LogMine: Fast Pattern Recognition for Log Analytics

Hossein Hamooni
University of New Mexico
1 University Blvd
Albuquerque, NM 87131
hamooni@unm.edu

Biplob Debnath
NEC Laboratories America
4 Independence Way
Princeton, NJ 08540
biplob@nec-labs.com

Jianwu Xu
NEC Laboratories America
4 Independence Way
Princeton, NJ 08540
jianwu@nec-labs.com

Hui Zhang
NEC Laboratories America
4 Independence Way
Princeton, NJ 08540
huizhang@nec-labs.com

Guofei Jiang
NEC Laboratories America
4 Independence Way
Princeton, NJ 08540
gfj@nec-labs.com

Abdullah Mueen
University of New Mexico
1 University Blvd
Albuquerque, NM 87131
mueen@unm.edu

ABSTRACT

Modern engineering incorporates smart technologies in all aspects of our lives. Smart technologies are generating terabytes of log messages every day to report their status. It is crucial to analyze these log messages and present usable information (e.g. patterns) to administrators, so that they can manage and monitor these technologies. Patterns minimally represent large groups of log messages and enable the administrators to do further analysis, such as anomaly detection and event prediction. Although patterns exist commonly in automated log messages, recognizing them in massive set of log messages from heterogeneous sources without any prior information is a significant undertaking. We propose a method, named LogMine, that extracts high quality patterns for a given set of log messages. Our method is fast, memory efficient, accurate, and scalable. LogMine is implemented in map-reduce framework for distributed platforms to process millions of log messages in seconds. LogMine is a robust method that works for heterogeneous log messages generated in a wide variety of systems. Our method exploits algorithmic techniques to minimize the computational overhead based on the fact that log messages are always automatically generated. We evaluate the performance of LogMine on massive sets of log messages generated in industrial applications. LogMine has successfully generated patterns which are as good as the patterns generated by exact and unscalable method, while achieving a 500× speedup. Finally, we describe three applications of the patterns generated by LogMine in monitoring large scale industrial systems.

CCS Concepts

•**Computing methodologies** → **MapReduce algorithms;**
•**Information systems** → *Clustering;*

CIKM'16 , October 24 - 28, 2016, Indianapolis, IN, USA

ISBN 978-1-4503-4073-1/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2983323.2983358

Keywords

Log analysis; Pattern recognition; Map-reduce

1. INTRODUCTION

The Internet of Things (IoT) enables advanced connectivity of computing and embedded devices through internet infrastructure. Although computers and smartphones are the most common devices in IoT, the number of "things" is expected to grow to 50 billion by 2020 [5]. IoT involves machine-to-machine communications (M2M), where it is important to continuously monitor connected machines to detect any anomaly or bug, and resolve them quickly to minimize the downtime. Logging is a commonly used mechanism to record machines' behaviors and various states for maintenance and troubleshooting. An acceptable logging standard is yet to be developed for IoT, most commonly due to the enormous varieties of "things" and their fast evolution over time. Thus, it is extremely challenging to parse and analyze log messages from systems like IoT.

An automated log analyzer must have one component to *recognize* patterns from log messages, and another component to *match* these patterns with the inflow of log messages to identify events and anomalies. Such a log message analyzer must have the following desirable properties:

- **No-supervision:** The pattern recognizer needs to be working from the scratch without any prior knowledge or human supervision. For a new log message format, the pattern recognizer should not require an input from the administrator.
- **Heterogeneity:** There can be log messages generated from different applications and systems. Each system may generate log messages in multiple formats. An automated recognizer must find all formats of the log messages irrespective of their origins.
- **Efficiency:** IoT-like systems generate millions of log messages every day. The log processing should be done so efficiently that the processing rate is always faster than the log generation rate.
- **Scalability:** Pattern recognizer must be able to process massive batches of log messages to maintain a current set of patterns without incurring CPU and memory bottlenecks.

Many companies such as Splunk[11], Sumo Logic[12], Loggly[6], LogEntries[7], etc. offer log analysis tools. Open

1. 2015-07-09 10:22:12,235 INFO action=set root="/"
2. 2015-07-09 12:32:46,806 INFO action=insert user=tom id=201923 record=abf343rf
3. 2015-07-09 14:24:16,247 WARNING action=remove home="/users/david"
4. 2015-07-09 20:09:11,909 INFO action=insert user=david id=455095 record=efrdf4w2
5. 2015-07-09 21:56:01,728 INFO action=set home="/users"
6. 2015-07-09 22:11:56,434 WARNING action=delete user=tom id=201923 record=asepg9e
7. 2015-07-09 22:32:46,657 INFO action=insert user=david id=455095 record=3jnsg67
8. 2015-07-09 22:34:12,724 WARNING action=remove home="/users/tom"

9. date time,number INFO action=insert user=david id=455095 record=XXX
10. date time,number XXX action=XXX user=tom id=201923 record=XXX
11. date time,number INFO action=set XXX=XXX
12. date time,number WARNING action=remove home=XXX

13. date time,number XXX action=XXX user=XXX id=XXX record=XXX
14. date time,number XXX action=XXX XXX=XXX

15. date time,number XXX action=XXX XXX=XXX XXX*=XXX* XXX*=XXX*

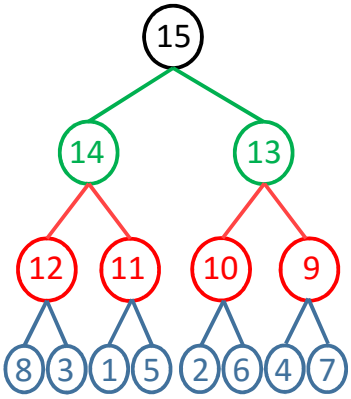

Figure 1: Extracting log patterns for a given set of logs. Hierarchy of patterns gives user the flexibility to choose a level based on his needs.

source packages such as ElasticSearch[2], Graylog[4] and OSSIM[9] have also been developed to analyze logs. Most of these tools and packages use regular expressions (regex) to match with log messages. These tools assume that the administrators know how to work with regex, and there are plenty of tools and libraries that support regex. However, these tools do not have the desirable properties mentioned earlier. By definition, these tools support only supervised matching. Human involvement is clearly non-scalable for heterogeneous and continuously evolving log message formats in systems such as IoT, and it is humanly impossible to parse the sheer number of log entries generated in an hour, let alone days and weeks. On top of that, writing regex rules is long, frustrating, error-prone, and regex rules may conflict with each other especially for IoT-like systems. Even if a set of regex rules is written, the rate of processing log messages can be slow due to overgeneralized regexes.

