LogTree: A Framework for Generating System Events from Raw Textual Logs

Liang Tang and Tao Li
School of Computing and Information Sciences
Florida Internation University
Miami, 33199, USA
ltang002,taoli@cs.fiu.edu

***Abstract*—Modern computing systems are instrumented to generate huge amounts of system logs and these data can be utilized for understanding and complex system behaviors. One main fundamental challenge in automated log analysis is the generation of system events from raw textual logs. Recent works apply clustering techniques to translate the raw log messages into system events using only the word/term information. In this paper, we first illustrate the drawbacks of existing techniques for event generation from system logs. We then propose *LogTree*, a novel and algorithm-independent framework for events generation from raw system log messages. *LogTree* utilizes the format and structural information of the raw logs in the clustering process to generate system events with better accuracy. In addition, an indexing data structure, Message Segment Table, is proposed in *LogTree* to significantly improve the efficiency of events creation. Extensive experiments on real system logs demonstrate the effectiveness and efficiency of *LogTree*.**

***Keywords*-log analysis; event creation; message clustering;**

I. Introduction

Modern computing systems are instrumented to generate huge amounts of system log data. The log data describes the status of each component and records system internal operations, such as the starting and stopping of services, detection of network connections, software configuration modifications, and execution errors. Analyzing system logs, as an attractive approach for automatic system management, monitoring and system diagnosis, has been enjoying a growing amount of attention [1] [2] [3] [4] [5] [6]. With the increase of the system complexity, most modern systems generate a huge amount of log data every day. For example, one PVFS2 server could produce more than 6000 text messages for every 5 seconds [7]. In current cloud computing environment, a data center typically maintains over thousands of servers. Therefore, it is a challenging task to analyze the huge amount of log data.

A. Analyzing System Logs

Generally there are two main challenges in performing automated analysis of system logs. The first challenge is transforming the logs into a collection of event types. Note that the number of distinct events observed can be very large and also grow rapidly due to the large vocabulary size as well as various parameters in log generation [8]. In addition, variability in log languages creates difficulty in deciphering events and errors reported by multiple products and components [4]. Once the log data has been transformed into the canonical form, the second challenge is the design of efficient algorithms for analyzing log patterns from the events. Table I shows an example of the SFTP [1] log collected from FileZilla [9]. In order to analyze the behaviors, the raw log messages need to be translated to several types of events. Figure 1 shows the corresponding event timeline created by the log messages. The event timeline provides a convenient platform for people to understand log behaviors and to discover log patterns.

Table I: An Example of FileZilla's log.

No.	Message
s_1	2010-05-02 00:21:39 Command: put "E:/Tomcat/apps/index.html" "/disk/...
s_2	2010-05-02 00:21:40 Status: File transfer successful, transferred 823 bytes...
s_3	2010-05-02 00:21:41 Command: cd "/disk/storage006/users/lt...
s_4	2010-05-02 00:21:42 Command: cd "/disk/storage006/users/lt...
s_5	2010-05-02 00:21:42 Command: cd "/disk/storage006/users/lt...
s_6	2010-05-02 00:21:42 Command: put "E:/Tomcat/apps/record1.html" "/disk/...
s_7	2010-05-02 00:21:42 Status: Listing directory /disk/storage006/users/lt...
s_8	2010-05-02 00:21:42 Status: File transfer successful, transferred 1,232 bytes...
s_9	2010-05-02 00:21:42 Command: put "E:/Tomcat/apps/record2.html" "/disk/...
s_{10}	2010-05-02 00:21:42 Response: New directory is: "/disk/storage006/users/lt...
s_{11}	2010-05-02 00:21:42 Command: mkdir "libraries"
s_{12}	2010-05-02 00:21:42 Error: Directory /disk/storage006/users/lt...
s_{13}	2010-05-02 00:21:44 Status: Retrieving directory listing...
s_{14}	2010-05-02 00:21:44 Command: ls
s_{15}	2010-05-02 00:21:45 Command: cd "/disk/storage006/users/lt...
...	... ...

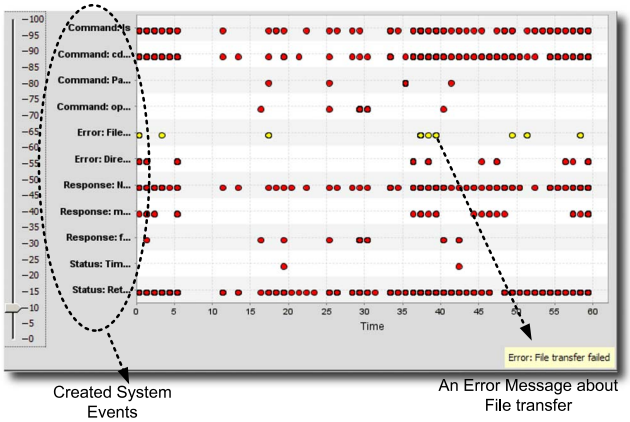


Figure 1: Event timeline for the FileZilla log example.

