangle$ 
(5)      $P_{\text{last}} \leftarrow P$ 
(6)     while  $P \leftarrow \text{BESTGROWTH}(P, S, \Sigma) \neq \text{null}$  do
(7)        $P_{\text{last}} \leftarrow P$ 
(8)     end while
(9)     if  $\text{gain}(P_{\text{last}}) > 0$  then
(10)      add  $P_{\text{last}}$  to  $\mathbb{CP}$ 
(11)    end if
(12)  end for
(13)  sort the patterns in  $\mathbb{CP}$  by  $\text{gain}(P)$ 
(14)  return  $\mathbb{CP}$ 
(15) end function

```

ALGORITHM 1: Initialize the candidate pattern set $\mathbb{CP}$.

Input: pattern P , sequence S , singleton event set Σ and candidate refinement map $\mathbb{CR}$

```

(1) function OBTAINREFINEMENTS ( $P, S, \Sigma, \mathbb{CR}$ )
(2)   for  $j = 0 \rightarrow |P|$  do
(3)      $R^*(P)_j^E \leftarrow \text{null}$ 
(4)     for  $E \in \Sigma$  do
(5)        $R(P)_j^E \leftarrow \text{REFINE}(P, S, j, E)$ 
(6)       if  $\text{cmp}(R(P)_j^E, R^*(P)_j^E) > 0$  then
(7)          $R^*(P)_j^E \leftarrow R(P)_j^E$ 
(8)       end if
(9)     end for
(10)    if  $\text{gain}(R^*(P)_j^E) > 0$  then
(11)      add  $R^*(P)_j^E$  to  $\mathbb{CR}.get(E)$ 
(12)    end if
(13)  end for
(14) end function

```

ALGORITHM 2: Obtain the best refinements for pattern P .

5. Experiments

In this section, we conduct extensive experiments to verify the effectiveness of our approach. All experiments were executed on a machine with Intel i7-3770 CPU @3.4 GHz, 24GB of RAM, and Window 10 OS.

5.1. Datasets and Settings

5.1.1. Datasets. We use four different log datasets collected from real systems. The basic statistics of these datasets are summarized in Table 2. The Zookeeper and OpenStack datasets are collected from the well-known loghub data repository [15], while the NameNode and DataNode logs are collected from our own HDFS cluster.

These datasets have different characteristics. The NameNode logs have more event types and more complex system behaviors interleaving together as a NameNode manages multiple DataNodes. The DataNode and

Zookeeper logs have less event types as well as simpler and more regular behaviors. The OpenStack logs contain only behaviors such as creating a project and other simple tasks, thus having the least event types.

For each log dataset, we calculate the ratio of event types which occur less than 50 times. As is shown in Table 2, three out of four datasets contain a high percentage of low-support event types (more than 65%). To discover nontrivial patterns in these datasets effectively, we need to set the support threshold to a low value, which can lead to the pattern explosion problem. We will later show that DTS can discover nontrivial patterns while maintaining a low level of redundancy.