A recent work on automated pattern recognition has shown a methodology, called *HLAer*, for automatically parsing heterogenous log messages [22]. Although *HLAer* is unsupervised and robust to heterogeneity, it is not efficient and scalable because of massive memory requirement and communication overhead in parallel implementation.

In this paper, we present an *end-to-end* framework, **LogMine**, that addresses all of the discussed problems with the existing tools and packages. LogMine is an unsupervised framework that scans log messages only *once* and, therefore, can quickly process hundreds of millions of log messages with a very *small amount of memory*. LogMine works in iterative manner that generates a hierarchy of patterns (regexes), one level at every iteration. The hierarchy provides flexibility to the users to select the right set of patterns that satisfies their specific needs. We implement a map-reduce version of LogMine to deploy in a massively parallel data processing system, and achieve an impressive 500× speedup over a naive exact method.

The rest of this paper is organized to discuss related work in Section 1, and background in Section 2. We describe our proposed method in Section 3. Section 4 and 5 discuss the experimental findings and case studies on the related problems respectively. Finally, we conclude in Section 6.

2. RELATED WORK AND BACKGROUND

Authors of [29] have proposed a method to cluster the web logs without any need to user-defined parameters. Their method is not scalable to large datasets because the time complexity is $O(n^3)$ where n is the number of the logs. [16] introduces a method to create the search index of a website based on the users' search logs. [25] discusses a preprocessing algorithm that extracts a set of fields such as IP, date, URL, etc. from a given dataset of web logs. Authors of [17] have proposed a method to help website admins by extracting useful information from users' navigation logs.

In [15], the authors have clearly reasoned why map-reduce is the choice for log processing rather than RDBMS. Authors have showed various join processing techniques for log data in map-reduce framework. This work, along with [20], greatly inspired us to attempt clustering on massive log data. In [19], the authors describe a unified logging infrastructure for heterogeneous applications. Our framework is well suited to work on top of both of these infrastructures with minimal modification. In HPC (High Performance Computing), logs have been used to identify failures, and troubleshoot the failures in large scale systems [23]. Such tools majorly focus on categorizing *archived* log messages into sequence of failure events, and use the sequence to identify root cause of a problem.

2.1 Motivating Examples

In Figure 1, we show examples of log patterns that our method generates. Our method clusters the messages into coherent groups, and identifies the most specific log pattern to represent each clusters. In Figure 1, the input log segment has four different clusters. The messages within each cluster can be merged to produce the log patterns shown in red. These four patterns can again be clustered and merged to form two more general patterns (in green). Same process can be repeated till we get to the root of the hierarchy which

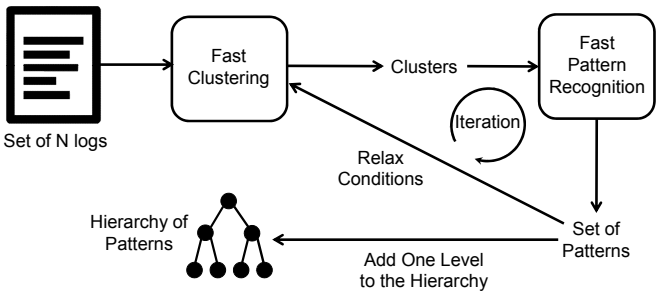

Figure 2: Creating hierarchy of patterns by using a fast clustering algorithm and a fast pattern recognition algorithm.

contains the most general pattern. Note that, there are three basic types of fields in each pattern: **Fixed value**, **Variable** and **Wildcard**. A fixed value field is matched just with one unique value like *www*, *httpd* and *INFO* . A variable field is matched with any value of a certain type such as *number*, *IP address* and *date*. Wildcards are matched with values of all types.

Clearly the most generic pattern is a set of wildcards, and the most specific pattern is a set of fixed attributes. None of these patterns are useful for the administrators. Our method produces a hierarchy of patterns: specific patterns are children of general patterns. Such hierarchy is useful for the system administrators to pick the right level of detail they want to track in the log messages as opposed to write regular expression manually.

2.2 Pattern Recognition Framework

With the above motivation, we design a novel framework for LogMine as shown in the Figure 2. LogMine takes a large batch of logs as input and clusters them very quickly with a restrictive constraints using the clustering module. A pattern is then generated for each cluster by our pattern recognition module. The sets of patterns form the leaf level of the pattern hierarchy. The method will continue to generate clusters with relaxed constraints in subsequent iterations and merge the clusters to form more general patterns, which will constitute a new parent level in the hierarchy. LogMine continues to iterate until the most general pattern has been achieved and/or the hierarchy of patterns is completely formed.

The framework meets all the criteria of a good log analytics tool. It is an unsupervised framework that does not assume any input from the administrator. The framework produces a hierarchy of patterns which is interpretable to the administrator. The framework does not assume any property in sources of the log messages. The recognizer can work on daily or hourly batches if the following challenges can be tackled.

2.3 Challenges

This framework for log pattern recognition is unsupervised and suitable for heterogeneous logs. However, the scalability of the framework depends on the two major parts of the framework: *log clustering* and *pattern recognition*. Standard clustering and recognition methods do not scale well, and it is non-trivial to design scalable versions of the clustering and recognition modules for massive sets of log messages. Since the two modules work in a closed loop, we must speedup both of them to scale the framework for large datasets.

To quantify the significance of the challenge, let us imagine an average website that receives about 2 million visitors a day (much less compared to 500 million tweets a day in Twitter, or 3 billion searches a day in Google). Even if we assume that each visit results in only one log message, 2 million log messages per day is a reasonable number. Clustering such a large number of log messages in only one iteration is extremely time consuming. A standard DBSCAN [28] algorithm takes about two days to process a dataset of this size with state-of-the-art optimization techniques [1]. Similar amount of time would be needed by k-means algorithm although no one would know the best value for the parameter k. Clearly, the framework cannot work on daily batches using standard clustering algorithms and, therefore, we need an optimized clustering algorithm developed for log analysis. A similar argument can be made for the recognition module where a standard multiple sequence alignment operation on a reasonable sized cluster may need more than a day to recognize the patterns.

3. FAST LOG-PATTERN MINING

The key intuition behind our log mining approach is that *logs are automatically generated messages* unlike sentences from a story book. There are some specific lines in an application source code that produce the logs, therefore, all log messages from the same application are generated by a finite set of formats. Standard clustering and merging methods do not consider any dependency among the objects in the dataset. In logs, the dependency among the messages is natural, and as we show in this paper, the dependency is useful to speedup the clustering and the merging process.