[1]SFTP: Simple File Transfer Procotol

1550-4786/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDM.2010.76

Recently, there has been lots of research on using data mining and machine learning techniques for analyzing system logs and most of them address the second challenge [1] [2] [3] [5] [6]. They focus on analyzing log patterns from events for problem determination such as discovering temporal patterns of system events, predicting and characterizing system behaviors, and performing system performance debugging. Most of these works generally assume the log data has been converted into events and ignore the complexities and difficulties in transforming the raw logs into a collection of events.

B. Contributions of the Paper

For modern complex systems, manually transforming raw log messages to system events is extremely expensive. Most modern complex systems are composed of various components developed by different development teams or even with different techniques [10]. Collecting all templates of logs from the system documents or source code is cumbersome and labor-intensive. In addition, there are some practical issues, such as the lack of the complete documents and the permissions of accessing the source code, that would make the manual approach impossible. Some recent works [1][8] apply clustering techniques to translate the raw log messages into system events automatically. The basic idea is to build clusters of the raw log messages where each cluster represents one type of events. In clustering the log messages, however, the existing techniques only make use of the word/term information and ignore the format and structural information.

In this paper, we first describe the drawbacks of existing clustering techniques for event generation from system logs. We show that, current methods which only make use of the word/term level information, are not able to achieve good performance in transforming the raw log data into system events. Note that events generation is the basis for further log pattern analysis. Thus low accuracy in log data transformation would greatly limit the success of log pattern analysis for problem determination. To address the limitations of existing methods, we then propose *LogTree*, a novel framework for events generation from raw system log messages. *LogTree* utilizes the format and structural information of the raw logs in the clustering process to generate system events with better accuracy. We collect system logs from 4 different and popular systems in real world applications. Extensive experiments are conducted to demonstrate the effectiveness of *LogTree*. An indexing data structure, Message Segment Table, is also introduced in *LogTree* to significantly improve the efficiency of events creation. Experiments show that *LogTree* is about 2 - 10 times faster than its competitors.

The rest of the paper is organized as follows: In Section II we formulate the problem of the system events generation and discuss the drawbacks of traditional solutions as well as our motivation. Section III first introduces the semi-structural model of the log messages and then presents our proposed similarity measurement based on this model. In Section IV, we describe the framework *LogTree* for the system events generation with its indexing data structure Message Segment Table. In Section V we present the experimental studies on 4 real system logs. Section VI summarizes the related studies on system events generation. Finally, Section VII concludes our paper and discusses the future work.

II. System Events Generation

Formally, a series of system log is a set of messages $S = \{s_1, s_2, \cdots, s_n\}$, where s_i is a log message, $i = 1, 2, \cdots, n$, and n is the number of log messages. The length of S is denoted by $|S|$, i.e., $n = |S|$. The objective of the event creation is to find a representative set of message S^*, to express the information of S as much as possible, where $|S^*| = k \leq |S|$, each message of S^* represents one type of event, and k is a user-defined parameter. The intuition is illustrated in the following Example.

Example 1: Table I shows a set of 15 log messages generated by the FileZilla client. It mainly consists of 6 types of messages, which include 4 different commands (e.g., "put", "cd", "mkdir", and "ls"), responses, and errors. Therefore, the representative set S^* could be created to be $\{s_1, s_2, s_3, s_7, s_{11}, s_{14}\}$, where every type of the command, response and error is covered by S^*, and $k = 6$.

We hope the created events to cover the original log as much as possible. The quality of S^* can be measured by the *event coverage*.

Definition 1: Given two sets of log messages S^* and S, $|S^*| \leq |S|$, the *event coverage* of S^* with respect to S is $J_C(S^*, S)$, which can be computed as follows:

$$J_C(S^*, S) = \sum_{x \in S} \max_{x^* \in S^*} F_C(x^*, x),$$

where $F_C(x^*, x)$ is the similarity function of the log message x^* and the log message x. We will discuss the similarity function in detail in Section III.

A. Problem Statement

Given a series of system log S with a user-defined parameter $0 \leq k \leq |S|$, the goal is to find a representative set $S^* \subseteq S$, which satisfies:

$$\begin{aligned} & \max J_C(S^*, S), \\ \text{subject to} \quad & |S^*| = k. \end{aligned}$$

Clearly, the system event generation can be regarded as a text clustering problem [12] where an event is the centroid or medoid of one cluster. However, those traditional text clustering methods are not appropriate for system logs. We show that those methods, which only extract the information at the word level, cannot produce an acceptable accuracy of the clustering of system logs.