5.1.2. Rival Approaches. We compare our approach against four approaches, GoKrimp [9], SQS [12], CSC [11], and ISM [16]. The first three approaches use the MDL principle [10] to discover patterns, while ISM is a probabilistic machine learning approach. We have not included SeqKrimp [9] in

Input: sequence S
Output: pattern set $\mathcal{P}$

```

(1) function DTS ( $S$ )
(2)    $\Sigma \leftarrow$  all event types in  $S$ ,  $\mathbb{P} \leftarrow \emptyset$ 
(3)    $\mathbb{CP} \leftarrow \text{INITIALIZECP}(\Sigma, S)$ ,  $\mathbb{CR} \leftarrow \text{newMap}$ 
(4)   while true do
(5)      $P^* \leftarrow \text{BESTCANDIDATEPATTERN}(\mathbb{CP})$ 
(6)     if  $P^* = \text{null}$  then
(7)       break
(8)     end if
(9)      $R^*(P')_j^E \leftarrow \text{BESTCONFLICTREFINEMENT}(P, \mathbb{CR})$ 
(10)    if  $R^*(P')_j^E = \text{null}$  or  $\text{cmp}(P^*, R^*(P')_j^E) \geq 0$  then
(11)       $\mathcal{P} \leftarrow \mathcal{P} \cup \{P^*\}$ 
(12)       $\text{UPDATECP}, \text{UPDATECR}, \text{ENCODE}(S, P^*)$ 
(13)    else
(14)       $\mathcal{P} \leftarrow \{\mathcal{P} \setminus P'\} \cup \{P'_m, P'_r\}$ 
(15)       $\text{UPDATECP}, \text{UPDATECR}, \text{ENCODE}(S, \{P'_m, P'_r\})$ 
(16)    end if
(17)  end while
(18)  while  $(R^*(P')_j^E \leftarrow \text{BESTCANDIDATEREFINEMENT}(\mathbb{CR})) \neq \text{null}$  do
(19)     $\mathcal{P} \leftarrow \{\mathcal{P} \setminus P^*\} \cup \{P'_m, P'_r\}$ 
(20)     $\text{UPDATECP}, \text{UPDATECR}, \text{ENCODE}(S, \{P'_m, P'_r\})$ 
(21)  end while
(22)  return  $\mathcal{P}$ 
(23) end function

```

ALGORITHM 3: DTS: discovering patterns from temporal sequences.

Pattern set $\mathcal{P}$	Candidate patterns $\mathbb{CP}$	Candidate refinements $\mathbb{CR}$
$P_1: ABC$ $P_2: DD$ $P_3: MN$	$P_1: EF$ $P_2: HK$	$E: \begin{cases} [3, E, P_1, P'_1] \\ [0, E, P_3, P'_3] \end{cases}$ $F: [1, F, P_2, P'_2]$
Pattern set $\mathcal{P}'$	Pattern set $\mathcal{P}''$	
$P_1: ABC$ $P_2: DD$ $P_3: MN$ $P_4: DFD$	$P_1: ABC$ $P_2: DD$ $P_3: MN$ $P_4: EF$	

FIGURE 6: An example of updating a pattern set.

our experiments because GoKrimp yields similar results and is more efficient than SeqKrimp. For CSC and our approach, we simply use the entire sequence as the input. For other approaches, the sequences are broken into disjoint sequences of size 10. Our algorithms are implemented in Java, while the implementations of other approaches are obtained from the original authors.

5.2. Evaluation of Efficiency. In the first experiment, we compare the efficiencies of all approaches as the length of the log sequences varies on the NameNode and DataNode

datasets. The results are shown in Figure 7. It is worth noting that although the results of all 5 approaches are plotted in the same figures, the time unit for CSC, GoKrimp, and our approach is in seconds, while that for SQS and ISM is in minutes.

As is shown, CSC is the faster than all other approaches because it has a gap constraint which restricts the size of the search space greatly. However, this constraint dictates that CSC cannot find complex patterns effectively, which will be shown in the next experiment. DTS is slower than GoKrimp as GoKrimp applies a dependency test for speedup. Nonetheless, this prevents GoKrimp from discovering low-frequency patterns.

Although DTS takes comparatively longer time, it only needs less than 400 seconds to process a complex sequence with 600,000 events, which is acceptable in real applications. Besides, the running time of DTS increases steadily with the increase of log size, indicating good scalability. On the contrary, SQS and ISM are much slower, with processing times ranging from dozens of minutes to several hours due to their complex iterative computations.

The running times on the Zookeeper and OpenStack datasets are shown in Table 3, which shows similar results to those on the NameNode and DataNode logs. CSC is still the most efficient, while DTS comes next.

5.3. Evaluation of Pattern Quality. In this section, we analyze the quality of the patterns returned by different approaches from three aspects: the compression ratio CR, the event coverage EC, and the redundancy among different patterns.