3.1 Tokenization and Type Detection

We assume all the logs are stored in a text file and each line contains a log message. We do a simple pre-processing on each log. We tokenize every log message by using white-space separation. We then detect a set of pre-defined types such as date, time, IP and number, and replace the real value of each field with the name of the field. For instance, we replace *2015-07-09* with *date*, or *192.168.10.15* with *IP*. This set of pre-defined types can be configured by the user based on his interest in the content over the type of a field. Figure 3 shows an example of tokenization and type replacements in a log message. Tokenization and type detection is embedded in the clustering algorithm without adding any overhead due to the one-pass nature of the clustering algorithm described in the next section. Although this step is not mandatory, we use it to make the similarity between two logs meaningful. If no type detection is done, two logs generated by the same pattern can have a low similarity, just because they have different values for the same field. Therefore, we may end up generating huge number of unnecessary patterns if we do not tokenize.

3.2 Fast and Memory Efficient Clustering

Our clustering algorithm is simply a one-pass version of the friends-of-friend clustering for the log messages. Our algorithm exploits several optimization techniques to improve the clustering performance.

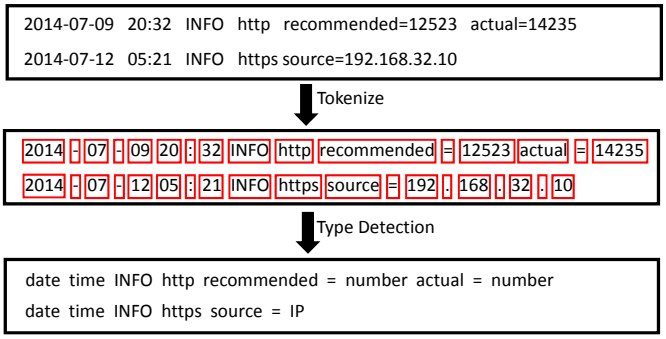

Figure 3: Two examples of how pre-processing works on the input log messages.

3.2.1 Distance Function

We first define the distance between two log messages by the following equations:

$$Dist(P,Q) = 1 - \sum_{i=1}^{Min(len(P),len(Q))} \frac{Score(P_i,Q_i)}{Max(len(P),len(Q))}$$

$$Score(x,y) = \begin{cases} k_1 & \text{if x=y} \\ 0 & \text{otherwise} \end{cases}$$

P_i is the ith field of log P, and $len(P)$ is the number of fields of log P. k_1 is a tunable parameters. We set $k_1 = 1$ in our default log distance function, but this parameter can be changed to put more or less weight on the matched fields in two log messages.

Since we want to cluster patterns in the subsequent iterations of our framework, we also need a distance function for patterns. The distance between two patterns is defined very similarly as the distance between two log messages, just with a new score function.

$$Score(x,y) = \begin{cases} k_1 & \text{if x=y and both are fixed value} \\ k_2 & \text{if x=y and both are variable} \\ 0 & \text{otherwise} \end{cases}$$

We again set $k_1 = k_2 = 1$ in our default pattern distance function. Our distance function is non-negative, symmetric, reflexive, and it satisfies the triangular inequality. Therefore, it is a metric. Log messages generated by the same format have a very small distance (zero in most cases), and log messages generated by different formats have larger distances. This is a desirable property for fast and memory-efficient log clustering algorithm. In the high dimensional space, the log messages form completely separated and highly dense regions. Therefore, finding the clusters using the above distance function is a straightforward task.

3.2.2 Finding Clusters

In this section, we explain how to find the dense clusters out of the input logs in a sequential fashion. The same approach will also be used when we create the hierarchy of log patterns by several iterations of clustering. First, we define an internal parameter named *MaxDist*, which represents the maximum distance between any log entry/message in a cluster and the cluster representative. Therefore, the maximum distance between any two logs in a cluster is $2\times MaxDist$. We start from the first log message and process all the log messages one by one until we reach the last message. Each cluster has a representative log message, which is also the first member of the cluster. For any new log message, we insert the message in one of the existing clusters only if the distance between the log and the representative is less than the *MaxDist*. Otherwise, when the message is not similar to any representative, we create a new cluster and put the log message as the representative of that new cluster.

The above process can be implemented in a very small memory footprint. We need to keep just one representative log for each cluster in the memory, and output the subsequent log messages without saving in the memory. This allows our algorithm to process massive number of logs with a small amount of memory. In fact, the memory usage of our clustering algorithm is O(number of clusters). Ignoring large number of log messages when deciding about cluster membership and using only one representative log message do not reduce the quality of the clusters. The major reason is that all the log messages in any given cluster are almost identical because they are most likely generated by the same code segment of the same application. Therefore, the above one-pass clustering algorithm with a very small *MaxDist* in the beginning can generate highly dense (i.e., consistent) clusters of log messages, where keeping one representative message is both sufficient and efficient.

We finally have a set of dense clusters with one representative each. As mentioned before, this algorithm is also used to cluster and merge patterns. In case of clustering the patterns, unlike the above approach, we keep all the patterns in each cluster because we will use them in the pattern recognition component. In most systems, the set of patterns generated after the first iteration fits in the memory. The speed and efficiency of this algorithms comes from the fact that the number of dense clusters does not scale with the number of log messages, because it is not possible for an application to generate huge number of unique patterns. In other words, finding a million unique patterns is impossible even in a dataset of hundreds of millions of log messages.

The one-pass clustering algorithm has a strong dependency on the order of the messages, which is typically the temporal order. The pathological worst case happens when one message from every pattern in every cluster appears very early in the log, and all of the remaining messages will have to be compared with all of the unique representatives. In practice, log messages from the same application show temporal co-location which makes them more favorable for the clustering algorithm.

3.2.3 Early Abandoning Technique

Early abandoning is a useful technique to speedup similarity search under Euclidean distance. Some of the initial mentions of early abandoning were in [18][13]. It has been extensively used later by many researchers for problems such as time series motif discovery [21], and similarity search under dynamic time warping (DTW) [24]. We adopt the technique for log analytics.

When comparing a new log message with a cluster representative, if the distance is smaller than *MaxDist*, we can add the new log to that cluster. Since the distance between two logs is calculated in one scan of the logs by summing up only non-negative numbers, early abandoning techniques can be applied. As we compare two logs field by field, we may discover that the accumulated distance has already exceeded the threshold, *MaxDist*, even though many of the fields are yet to be compared. In this case, we don't need to

continue calculating the distance completely, because we are certain that these two logs are not in *MaxDist* radius of each other. Since the number of fields in a log can be large, this technique helps us skip a significant amount of calculation, especially when *MaxDist* is small.