B. Why is the word level information not enough?

It has been shown in [4] that log messages are relatively short text messages but have large vocabulary size. As a result, two messages of the same event type shares very few common words. It is possible two messages of the same type has two totally different sets of words.

The following is an example of two messages from the PVFS2 log file [7]. The two messages are status messages. Both of them belong to the same event type ***status*** which prints out the current status of the PVFS2 internal engine.

bytes read : 0 0 0 0 0 0
metadata keyval ops : 1 1 1 1 1 1

Note that the two messages have no words in common and clustering analysis purely based on the word level information would not reveal any similarity between the two messages. The similarity scores between the two messages (the cosine similarity [12], the Jaccard similarity [13] or the words matching similarity [8]) are 0.

C. Motivation

Although there is no common words between the two messages in Section II-B, the structure and format information implicitly suggest that the two messages could belong to a same category as shown in Figure 2. The intuition is straightforward: two messages are both split by the ':'; the left parts are both English words, and the right parts are 6 numbers separated by a tab. Actually, people often guess the types of messages from the structure and format information as well.

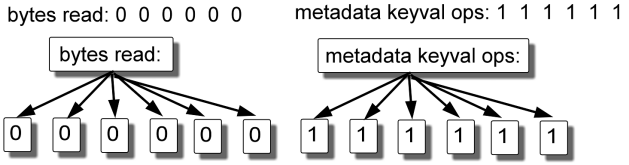


Figure 2: Two status messages in PVFS2.

In real system applications, the structure of log messages often implies critical information. The same type of messages are usually assembled by the same template, so the structure of log messages indicates which internal component generates this log message. Therefore, we should consider the structure information of each log message instead of just treating it as a sentence.

III. Semi-structural Log Message

We propose a general semi-structural model to represent the system log messages. It is able to capture the important structural and format information in many real-world system logs. Formally, a semi-structural log is represented by a tree $T = \{V, E, L, v_{root}, P\}$, where V is the set of nodes, E is the set of edges, L is a mapping function and defined as L: $V \rightarrow P$, P is the set of log message segments (or phrases), and v_{root} is the root node. Note that this tree is just semi-structural as we don't perform information extraction from the log messages. For building the tree, we only employ the context-free language parser to construct a hierarchical structure of message segments. Example 2 shows examples for FileZilla [9] and MySQL [14] logs.

Example 2: Figure 3 shows two semi-structural log messages for the FileZilla client logs. The first tree in Figure 3 shows an execution of a SFTP command "cd". The second one is about a response message from the SFTP server. We transform it to the semi-structural log T_1 as follows:

$$
\begin{aligned}
T_1 &= \{V_1, E_1, L_1, v_{root_0}, P_1\},\\
V_1 &= \{v_1, v_2, v_3\},\\
E_1 &= \{(v_1, v_2), (v_1, v_3)\},\\
v_{root_1} &= v_1,\\
P_1 &= \{\text{"Response:", "New directory is:", "/disk}\\
&\quad \text{/storage006/users/ltang002/MyFiles"}\},\\
L_1(v_1) &= \text{"Response:",}\\
L_1(v_2) &= \text{"New directory is:",}\\
L_1(v_3) &= \text{"/disk/storage006/users/ltang002/MyFiles".}
\end{aligned}
$$

Figure 4 shows a semi-structural log message of the MySQL Server, which indicates the starting of the MySQL backend server.

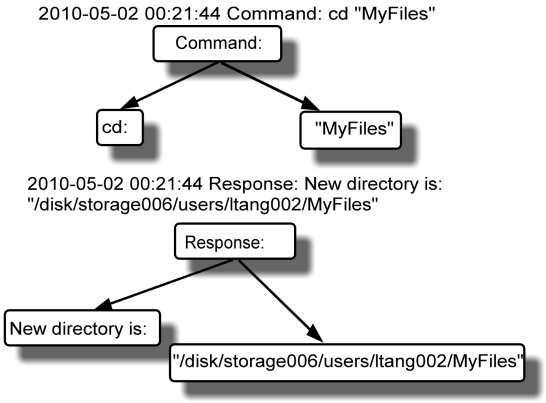


Figure 3: A Semi-structural FileZilla log messages.

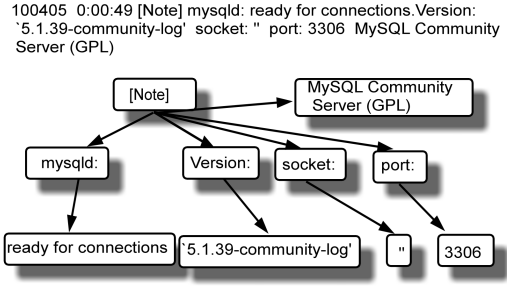


Figure 4: A Semi-structural MySQL log Message.