TABLE 2: Basic statistics of four log datasets.

Log type	# log entries	# event types	% supp(E) ≤ 50
NameNode	600,000	180	75%
DataNode	1,000,000	106	88%
Zookeeper	74,380	85	68%
OpenStack	91,447	36	2%

FIGURE 7: Running time on varying length sequences. (a) NameNode. (b) DataNode.

TABLE 3: Running time on the Zookeeper and OpenStack logs (in seconds).

	DTS	SQS	CSC	GoKrimp	ISM
Zookeeper	19	666	20	45	2040
OpenStack	193	1578	15	231	59

5.3.1. Evaluation of the Compression Ratio. According to the MDL principle, we can encode the sequence with the discovered pattern set. The smaller the description length $L(S, \mathcal{P})$ is, the stronger the expressiveness of the encoding pattern set is. Here, we use compression ratio [9] (denoted as CR) to measure the effectiveness of the encoding. CR is defined as the ratio of the description length of the encoding with only singleton patterns to that with the discovered patterns. The higher the ratio is, the better the expressiveness is. Note that here we only compare the compression ratios for MDL-based approaches.

Figure 8 shows the compression ratios on all four datasets. As indicated, DTS generally achieves the best compression ratio, thanks to our effective encoding scheme as well as the proposed refinement operations and the heuristic strategy based on the “event conflict.” Among the other 3 comparison algorithms, the iterative searching strategy of GoKrimp is too simple to yield expressive results. Therefore, it generally achieves the lowest compression ratio. CSC has a good performance on the DataNode and Zookeeper logs, yielding similar performance to DTS. This can be attributed to the fact that CSC is suitable for processing log data in which the time intervals between adjacent events

are short and constant. On the contrary, for log sequences such as NameNode logs which contain complex patterns, CSC tends to perform poorly. As with SQS, thanks to its complex search strategy, it performs well on more complex data. The problem with SQS is that it generates too many patterns, and thus, the model used to describe the data is too complicated, resulting in greater description length. This can negatively impact the compression ratio of SQS, especially when the patterns in the log data are simple.

It is worth noting that while our approach has generally higher compression ratios, this does not mean that it results in more redundancies, on which we will discuss later.

5.3.2. Evaluation of the Event Coverage. In this section, we evaluate whether the patterns returned by different approaches cover various event types adequately. We use the event coverage [17] (denoted as EC) as the metric, which measures the percentage of event types covered by the discovered patterns. A larger EC indicates better performance.

Figure 9 demonstrates the event coverage of all 5 approaches on the four log datasets. As is shown, except for

FIGURE 8: Compression ratios on different datasets.

FIGURE 9: Event coverage on different datasets.

GoKrimp, all other approaches have event coverage over 80%. By contrast, while GoKrimp achieves an EC of 91% on the OpenStack dataset, it fails to achieve an EC of over 50% on the remaining datasets. This is related to the proportion of rare event types in the log data. As was mentioned before, the search heuristic of GoKrimp is oversimplified, which prevents GoKrimp from discovering patterns with infrequent event types. Therefore, the more infrequent event types contained in the log data, the lower the event coverage of GoKrimp is. By contrast, this factor has little effect on the other five algorithms.

Overall, our DTS achieves the best coverage on the three of the four log datasets. However, it is slightly inferior to SQS and CSC on the OpenStack logs. This is because there are three event types in the OpenStack dataset which occur randomly in the sequence, exhibiting no significant temporal regularity. Therefore, DTS does not consider these event types as part of any pattern, while CSC and SQS greedily include these event types in some patterns. This showcases the advantage of DTS, which takes the temporal regularities of patterns into consideration, reducing the number of invalid patterns made up of random combinations of unrelated events. This can lead to a far less redundant result than CSC and SQS.