3.2.4 Scaling via Map-Reduce Implementation

We mentioned earlier that the memory usage of our one-pass clustering algorithm is O(number of clusters). The one-pass clustering algorithm is very amenable to parallel execution via map-reduce approach. For each log in our dataset, we create a key-value pair. The key is a fixed number (in our case 1), and the value is a singleton list containing the given log. We also add the length based index to the value of each tuple. In the reduce function, we can merge every pair of lists. Specifically, we always keep the bigger list as the base list, and update this base list by adding all elements of the smaller list to it (if needed). This makes the merging process faster. While adding the elements of the smaller list to the base set, we add only the elements which do not have any similar representative in the base set. If a very close representative already exists in the base list, we ignore the log. We also update the length based index of the base list meanwhile. Finally, the base list will be the result of the merging of two given lists. The pseudo code of the reduce function can be found in Algorithm 1.

Algorithm 1 Reduce

Input: Two tuples $A = (1, List_1), B = (1, List_2)$
Output: A tuple
if $size(List_1) >= size(List_2)$ **then**
  $Base\_list \leftarrow List_1$
  $Small\_list \leftarrow List_2$
else if $size(List_1) < size(List_2)$ **then**
  $Base\_list \leftarrow List_2$
  $Small\_list \leftarrow List_1$
for $i = 1, \ldots, size(Small\_list)$ **do**
  Found=False
  for $j = 1, \ldots, size(Base\_list)$ **do**
    if $d(Small\_list(i), Base\_list(j)) \leq MaxDist$ **then**
      Found=True
      break
  if $\neg$ Found **then**
    Append $Small\_list(i)$ in the $Base\_list$
return (1,$Base\_list$)

Since we use the same key for all the logs, we will get one tuple as the final output which contains all the log representative (dense clusters). As we need to create a key-value tuple for each log, the memory usage of the map-reduce implementation is no longer O(number of dense clusters), in fact it is O(number of log entries). This is not a problem because each worker in map-reduce platform loads a chunk of the logs. Even if a chunk of the data does not fit in memory, new map-reduce frameworks like *Spark*[30] can handle that with a small overhead. We compare the running time of both sequential and parallel implementations in Section 4.

3.3 Log Pattern Recognition

After we cluster the logs, we need to find a pattern for each cluster. Since we keep one representative for each dense cluster in the first round, the representative itself is the pattern of its cluster, but in the subsequent rounds, after we cluster the patterns, we need an algorithm that can generate one pattern for a set of logs/patterns in a cluster. We start with the pattern generation process for a pair of patterns and then generalize to a set of patterns.

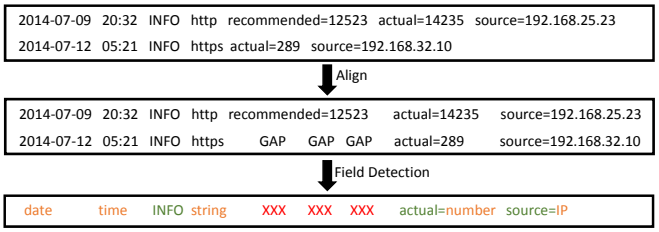

Figure 4: An example of how Algorithm 2 works.

3.3.1 Pattern Generation from Pairs

Irrespective of the pattern recognition algorithm, we always need to merge two logs at some point in the algorithm. Therefore, we shortly discuss our *Merge* algorithm here. Given two logs to be merged, we first need to find their best alignment. The best alignment of two logs is the one that generates the minimum number of wildcards and variables after merging. In the alignment process, some gaps may be inserted between the fields of each log. The alignment algorithm ensures that the length of two logs are equal after inserting the gaps. Once we have two logs with the same length, we process them field by field and generate the output. An example is shown in Figure 4. Note that the align step introduces gaps in the second message. The field detection step requires a straightforward scan of the two logs. A detailed pseudocode can be found in Algorithm 2. There are different algorithms for aligning two sequences. We use *Smith-Waterman* algorithm which can align two sequences of length l_1 and l_2 in $O(l_1.l_2)$ time steps [26]. Therefore, the time complexity of our *Merge* function is also $O(l_1.l_2)$. We use the same score function as in [22] for the Smith-Waterman algorithm.

Algorithm 2 Merge

Input: Two logs (Log_a, Log_b)
Output: A merged log
$Log_a^{'}, Log_b^{'} \leftarrow Align(Log_a, Log_b)$
for i , $i = 2, 3, \ldots, |Log_a^{'}|$ **do**
  $x \leftarrow Field_i(Log_a^{'})$ and $y \leftarrow Field_i(Log_b^{'})$
  if $x = y$ **then**
    $Field_i(Log_{new}) \leftarrow x$
  else if $Type(x) = Type(y)$ **then**
    $Field_i(Log_{new}) \leftarrow Variable_{Type(x)}$
  else
    $Field_i(Log_{new}) \leftarrow Wildcard$
return Log_{new}

3.3.2 Sequential Pattern Generation

To generate the pattern for a set of patterns, we start from the first log message, merge it with the second log, then merge the result with the third log and we go on until we get to the last one. Clearly, the success of this approach largely depends on the ordering of the patterns in the set.

However, as described before, the logs inside each of the dense clusters are almost identical. This is why, in practice, the merge ordering does not associate with the quality of the final pattern. In other words, we will get the same results if we do the merging in reverse or any arbitrary order. If the logs to be merged are not similar, sequential merging may end up producing a pattern with many wildcards which is not desirable. There exists techniques to find the optimal merge ordering for a set of patterns. We provide detailed experiments in the Section 4 to empirically show that sequential merging does not lose quality.

3.3.3 Scaling via Map-Reduce Implementation

As discussed in Section 3.3.2, the order of merging the logs in a cluster to create the final pattern has no effect on the output. Such sequential pattern generation can be parallelize very easily. An efficient way to implement sequential merging is using map-reduce framework. This framework can be useful whenever the order of the operation does not matter, and that is true for our case. Since the pattern recognition is done after clustering the logs, we know the exact cluster for each log. In the map function, we create a key-value pair for each log. The key is the cluster number of the log and the value is the log itself. The map-reduce framework will reduce all the key-value pairs with the same key. In the reduce function, two logs from the same cluster are merged. The final output of the reduce phase is one pattern for each cluster which is exactly what we want. If we ignore the map-reduce framework overhead, in a full parallel running of this algorithm on m machines, its time complexity is $O(\frac{n}{m}.l^2)$, where n is the number of the logs, and l is the average number of fields in each log.