A. Log parsing

Building semi-structural log messages from the plain text is accomplished by a simple context-free grammar parser[2]. For example, the context-free grammar for the FileZilla log could be established by the following 5 rules:

Message	$\rightarrow$	Indicator **:** Content
Content	$\rightarrow$	Segment | Content **,** Segment| Content **tab** Segment
Segment	$\rightarrow$	Sentence | Sentence **:** Term
Sentence	$\rightarrow$	Term | Sentence **whitespace** Term
Term	$\rightarrow$	**word** | **“ word ”**

In the experiments of this paper, we have implemented all the parsers for 4 different system logs in Java. None of the parsers costs more than 200 lines source code. Existing tools, such as JFlex[3] and Cup [4], can generate the Java source code for the parsers based on the grammar.

It is important to note that the result of the system event creation depends on the implementation of the parser just as many other data preprocessings. However, a lot of previous studies focus on analyzing the format of textual logs. Some recent work even suggests the parser can be automatically created by an incremental learning approach [16]. So the discussion of the parser is beyond the scope of this paper.

B. Similarity of Semi-structural Log Messages

Intuitively, for two semi-structural log messages T_1 and T_2, the coverage of T_1 with respect to T_2 depends on the number of similar nodes and edges of T_1 and T_2. A lot of methods have been proposed to compute the distance of two labeled trees in previous literatures [17] [18]. The edit distance and alignment distance are two typical methods where only the ancestral relation of nodes, not the immediate parental relation of nodes, are maintained during the computation. However, they are not appropriate for semi-structural log messages. For example, in Figure 4, we cannot derive the relation (“[Note]”, “3306”) from the two edges (“[Note]”, “port”) and (“port”, “3306”), because (“[Note]”, “3306”) expresses a different meaning.

We have analyzed many different kinds of system logs, such as the FTP/SFTP Client: FileZilla [9], Database Server: MySQL [14], Parallel File System: PFVS2 [7] and Apache HTTP Server [19]. One observation is that, ***higher level nodes is more important to discriminate the log message than lower level nodes***. As shown in the log messages of Figure 3, the root node indicates the current action of the log message, which is the most important node. Therefore, high importance should be assigned to high level nodes.

[2]Note that we do not use deep natural language processing techniques to parse the message sentences into dependency trees [15] since the grammar of the log messages are regular or context-free, which is much simpler than natural language. In addition, we do not need a large and labeled training data set for word tagging.

[3]http://jflex.de

[4]http://www2.cs.tum.edu/projects/cup/

Let $V(T)$ denote the set of nodes of tree T. For a node v, let $T(v)$ denote the subtree rooted at node v, and $C(v)$ denote the set of the children of v. Let $d(L(v), L(u))$ denote the similarity of message segments of node v, u (We will discuss the definition $d(\cdot,\cdot)$ in Section III-C.). For two nodes v and u, let $M^*_C(v,u)$ denote the best matching between v's children and u's children. Formally, $M^*_C(v,u)$ is a set of pairs $\{(v_i,u_j)\}$, which maximizes $\sum_{(v_i,u_j)\in M^*_C(v,u)} d(L(v_i), L(u_j))$, where $v_i \in C(v)$, $v_j \in C(u)$ and each node of $C(v)$ and $C(u)$ can be contained by exactly one pair.

Definition 2: Given two log messages s_1 and s_2, let $T_1 = \{V_1, E_1, L, r_1, P\}$ and $T_2 = \{V_2, E_2, L, r_2, P\}$ be the corresponding semi-structural log messages of s_1 and s_2 respectively, the coverage function $F_C(s_1, s_2)$ is computed as follows:

$$F_C(s_1, s_2) = \frac{F'_C(r_1, r_2, \lambda) + F'_C(r_2, r_1, \lambda)}{2},$$

where

$$F'_C(v_1, v_2, w) = w \cdot d(L(v_1), L(v_2)) + \sum_{(v,u)\in M^*_C(v_1,v_2)} F'_C(v, u, w \cdot \lambda),$$

$M^*_C(v_1, v_2)$ is the best matching between v_1's children and v_2's children, and λ is a parameter, $0 \leq \lambda \leq 1$.