5.3.3. Evaluation of the Pattern Redundancy. The goal of this work is to mine high-quality and low-redundancy pattern sets from log sequences. In previous sections, we have

evaluated the pattern qualities using compression rate and event type coverage. We now shift our attention to the evaluation of pattern redundancy. Here, we utilize the following four metrics:

- (1) Average intersequence edit distance (AED): this metric measures the edit similarity among the discovered patterns. Concretely, the AED of a pattern set is calculated as follows: for each pattern P in set $\mathcal{P}$, we calculate the edit distance between P and each of the remaining discovered patterns P' ($P' \in \mathcal{P}$) and normalize the distance with the pattern P 's length. We record the minimum distance between P and all other patterns. AED is obtained by averaging these minimum distances for all patterns in the pattern set. The larger the distance is, the less redundant the pattern set is.
- (2) Average count of supersets (ACS): this metric measures the diversity of the event types contained in the discovered pattern set. Concretely, the ACS of a pattern set is calculated as follows: for each pattern P in $\mathcal{P}$, we denote the event type set contained in P as $\text{Set}(P)$. For each P , we calculate the number of remaining patterns P' ($P' \in \mathcal{P}$) whose event type set $\text{Set}(P')$ is the superset of $\text{Set}(P)$. ACS is obtained by averaging this number for all patterns in the pattern set. For each pattern, the less patterns whose event type set is its superset, the less redundant the pattern is. Therefore, the smaller the ACS is, the more diverse the event types are and the less redundant the pattern set is.
- (3) Number of patterns (No. P): this metric measures the number of discovered patterns.
- (4) Average length of patterns (AL): this metric measures the average length of the discovered patterns.

Table 4 reports the results of four evaluation metrics on different datasets, respectively. As is shown, our approach has a larger intersequence edit distance and a smaller count of supersets on average, which indicate that the patterns generated by our approach have lower redundancy and higher diversity. In terms of the number of patterns, DTS has relatively small No. P values while ensuring adequate coverage of event types and a high level of compression. Overall, the pattern sets discovered by DTS can express the meaningful information of a log sequence with low redundancy.

Among the four rival algorithms, GoKrimp yields relatively good performance, generating the smallest number of patterns with relatively low redundancy. However, as was previously shown, the expressiveness of GoKrimp is generally the worst among all methods. As a matter of fact, the search strategy of GoKrimp acts as a double-edged sword. On the one hand, it helps achieve compact results. On the other hand, it limits GoKrimp's ability to mine meaningful patterns. Overall, this tradeoff is not desirable in practice. As with ISM, it has reasonable AED and ACS values, yet it is inferior when compared against DTS. By contrast, the results of CSC on the Zookeeper and DataNode logs are less redundant, which again manifests the effectiveness of CSC for

TABLE 4: Redundancy metrics on different datasets.

	NameNode				DataNode				Zookeeper				OpenStack			
	AED	ACS	No. P	AL	AED	ACS	No. P	AL	AED	ACS	No. P	AL	AED	ACS	No. P	AL
DTS	0.538	1.423	52	9.846	0.751	0.455	33	6.818	0.446	0.936	47	6.255	0.911	0.625	8	7.250
SQS	0.282	22.758	389	6.419	0.309	15.396	202	4.168	0.280	33.703	239	5.678	0.201	15.761	289	7.467
CSC	0.284	14.591	159	7.881	0.248	3.436	55	8.109	0.363	5.075	107	4.972	0.091	12.269	93	28.548
GoKrimp	0.476	1.569	51	3.039	0.517	1.333	18	2.667	0.472	0.955	44	2.659	0.416	1.417	36	3.250
ISM	0.322	1.986	141	6.901	0.366	1.087	103	6.214	0.357	2.201	159	5.145	0.453	1.273	33	3.333

such type of log data. However, on the NameNode and OpenStack logs, the redundancy of CSC is high. Especially, the values of ACS are more than 10, namely, each combination of event types generates more than 10 patterns on average. As for SQS, it has a high degree of redundancy on every type of log data. This is because the encoding scheme of SQS only considers the support of the patterns and does not fully take into account the temporal regularity. Therefore, any sequential pattern generated by frequent behaviors can result in good compression, which leads to many redundant patterns.