3.4 Hierarchy of Patterns

In Sections 3.2.2 and 3.3, we explain how to find dense clusters of logs, and how to find one pattern that covers all the log messages in each cluster. These two modules constitute an iteration in our pattern recognition framework. We also motivate that one set of patterns generated in one of the iterations can be too specific or general, and may not satisfy the administrator. In contrast, a hierarchy of patterns can provide an holistic view of the log messages, and the administrator can pick a level with the right specificity in the hierarchy to monitor for anomalies.

In order to create the hierarchy, we use both clustering and pattern recognition algorithms iteratively as shown in Figure 2, and produce the hierarchy in bottom-up manner. In the first iteration, we run the clustering algorithm with a very small *MaxDist* on the given set of logs. This is our most restrictive clustering condition. The output of the clustering is a (possibly large) set of dense clusters each of which has a representative log. The representative log is trivially assigned as the pattern for a dense cluster without calling the pattern recognition module. We treat these patterns as the leaves (lowest level) of the pattern hierarchy. To generate the other levels of the hierarchy, we increase the *MaxDist* parameter of the clustering algorithm by a factor of α ($MaxDist_{new} = \alpha MaxDist_{old}$) and run the clustering algorithm on the generated patterns. In other words, we run a more relaxed version of the clustering algorithm on the patterns which will produce new clusters. We then run the pattern recognition module on all the patterns that are clustered together to find more general patterns. These set of new patterns will be added to the hierarchy as a new level. In each iteration of this method, a new level is added to the hierarchy. As we go higher in the hierarchy, we add less number of patterns, which are more general than the patterns in the lower levels. This structure gives us the flexibility to choose whatever level of the hierarchy as the desired set of patterns.

3.4.1 Hybrid Pattern Recognition

When we explain our sequential pattern recognition is Section 3.3.2, we assume that the logs/patterns inside a cluster are very close together. In order to generate the hierarchy of patterns, we start from the leaf level which has the most specific patterns, and the clusters are also dense. As we go up in the hierarchy and merge patterns, the clusters become less dense. Since the *MaxDist* parameter has been relaxed, we allow patterns with larger distances to group together. This creates a chance of being incorrect at the higher levels of the hierarchy if we use the sequential pattern recognition algorithm. Fortunately, the number of patterns inside each cluster at the higher levels of the hierarchy is much less than the lower levels, and we can use a selective merge order, instead of the sequential order, found by the classic UPGMA (Unweighted Pair Group Method with Arithmetic Mean) method. UPGMA, which is simply the hierarchical clustering with average linkage [27], is optimal when the clusters are spherical in shape in the high dimensional space, and we conjecture that this is asymptotically true for log patterns when clusters contain large number of messages.

Our final pattern recognition algorithm is a hybrid of UPGMA, the sequential recognition and the map-reduce implementation. Algorithm 3 shows how We pick the best choice for each cluster of logs. th_1 and th_2 are the thresholds for density and the number of patterns in a cluster respectively.

Algorithm 3 HybridPatternRecognition

```
Input: A cluster of logs (Logs)
Output: A pattern
if density(Logs) ≤ th1 then
  pattern ← UPGMA(Logs)
else
  if Size(Logs) ≤ th2 then
    pattern ← SequentialPatternGeneration(Logs)
  else
    pattern ← MapReducePatternGeneration(Logs)
return pattern
```

3.4.2 Cost of a Level

Given a hierarchy of patterns for a set of logs, the user may be interested in a level with specific properties. Some users may prefer to get the minimum number of patterns while the others may be interested to get very specific patterns and not care about the number of patterns. There are many different criteria one can say a level is satisfying or not. We introduce an intuitive cost function to pick the best descriptive level of the hierarchy. We also propose a cost function that suits our datasets. This cost function can easily serve as a general template to calculate the cost of a level of the hierarchy.

$$Cost = \sum_{i=1}^{\#\text{ of clusters}} Size_i \times (a_1 WC_i + a_2 Var_i + a_3 FV_i)$$

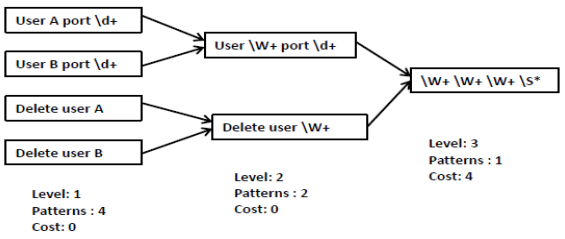

Figure 5: Pattern selection illustration with $a1 = 1$, $a2 = 0$, and $a3 = 0$ in the cost function.

where $Size_i$ is the number of logs in cluster i and WC_i, Var_i and FV_i are the number of wildcards, variable fields and fixed value fields in the pattern of cluster i respectively. a_1, a_2 and a_3 are tunable parameters that can be set in such a way that satisfies user's requirements.

If a user has no preference, we can set $a1 = 1$, $a2 = 0$, and $a3 = 0$ in the cost function, and select the level having no wildcards with minimum number of patterns as the final set of patterns. For example, in Figure 5 we find that Level 2 generates two patterns with no wildcards, so we select these two patterns from Level 2 as the final set of patterns. In our experiments, we assume that a user will not provide any preferences. A user can also provide preferences by specifying the maximum number of expected patterns. For example, a user may want to generate at most 4 patterns. In this case, we select two patterns from the Level 2 in Figure 5 because it will generate minimum number of wildcards while not exceeding the user given maximum of 4 patterns.

4. EVALUATION

We describe the baseline method that we compare LogMine against first. We then discuss our datasets and provide detailed experimental results.

4.1 HLAer: The Baseline

We pick the method *HLAer* [22] as a baseline algorithm, that is similar to our method in being unsupervised and supporting heterogeneous logs. *HLAer* finds very good sets of patterns, if not the optimal patterns, under reasonable assumptions. *HLAer* uses a highly accurate and robust clustering algorithm, OPTICS (Ordering Points To Identify the Clustering Structure), and the average linkage technique (UPGMA) instead of sequential merging for pattern recognition. We describe these two modules next.