Note that the function F_C is obtained by another recursive function F'_C. F'_C computes the similarity of two subtrees rooted at two given nodes v_1 and v_2 respectively. To compare the two subtrees, besides the root nodes v_1 and v_2, F'_C needs to consider the similarity of their children as well. Then, there is a problem that which child of v_1 should be compared with which child of v_2. In other words, we have to find the best matching $M^*_C(v_1, v_2)$ in computing F'_C. Finding the best matching is actually a maximal weighted bipartite matching problem. In our implementation, we use a simple greedy strategy to find the matching. For each child of v_1, we assign it to the maximal matched node in unassigned children of v_2. This time complexity of the greedy approach is $O(n_1 n_2)$ where n_1 and n_2 are the numbers of children of v_1 and v_2, respectively. F'_C requires another parameter w, which is a decay factor. In order to improve the importance of higher level nodes, this decay factor is used to decrease the contribution of similarities at a lower level. Since $\lambda \leq 1$, the decay factor w decreases along with the recursion depth.

C. Similarity of Log Message Segments

Function $d(\cdot,\cdot)$ determines the similarity between two log message segments. A log message segment is a phrase of a log message, which is a sequence of words and symbols. In information retrieval, there are a lot of measurements to compute the similarity between two sentence phrases. But for the log message segments, we consider two additional information as follows.

- symbols, such as ':', '[', are important to identify the templates of the log message. Function $d(\cdot,\cdot)$ makes use of those symbols in computing the similarity of two log message segments.
- The type of a word/term implies the format information. If two log messages are generated by the same template, even if they have different sets of words/terms, the formats of words should be similar. In our system, there are six types $\boldsymbol{T} = \{$ *word, number, symbol, date, IP, comment* $\}$. Given a term w in a message segment m_1, $t(w)$ denotes the type of the w. $t(w) \in \boldsymbol{T}$.

As mentioned in [8], the word ordering is important to identify the similarity of two message segments. Therefore, we define the function $d(\cdot,\cdot)$ as follows:

Definition 3: Given two message segments $m_1 = p_1 \cdots p_{n_1}$ and $m_2 = q_1 \cdots q_{n_2}$, where $p_1, \cdots, p_{n_1}$ and $q_1, \cdots, q_{n_2}$ are terms of m_1 and m_2 respectively, $d(m_1, m_2)$ is computed as follows:

$$d(m_1, m_2) = \frac{1}{\sqrt{n_1 \cdot n_2}} \sum_{i=1}^{min(n_1,n_2)} x_i$$

where

$$x_i = \begin{cases} 0, & \text{for } t(p_i) \neq t(q_i); \quad (1) \\ \alpha, & \text{for } t(p_i) = t(q_i), p_i \neq q_i; \quad (2) \\ 1, & \text{for } p_i = q_i, \quad (3) \end{cases}$$

and α is a user-defined parameter, $0 \leq \alpha \leq 1$.
Function d can be computed in a linear time complexity.

D. Comparison with Tree kernel

In natural language processing, tree kernel is a kind of similarity measurement for dependency trees [20] [21], which is similar to our coverage function F_C. However, directly applying tree kernel for clustering the log message is not appropriate in our work. The reasons are as follows.

- Tree kernel does not assign different importance values for nodes at different levels. For most log message, the high level nodes have more importance than lower nodes.
- Computing tree kernels is time-consuming. Hence, it is not appropriate to use tree kernels in analyzing massive system logs. In the experimental section, we systematically compare the performances of tree kernel measurement with our method in real system logs.

IV. The Framework of Event Generation from System Logs

The generation of system events from the logs can be achieved by a clustering algorithm. We develop an algorithm independent framework to generate system events, which has the following two major features:

- *Efficiency:* For current large complex and distributed systems, even the algorithm with polynomial time complexity cannot efficiently handle the massive logs. Our framework improves the efficiency of the clustering algorithms with the auxiliary index structure.
- *Incremental Maintenance:* Log messages are constantly generated over time. Our framework for system event generation can be maintained incrementally.

In the next two subsections, we discuss the two features in detail.

A. Message Segment Table

Many message segments are fixed in the source code of the system. For example, the second log message in Figure 3 has a message segment: *"New directory is"*. This message segment is generated with different targets or parameters to assemble many different response logs. Figure 4 shows another example for MySQL. The *"ready for connection"* can be assembled with different daemon processes of the storage engine. Therefore, a lot of message segments are duplicated in the log data. Based on this fact, we propose an indexing data structure, called *Message Segment Table* (MST) to improve the efficiency of the events creation.

Figure 5 shows the overview of the *Message Segment Table*. This table is a two dimensional dynamic table. Each entry stores the similarity score of a pair of two message segments. Each column and row represent a unique message segment. Let $col(i)$ and $row(i)$ denote the message segments of i-th column and i-th row index respectively. $entry(i,j)$ denotes the entry at column i and row j, which is the similarity score of $col(i)$ and $row(j)$. So the table can be viewed as a dynamic similarity matrix of message segments. Since every node is only possible to be compared with the same level's nodes, we don't need to put nodes at different levels in one table. Thus, we create separate MSTs for nodes at different levels.