6. Related Work

As an important data source for system management, logs have been widely used in many tasks such as anomaly detection [5, 13], program workflow modeling [1], failure diagnosis [18], and performance monitoring [19]. These works mostly focus on automatic system maintenance and diagnosis through log analysis. In this paper, we aim at discovering meaningful patterns from raw log sequences. The discovered patterns can provide useful information that can support the aforementioned tasks.

Sequential pattern mining was first introduced by Agrawal and Srikant [6]. Since then, various sequential pattern mining algorithms have been proposed, such as PrefixSpan [20], SPAM [21], and BIDE [7]. Traditional pattern mining approaches usually generate a huge number of redundant patterns. This problem is commonly known as pattern explosion. Several restrictions on pattern semantics have been proposed to tackle this problem, such as closed frequent patterns [22] and maximal frequent patterns [23]. However, these measures do not fully resolve the problem. In order to reduce pattern redundancy, modern sequential pattern mining approaches resort to the minimum description length (MDL) principle [10] to select a set of patterns that can best compress the data. KRIMP [24] pioneered the use of MDL in identifying good pattern sets, which mines itemsets that can well describe a transaction database. GoKrimp [9], SQS [12], and CSC [11] extend this methodology to the sequential pattern mining task. The basic idea of these approaches is to cover a sequence database with a set of patterns that can achieve the highest compression. Jianxin Li [25] proposed a parallel approach FCT to efficiently find frequent co-occurring terms in relational data.

However, these existing approaches do not take the temporal regularity of timestamped sequences into consideration. By comparison, our approaches can achieve

better results by utilizing the event relationships in a pattern as well as the temporal regularities among events and are capable of discovering informative patterns with low redundancy. Instead of designing a MDL-based encoding scheme, ISM [16] presents a subsequence interleaving method based on a probabilistic model of the sequence database, which searches for the most compressed set of patterns. However, this approach, along with the aforementioned SQS, is slow according to our experiments due to its exhaustive nature.

7. Conclusions

In this paper, we have proposed a novel approach to discover sequential patterns from log sequences with temporal regularities, which process the log data in a black-box manner and do not need any domain knowledge on the system. Specifically, we have drawn on the MDL principle and formalized an encoding scheme that takes event relationships as well as temporal regularities into consideration. Based on this scheme, we have proposed DTS, an efficient heuristic algorithm, which greedily updates the pattern set. Extensive experiments on real datasets show that the proposed approaches can discover high-quality patterns efficiently.

Data Availability

The NameNode and DataNode datasets are available from the corresponding author upon request. The Zookeeper and OpenStack datasets are collected from the well-known loghub data repository (<https://github.com/logpai/loghub>).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by MIIT Project (Data Management Standards and Verification for Industrial Internet Identifier Resolution).

References

- [1] J. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, "Mining program workflow from interleaved traces," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, DC, USA, July 2010.
- [2] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their

- behavior with CSight,” in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, May 2014.
- [3] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson, “Mining temporal invariants from partially ordered logs,” *Operating Systems Review*, vol. 45, pp. 39–46, 2011.
 - [4] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, “CloudSeer: workflow monitoring of cloud infrastructures via interleaved logs,” in *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS’16*, Atlanta, GA, USA, April 2016.
 - [5] M. Du, F. Li, G. Zheng, and V. Srikumar, “DeepLog: anomaly detection and diagnosis from system logs through deep learning,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications*, New York, NY, USA, 2017.
 - [6] R. Agrawal and R. Srikant, “Mining sequential patterns,” in *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, March 1995.