4.1.1 Clustering using OPTICS

OPTICS (Ordering Points To Identify the Clustering Structure) is a famous hierarchical density-based clustering algorithm [14] used in *HLAer*. In OPTICS, a priority queue of the objects (e.g. using an indexed heap) is generated [28]. The priority queue can be used to cluster the objects at many different levels without redoing the bulk of the computation. OPTICS has two parameters: ϵ and *MinPts*. It ensures that the final clusters have at least *MinPts* objects, and the maximum distance between any two objects in a cluster is less than or equal to ϵ.

OPTICS is an expensive clustering algorithm for large datasets because it needs to calculate the *MinPts*-nearest neighbors for each object, which requires $O(n^2)$ pair-wise distance calculations for n objects. Typical improvement strategies include parallel computation and indexing techniques. However, these techniques become invalid in the iterative framework and for near-online batch processing. One may also think of pre-computing the pairwise distances, which requires loading all the computed distances in memory in $O(n^2)$ space for random accesses during clustering. Irrespective of non-scalability, OPTICS is a good baseline to compare accuracy with. The algorithm can find clusters of arbitrary shapes and densities in the high dimensional space. The algorithm is easy to reconfigure without recalculating the pair-wise distances.

Table 1: A summary of all datasets.

Dataset	# Logs	# Fields	Availability
D1	2,000	65	proprietary
D2	8,028	200	proprietary
D3	10,000,000	90	proprietary
D4	10,000	37	public
D5	10,000	45	public
D6	10,000	25	public

4.1.2 Log pattern recognition via UPGMA

Once the *HLAer* finds all the clusters, it needs to find a pattern for each cluster that covers all the log entries in it. *HLAer* considers a log entry as a sequence of fields. It finds the best way to align multiple log entries together based on a cost function. After alignment, the algorithm merges each field by selecting a representative field to output. Unweighted Pair Group Method with Arithmetic Mean (UPGMA) is one of the most commonly used multiple sequence alignment (MSA) algorithms. It is a simple bottom-up hierarchical clustering algorithm [27] which can produce a tree of input objects on the basis of their pairwise similarities. *HLAer* runs UPGMA on the set of logs and finds the best order of merging the log entries. The *Merge* function that *HLAer* uses is identical to the one in Section 3.3.

If there are n log entries with l fields (on average), the time complexity of running UPGMA is $O(n^2l^2)$ which takes 300 years for 10 million logs. However, UPGMA can be used for smaller number of log entries, and generate valuable ground truth to evaluate the performance of our pattern recognition algorithms.

4.2 Datasets

For evaluation, we use six different datasets as summarized in Table 1. **D1** and **D2**: Small proprietary datasets. **D3**: An industrial proprietary dataset of size 10,000,000 generated by an application during 30 days. The size of D3 on disk is 10 GB. **D4** and **D5**: Random log entries from two traces which contain a day's worth of all HTTP requests to the EPA webserver located at Research Triangle Park, North Carolina [3], and the San Diego Supercomputer Center in San Diego, California [10] respectively. **D6**: A synthetic dataset generated from 10 pre-defined log patterns. We fix the type of each field in the patterns, and generate random values for that field. This dataset is used as a ground truth to evaluate the accuracy of our method.

We set $MaxDist = 0.01$ and $\alpha = 1.3$ in all the experiments unless otherwise stated. Since *HLAer* takes ϵ and *MinPts* as the input parameters, an expert set them for each dataset. As *HLAer* is a single CPU algorithm, for fair com-

parison, we run the sequential version of our algorithm on a single machine for all experiments unless otherwise stated.

4.3 Accuracy

Since our algorithm is made up of two main components, we test the accuracy of each component separately.

4.3.1 Accuracy of Clustering

We use the clusters generated by OPTICS as a baseline. We define an *agreement* metric to calculate how close the output of our fast clustering algorithm is to the output of OPTICS. Given a set of n log entries $S = \{l_1, l_2, \ldots, l_n\}$, we run OPTICS to get the set of clusters $X = \{X_1, X_2, \ldots, X_r\}$ and we run our clustering algorithm to get the set of clusters $Y = \{Y_1, Y_2, \ldots, Y_s\}$. The *agreement* score is $\frac{a}{b}$, where a is the number of pairs of logs in S that are in the same set in Y and in the same set in X, and b is the number of pairs of logs in S that are in the same set in Y. This metric takes a value between 0 (the worst) and 1 (the best). Note that splitting a cluster of OPTICS into multiple clusters does not harm us because it leads to more accurate patterns and we can merge the sub-clusters in higher levels of the hierarchy. On the other hand, having a cluster which has log entries from multiple OPTICS's clusters can generate a meaningless pattern.

As shown in Table 2, in all datasets except D2, we capture the OPTICS's clusters. The problem with D2 is that the log entries do not have a clear underlying structure. Logs in D2 have many strings and commas, and they are very similar to each other. In addition, OPTICS throws away 38% of the entries as outliers because they do not fall in a cluster with at least MinPts entries. Therefore, a set of separable clusters may not exist in this dataset failing both LogMine and OPTICS.

4.3.2 Accuracy of Pattern Recognition

As discussed in Section 4.1.2, UPGMA finds the best order to merge the log entries, and it produces the best possible pattern for a given cluster. We use the results of UPGMA as a ground truth to evaluate the accuracy of our pattern recognition algorithm. We cluster each dataset by both OPTICS and our clustering algorithm, and then give each cluster to both UPGMA and our pattern recognition algorithm to produce one pattern. We compare the patterns generated by the two algorithms, field by field. The accuracy of a given pattern compared to the ground truth is simply the number of matched fields over the number of all fields. The accuracy of pattern recognition for each dataset is:

$$\text{Total Accuracy} = \sum_{i=1}^{\#\text{ of clusters}} (Acc_i \times Size_i) \div \sum_{i=1}^{\#\text{ of clusters}} Size_i$$

where Acc_i is the accuracy of pattern recognition on cluster i and $Size_i$ is the number of log entries in cluster i. As Table 2 shows we can get almost same patterns as UPGMA except in D2. Since the quality of clustering on D2 is low, the logs inside each cluster are not very similar, and the order of merging can change the structure of the final pattern. The fact that our patterns for D2 are 73% similar to UPGMA patterns does not mean that ours are not accurate, because the patterns generated by UPGMA are also low quality. All the other results support the fact that the order of merging the logs has no effect on the final generated pattern in a cluster with similar logs.

Table 2: The accuracy of pattern recognition and the agreement score for different datasets. We do not report these measurements on D3 because HLAer cannot handle it.