A hash table is used to maintain the indexes of message segments in the MST. For each message segment, the corresponding column index and row index can be searched from this hash table. So this table is called Column Hash Table. Note that for a unique message segment, its column index and row index are the same. Figure 5 shows an example of the column hash table. The search key of the hash table is the message segment *"Segment"*, and the search value is a tuple consists of the corresponding column index and the number of occurrences *"< Col, Occur >"*. The number of the occurrences is used to distinguish the frequent message segments and infrequent message segments. Some message segments only appear very few times in the log, such as the parameters of an event. Considering the limitation of the main memory size, we only store those frequent message segments in the MST. For this purpose, when we scan all log messages, we keep track of the number of occurrences of each message segment using the column hash table. After we put all message segments into the column hash table, we remove those segments whose frequency is less than a user-defined threshold f_{min}, $0 \leq f_{min} \leq 1$. The MST

and the column hash table both become smaller when f_{min} increases.

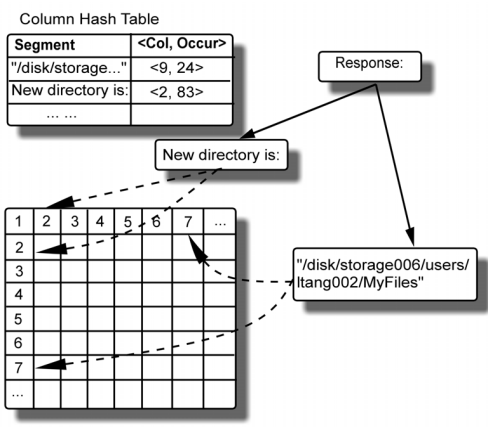


Figure 5: Overview of the Message Segment Table.

1) Building the MST: This message segment table is built on a given set of semi-structural log messages. Using the depth-first or breadth-first traversal, every log message segment can be visited and inserted into the column hash table. Once the column hash table is created, the MST can be built by computing the similarity score of every pair of message segments in the table. Algorithm 1 describes the detail of building the MST and its column hash table. $V(\boldsymbol{T})$ denotes the set of all nodes in the forest $\boldsymbol{T}$. For the hash table CT, $CT[k]$ denotes the value of key k, and $CT.i$ denotes the i-th key. For each node v of the log tree, $v.col$ indicates the field of its corresponding column index at the MST. It is the virtual link of the node as shown Figure 5. The time complexity of building the MST is $O(|V(\boldsymbol{T})|)$.

2) Computing F_C: In computing the coverage function F_C of two given trees T_1 and T_2, we do not need to access the message segment of each node. We could obtain the similarity score directly from the virtual link to the MST. The virtual link of the node is stored by an integer. Since the maximal length of message segment is a constant, the time complexity of computing F_C doesn't change here. But the I/O cost of computing F_C is reduced largely. The similarity score of $d(\cdot,\cdot)$ in the MST is stored by a float number. The total I/O cost of computing $d(\cdot,\cdot)$ is reduced from the total length of two message segments to be just 3 numbers. Furthermore, it enhances the cache-consciousness of the algorithm, which can bring a huge improvement on the algorithm efficiency in modern computing systems.

3) Update: The log message is a kind of streaming data, which is constantly generated over the time. So our indexing data structure should be updated efficiently. Since our MST is built by a dynamic table with a hash table, we can easily insert or remove a message segment. The time complexities of the two operations are $O(1)$.

Data: The set of semi-structural log messages $\boldsymbol{T}$
Result: The Message Segment Table MST_l and the column hash table CT_l for tree level l

```
1  col_max ← 0
2  create column hash table CT_l
3  // fill each message segment into CT_l
4  foreach T ∈ T do
5      foreach v ∈ V(T) and v is at level l do
6          if L(v) is in CT_l then
7              < col, occur > ← CT_l[L(v)]
8              occur ← occur + 1
9          else
10             col ← col_max
11             insert (L(v), < col, 1 >) into CT_l
12             col_max ← col_max + 1
13         end
14         v.col ← col
15     end
16 end
17 // remove infrequent message segments from CT_l
18 foreach (e, < col, occur >) ∈ CT_l do
19     if occur < f_min · |V(T)| then
20         remove this message segment e from CT_l
21     end
22 end
23 // create message segment table based on CT_l
24 create the 2D dynamic table MST_l
25 for i ← 0 to col_max − 1 do
26     for j ← i to col_max − 1 do
27         MST_l[i, j] ← d(CT_l.i, CT_l.j)
28     end
29 end
```

Algorithm 1: MST building algorithm.

B. The Framework of Events Generation

The system event generation is based on the data clustering algorithm. Various clustering algorithms can be plugged in the *LogTree* framework. We have developed a log analysis system which uses *LogTree* to create events. In our system, we choose a hierarchical clustering algorithm. Hierarchical clustering can provide a multi-level view of the events to the users. The users can roll up or drill down at different level in the event timeline as the OLAP operations in the data cube.