 - [7] J. Wang and J. Han, “BIDE: efficient mining of frequent closed sequences,” in *Proceedings of the 20th International Conference on Data Engineering*, Boston, MA, USA, March 2004.
 - [8] J. Li, C. Liu, R. Zhou, and W. Wang, “XML keyword search with promising result type recommendations,” *World Wide Web*, vol. 17, no. 1, pp. 127–159, 2014.
 - [9] H. T. Lam, F. Mörchén, D. Fradkin, and T. Calders, “Mining compressing sequential patterns,” *Statistical Analysis and Data Mining*, vol. vol. 7, no. no. 1, pp. 34–52, 2014.
 - [10] P. D. Grünwald, *The Minimum Description Length Principle (Adaptive Computation and Machine Learning)*, The MIT Press, Cambridge, MA, USA, 2007.
 - [11] A. Ibrahim, S. Sastry, and P. S. Sastry, “Discovering compressing serial episodes from event sequences,” *Knowledge and Information Systems*, vol. vol. 47, no. no. 2, pp. 405–432, 2016.
 - [12] N. Tatti and J. Vreeken, “The long and the short of it: summarising event sequences with serial episodes,” in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Beijing, China, August 2012.
 - [13] Q. Fu, J. Lou, Y. Wang, and J. Li, “Execution anomaly detection in distributed systems through unstructured log analysis,” in *Proceedings of the 2019 IEEE International Conference on Data Mining*, Miami, FL, USA, 2009.
 - [14] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: an online log parsing approach with fixed depth tree,” in *Proceedings of the 24th IEEE International Conference on Web Services*, Honolulu, HI, USA, June 2017.
 - [15] S. He, J. Zhu, P. He, and M. R. Lyu, “Loghub: a large collection of system log datasets towards automated log analytics,” 2020, <https://arxiv.org/abs/2008.06448>.
 - [16] J. M. Fowkes and C. A. Sutton, “A subsequence interleaving model for sequential pattern mining,” in *Proceedings of the 2016 ACM Conference on Knowledge Discovery and Data Mining*, San Francisco, CA, USA, August 2016.
 - [17] Y. Yan, L. Cao, S. Madden, and E. A. Rundensteiner, “SWIFT: Swift,” *Proceedings of the VLDB Endowment*, vol. 12, no. 3, pp. 265–277, 2018.
 - [18] S. Zhang, W. Meng, J. Bu et al., “Syslog processing for switch failure diagnosis and prediction in datacenter networks,” in *Proceedings of the IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, Austin, TX, USA, 2017.
 - [19] D. Yuan, S. Park, P. Huang et al., “Be conservative: enhancing failure diagnosis with proactive logging,” in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC’12)*, Hollywood, CA, USA, 2012.
 - [20] J. Pei, J. Han, B. Mortazavi-Asl et al., “PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth,” in *Proceedings of the 2001 International Conference on Data Engineering (ICDE 2001)*, Washington, DC, USA, 2001.
 - [21] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, “Sequential pattern mining using a bitmap representation,” in *Proceedings of the SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, 2002.
 - [22] J. Wang, J. Han, and C. Li, “Frequent closed sequence mining without candidate maintenance,” *IEEE Transactions on Knowledge and Data Engineering*, vol. vol. 19, no. no. 8, pp. 1042–1056, 2007.
 - [23] P. Fournier-Viger, C. Wu, A. Gomariz, and V. S. Tseng, “VMSP: efficient vertical mining of maximal sequential patterns,” in *Proceedings of the 34th Artificial Intelligence*, Cambridge, UK, December 2014.
 - [24] J. Vreeken, M. van Leeuwen, and A. Siebes, “Krimp: mining itemsets that compress,” *Data Mining and Knowledge Discovery*, vol. 23, pp. 169–214, 2011.
 - [25] J. Li, C. Liu, R. Zhou, and J. X. Yu, “Query-driven frequent Co-occurring term computation over relational data using MapReduce,” *The Computer Journal*, vol. 58, no. 3, pp. 497–513, 2015.