	Accuracy	**Agreement Score**	**HLAer Memory (MB)**	**LogMine Memory (MB)**
D1	100%	86%	32	2
D2	73%	48%	520	11
D4	96%	95%	801	15
D5	98%	100%	802	14
D6	100%	100%	795	13

4.4 Memory Usage

HLAer calculates all the logs pairwise distances and use them while running OPTICS. This needs either $O(n^2)$ memory space or multiple disk accesses in case all the distances are stored back in the disk for n logs. Conversely, LogMine is very memory efficient with a space complexity of O(number of clusters) in sequential fashion. We run our sequential implementation and measure the amount of memory used by both *HLAer* and LogMine. Results are shown in Table 2.

4.5 Running Time

HLAer has maximum processing capacity of 10,000 log entries because of the quadratic memory requirement. For the rest of the datasets, results are shown in Figure 6(right). **LogMine is up to** 500× **faster than *HLAer***. It takes 1,524 seconds for LogMine to cluster the logs, and find all the patterns in dataset D3.

It is worth to mention that LogMine has an advantage over *HLAer* in terms of running time of pattern recognition. Pattern recognition component of LogMine has a fixed running time for datasets of the same size, because it just scans the data once no matter how many cluster exists in the dataset. In contrast, *HLAer* depends on domain and data properties

4.6 Map-Reduce vs. Sequential

We discussed the way we find the dense clusters in Section 3.2.2 both in sequential and map-reduce fashion. In this experiment we compare them. We generate synthetic data by changing number of log entries (10 million default) and number of patterns (1500 default). We change the number of map-reduce workers (8 default) to understand scalability. Each worker has 1 GB of memory and a single-core CPU.

As shown in Figure 6(left), the execution time of the map-reduce implementation grows slowly compared to the growth of the sequential implementation. Map-reduce implementation reaches up to 5× speed-up by using 8 workers compared to the sequential implementation. Note that we have a fixed number of patterns in this experiment. Our map-reduce implementation can handle millions of logs in few minutes, because the number of patterns does not grow at the same rate as the number of logs grows in real world applications. Figure 6(second-left) shows that with increasing number of patterns, the execution time of both sequential and map-reduce implementation consistently grows. In Figure 6(second-right), we show that doubling the number of workers reduces the running time by 40%. The reason is that as we add more workers, the algorithm needs to perform more merges (see Algorithm 1), and this adds more overhead to the algorithm.

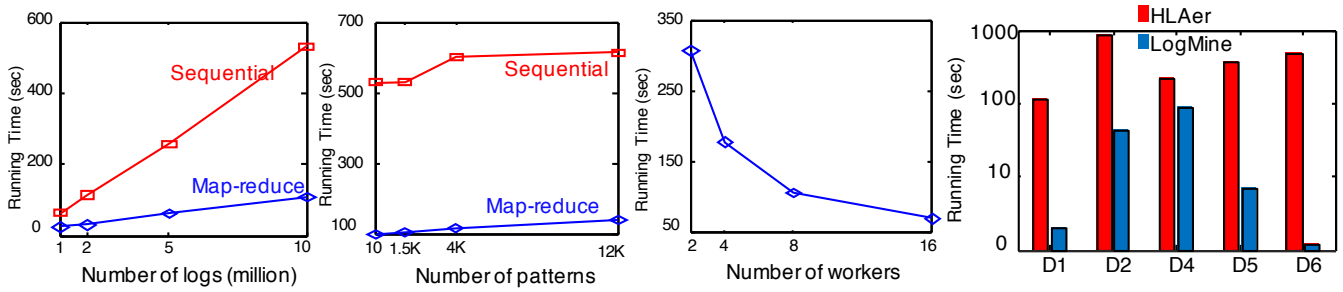

Figure 6: Comparison of running times of the sequential and the map-reduce clustering. (right) Comparing the running time of LogMine with *HLAer*. The y-axis is in log scale.

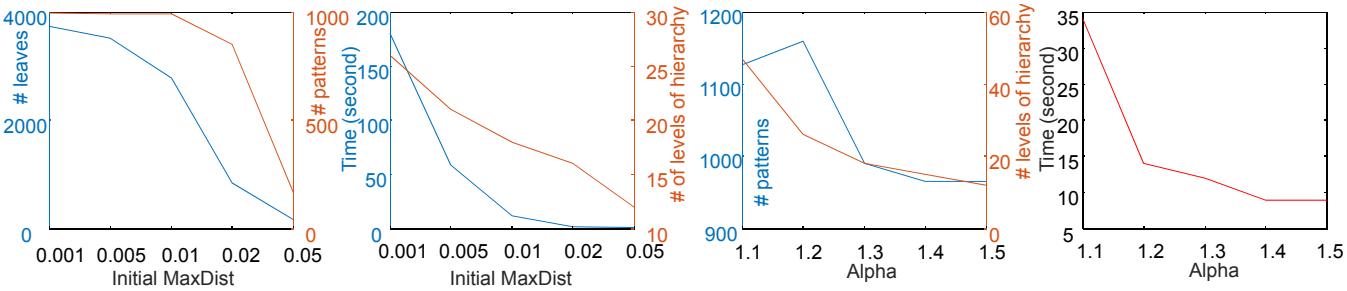

Figure 7: Sensitivity of the algorithm to the parameters $MaxDist_0$ and α.

4.7 Parameter Sensitivity

We have two parameters in our algorithm, and both of them are used in the clustering phase. The first one is the $MaxDist_0$ which is the value we use to run the clustering algorithm at the lowest level of the hierarchy. After we find the leaves, we relax the clustering algorithm condition by increasing the $MaxDist$ by a factor of α. We run an experiment 10 times on a dataset of 10,000 log entries picked randomly from dataset D3, and report the average of different measurements change by increasing $MaxDist_0$(0.01 default) and α (1.3 default). Results are shown in Figure 7. We make the following observations:

• As we increase $MaxDist_0$ we find fewer number of leaves in shorter time. Since the leaves are the basis of our hierarchy, we don't want to lose many of them due to a large $MaxDist_0$. On the other hand, small $MaxDist_0$ usually (not always) yields to longer running time. Although the best value for $MaxDist_0$ is to some extent depends on the dataset, we recommend to set it to 0.01. • As the $MaxDist_0$ grows, we may end up extracting fewer number of patterns, but the plot shows that the algorithm can capture almost the same set of patterns even if we change $MaxDist_0$ in a wide range [0.001,0.02]. Thus, our algorithm is not very sensitive to this parameter in terms of the final set of patterns. • Obviously if we pick smaller values for $MaxDist_0$ and α, the final hierarchy will have more levels and we have more options to choose a satisfactory level in the hierarchy. However, it takes longer to produce such a hierarchy. • Number of patterns does not change drastically with changes in α. We vary this parameter from 1.1 to 1.5 and the number of final patterns stays within the range [966,1127]. We find $\alpha = 1.3$ is the best performing value.