Algorithm 2 describes the process of events creation in our framework. The data clustering Clu is input by the user. As for Clu, the input data objects and the similarity function are provided. The clustering algorithm returns representative data objects as the a set of events.

The data clustering algorithm Clu is an input specified by the user. The input data objects and the similarity function are provided to the clustering algorithm Clu and representative data objects for each cluster are returned as the a set of events. The entire log data is usually too large to fit in the main memory. *LogTree* only executes message clustering on a time-frame of the log. Then it scans the entire log and assigns each log message to one of created events.

Data: A sequence of log messages S,
A clustering algorithm Clu
Result: A set of events S^*
$\boldsymbol{T} \leftarrow \varnothing$
foreach $s \in S$ **do**
  transform s to tree t
  $\boldsymbol{T} \leftarrow \boldsymbol{T} \cup \{t\}$
end
find the maximal level of trees l_{max}.
build the Message Segment Tables MST_1,..., $MST_{l_{max}}$ for level 1,2,...,l_{max}
call Clu with parameters $\boldsymbol{T}$, MST_1,..., $MST_{l_{max}}$
collect cluster representative objects from Clu to S^*

Algorithm 2: Framework of Event Creation

V. Evaluation

A. Experimental Platforms

Our system is developed in Java 1.5 Platform. Table II shows the summary of two machines where we run our experiments. All experiments except for scalability test are conducted in Machine1, which is a 32-bits machine. As for the scalability experiment, the program needs over 2G main memory, so the scalability experiment is conducted in Machine2, which is a 64-bits machine. All the experimental programs are single-threaded.

Table II: Experimental Machines

Machine	OS	CPU	Memory	JVM Heap Size
Machine1	Windows 7	Intel Core i5 @2.53GHz	4G	1.5G
Machine2	Linux 2.6.18	Intel Xeon(R) X5460@3.16GHz	32G	3G

B. Data Collection

In order to evaluate our work, we collect the log data from 4 different and popular real systems. Table III shows the summary of our collected log data. The log data is collected from the server machines/systems in the computer lab of a research center. Those systems are very common system services installed in the many data centers.

- FileZilla client 3.3[9] log, which records the client's operations and responses from the FTP/SFTP server.
- MySQL 5.1.31[14] error log. The MySQL database is hosted in a developer machine, which consists of the error messages from the MySQL database engine.
- PVFS2 server 2.8.2[7] log. It contains errors, internal operations, status information of one virtual file sever.
- Apache HTTP Server 2.x[19] error log. It is obtained from the hosts for the center website. The error log mainly records various bad HTTP requests with corresponding client information.

C. Comparative Methods

In order to evaluate the effectiveness and efficiency of our work, we use 4 other related and traditional methods in the experiments. Table IV shows all the comparative methods used in the experiments. As for "Tree Kernel", the tree structure is the same as that used in the our method *LogTree*. Since the tree node of the log message is not labeled, we can only choose sparse tree kernel for "Tree Kernel" [21]. The experiments of the event generation are conducted using two clustering algorithms, K-Medoids [22] and Single-Linkage [13]. The reason that we choose the two algorithms is that K-Medoids is the basic and classical algorithm for data clustering, and Single-Linkage is a typical hierarchical clustering which is actually used in our system. It should be pointed out that our comparisons are focus on similarity measurements which are independent from a specific clustering algorithm. We expect that the insights gained from our experiment comparisons can be generalized to other clustering algorithms as well.

Table III: Log data summary.

System	System Type	#Messages	#Words per message	#Types
FileZilla	SFTP/FTP Client	22,421	7 to 15	4
MySQL	Database Engine	270	8 to 35	4
PVFS2	Parallel File System	95,496	2 to 20	4
Apache	Web Server	236,055	10 to 20	5

Table IV: Summary of comparative methods.

Method	Description
"TF-IDF"	the classical text clustering method using the vector space model with tf-idf transformation.
"Tree Kernel"	the tree kernel similarity introduced in [21].
'Matching"	the method using words matching similarity in [8].
"LogTree"	our method using semi-structural log and Message Segment Table.
"Jaccard"	Jaccard Index similarity of two log messages.

D. The Quality of Events Generation

The entire log is split into different time frames. Each time frame is composed of 2000 log messages and labeled with the frame number. For example, Apache2 denotes the 2th frame of the Apache log. The quality of the results is evaluated by the F-measure (F1-score) [12]. First, the log messages are manually classified into several types. Then, the cluster label for each log message is obtained by the clustering algorithm. The F-measure score is then computed from message types and clustered labels. Table V and Table VI show the F-measure scores of K-Medoids and Single-Linkage clusterings with different similarity approaches respectively. Since the result of K-Medoids algorithm varies by initial choice of seeds, we run 5 times for each K-Medoids clustering and the entries in Table V are computed by averaging the 5 runs.