5. CASE STUDY

We use LogMine to analyze logs collected from the OpenStack framework, which an open-source platform for cloud computing [8]. We have collected logs from the execution of `nova-boot` commands to demonstrate how LogMine can be used to build a log analytics solution. LogMine enable us to reveal various insights on logs that helps us to quickly diagnose the cause of failures.

Dataset: We have collected logs generated by 200 successful execution of the nova-boot commands. This is our training dataset. Using LogMine we have found 28 patterns which can correctly identify all training logs. These patterns serve as a basis for analyzing OpenStack logs. Next, we have collected logs from six failed executions (i.e., abnormal) of nova-boot commands. This is our testing dataset.

Detecting New Logs: To identify the cause of a failure, we need to detect logs which are not seen during the successful (i.e., normal) executions of the nova-boot commands. We use the 28 patterns generated by LogMine to detect those unseen logs. If a log in the testing dataset does not match with any of the 28 patterns, then we conclude that it is a new log. Using LogMine we have correctly identify those new logs, and report them to the system administrators for further analysis. Typically, administrators run adhoc keyword-based search on these new logs using their domain knowledge. In this case, searching few keywords, they have identified a subset of new logs, which help them to quickly localize the cause of the failed executions.

Detecting Logs with New Content: To analyze a failure, we are interested to find out whether or not there is any content-wise anomaly among the failure logs. To this end, we have structured 28 patterns generated by LogMine into various fields, and built a content-profile map for each field using the training dataset. During testing, if an incoming log matches with any of the 28 patterns, we identify its fields from the content. Now, if the content of any field value is not present in our training content-profile map, then we report corresponding log to the system administrators for detailed analysis. Using this content analysis, they have correctly

identified new contents in the testing logs, which help them to diagnose failure scenarios quickly.

Detecting Logs with Abnormal Execution Sequence: The execution of OpenStack could result abnormal log pattern sequence if any failure happens. Therefore, we can detect system failure by discovery of anomalous log sequences. In order to achieve this functionality, we built log sequence order for any pair of 28 patterns and modeled their statistics such as the maximal elapse time, maximal concurrency of log pattern during training stage. During testing, we detect if the incoming log violates any of the following rules: log sequence reversal, exceeding the maximal elapse time or concurrency number, or missing the matching log for the pair. System administrators find these violations very useful to debug failures.

Detecting Log Rate Fluctuations: In order to analyze logs, it is helpful to find out whether or not there is any fluctuation in the log rates compared to the normal working scenarios. To detect fluctuations, we keep track of the range (i.e., minimum and maximum counts) that we have observed in a fixed interval of the training dataset for all 28 patterns generated by LogMine. During testing, if the matched logs count of any pattern falls out its training range in an interval, we report corresponding time range and pattern information to the system administrators for the further analysis, and they find it very useful to diagnose failure scenarios.

6. CONCLUSION

We have proposed an end-to-end framework, LogMine, to identify patterns in massive heterogeneous logs. LogMine is the first such framework that is ① unsupervised, ② scalable and ③ robust to heterogeneity. LogMine can process millions of logs in a matter of seconds on a distributed platform. It is a one-pass framework with very low memory footprint, which is useful to scale the framework up to hundreds of millions of logs.

7. REFERENCES

[1] Benchmarking for DBSCAN and OPTICS. http://elki.dbs.ifi.lmu.de/wiki/Benchmarking.

[2] Elasticsearch: Store, Search, and Analyze. https://www.elastic.co/guide/index.html.

[3] EPA dataset. http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html.

[4] GrayLog. https://www.graylog.org.

[5] Internet of Things (IoT). http://www.cisco.com/web/solutions/trends/iot/overview.html.

[6] Log Management Explained. https://www.loggly.com/log-management-explained/.

[7] LogEntries. https://logentries.com/doc/.

[8] OpenStack. https://en.wikipedia.org/wiki/OpenStack.

[9] OSSIM (Open Source Security Information Management). https://en.wikipedia.org/wiki/OSSIM.

[10] SDSC dataset. http://ita.ee.lbl.gov/html/contrib/SDSC-HTTP.html.

[11] Splunk. http://www.splunk.com/en_us/solutions/solution-areas/internet-of-things.html.

[12] Sumo Logic. https://www.sumologic.com/.

[13] E. and S. Kasetty. On the need for time series data mining benchmarks: a survey and empirical demonstration. *Data Mining and knowledge discovery*, 7(4):349–371, 2003.

[14] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod Record*, volume 28, pages 49–60. ACM, 1999.

[15] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986. ACM, 2010.

[16] C. Ding and J. Zhou. Log-based indexing to improve web site search. In *SAC*, pages 829–833. ACM, 2007.

[17] M. Eltahir and A. Dafa-Alla. Extracting knowledge from web server logs using web usage mining. In *Computing, Electrical and Electronics Engineering (ICCEEE)*, pages 413–417, Aug 2013.

[18] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.

[19] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at twitter. *Proceedings of the VLDB Endowment*, 5(12):1771–1780, 2012.

[20] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.

[21] A. Mueen, E. J. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, pages 473–484. SIAM, 2009.

[22] X. Ning and G. Jiang. HLAer: A system for heterogeneous log analysis, 2014. *SDM Workshop on Heterogeneous Learning*.

[23] R. Rajachandrasekar, X. Besseron, and D. K. Panda. Monitoring and predicting hardware failures in hpc clusters with ftb-ipmi. In *IPDPSW Workshops*, pages 1136–1143. IEEE, 2012.

[24] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *SIGKDD*, pages 262–270. ACM, 2012.

[25] K. S. Reddy, G. P. S. Varma, and I. R. Babu. Preprocessing the web server logs: An illustrative approach for effective usage mining. *SIGSOFT Softw. Eng. Notes*, 37(3):1–5, May 2012.

[26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[27] P. Sneath and R. Sokal. Unweighted pair group method with arithmetic mean. *Numerical Taxonomy*, pages 230–234, 1973.

[28] Wikipedia. Dbscan — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DBSCAN&oldid=672504091, 2015.

[29] C. Xu, S. Chen, and J. Cheng. Network user interest pattern mining based on entropy clustering algorithm. In *Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 200–204, Sept 2015.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the HotCloud'10*.