Only "Tree Kernel" and "LogTree" need to set parameters. "Tree Kernel" has only one parameter, λ_s, to penalize matching subsequences of nodes [21]. We run it under different parameter settings, and select the best result for comparison. Another parameter k is the number of clusters for clustering algorithm, which is equal to the number of the types of log messages. Table VII shows the parameters used for "Tree Kernel" and "LogTree".

Table V: F-Measures of K-Medoids

Logs	TF-IDF	Tree Kernel	Matching	LogTree	Jaccard
FileZilla1	0.8461	**1.0**	0.6065	**1.0**	0.6550
FileZilla2	0.8068	**1.0**	0.5831	**1.0**	0.5936
FileZilla3	0.6180	**1.0**	0.8994	**1.0**	0.5289
FileZilla4	0.6838	0.9327	**0.9545**	0.9353	0.7580
PVFS1	0.6304	0.7346	0.7473	**0.8628**	0.6434
PVFS2	0.5909	0.6753	**0.7495**	0.6753	0.6667
PVFS3	0.5927	0.5255	0.5938	**0.7973**	0.5145
PVFS4	0.4527	0.5272	0.5680	**0.8508**	0.5386
MySQL	0.4927	0.8197	**0.8222**	**0.8222**	0.5138
Apache1	0.7305	0.7393	0.9706	**0.9956**	0.7478
Apache2	0.6435	0.7735	0.9401	**0.9743**	0.7529
Apache3	0.9042	0.7652	0.7006	**0.9980**	0.8490
Apache4	0.4564	0.8348	0.7292	**0.9950**	0.6460
Apache5	0.4451	0.7051	0.5757	**0.9828**	0.6997

Table VI: F-Measures of Single-Linkage

Logs	TF-IDF	Tree Kernel	Matching	LogTree	Jaccard
FileZilla1	0.6842	**0.9994**	0.8848	0.9271	0.6707
FileZilla2	0.5059	0.8423	0.7911	**0.9951**	0.5173
FileZilla3	0.5613	**0.9972**	0.4720	0.9832	0.5514
FileZilla4	0.8670	**0.9966**	0.9913	0.9943	0.6996
PVFS1	0.7336	0.9652	0.6764	**0.9867**	0.4883
PVFS2	0.8180	**0.8190**	0.7644	0.8184	0.6667
PVFS3	0.7149	0.7891	0.7140	**0.9188**	0.5157
PVFS4	0.7198	0.7522	0.6827	**0.8136**	0.6345
MySQL	0.4859	0.6189	**0.8705**	0.8450	0.5138
Apache1	0.7501	0.9148	0.7628	**0.9248**	0.7473
Apache2	0.7515	**0.9503**	0.8178	0.9414	0.7529
Apache3	0.8475	0.8644	0.9294	**0.9594**	0.8485
Apache4	0.9552	0.9152	0.9501	**0.9613**	0.6460
Apache5	0.7882	0.9419	0.8534	**0.9568**	0.6997

Table VII: Parameter settings

Log Type	k	λ_s	λ	α
FileZilla	4	0.8	0.7	0.1
MySQL	4	0.8	0.3	0.1
PVFS2	4	0.8	0.7	0.1
Apache	5	0.8	0.01	0.1

FileZilla log consists of 4 types of log messages. One observation is that, the root node of the semi-structural log is sufficient to discriminate the type of a message. Meanwhile, the root node produces the largest contribution in the similarity in "Tree Kernel" and "LogTree". So the two methods benefit from the structural information to achieve a high clustering performance.

PVFS2 log records various kinds of status messages, errors and internal operations. None of the methods can perform perfectly. The reason is that, in some cases, two log messages composed of distinct sets of words could belong to one type. Thus, it is difficult to cluster this kind of messages into one cluster. Section II-B gives an example about this.

MySQL error log is small, but some messages are very long. Those messages are all assembled by fixed templates. The parameter part is very short comparing with the total length of the template, so the similarity of [8] based on the templates wouldn't be interfered by the parameter parts very much. Therefore, "Matching" always achieves the highest performance.

Apache error log is very similar to FillZilla log. But it contains more useless components to identify the types of the error message, such as the client information. In our semi-structural log, those useless components are located at low level nodes. Therefore, when the parameter λ becomes small, their contributions to the similarity are reduced, then the overall performance becomes better.

To sum up, the "Tree Kernel" and "LogTree" methods outperform other methods. The main reason is that, the two methods capture both the word level information as well as the structural and format information of the log messages. In the next subsection, we show that our "LogTree" is more efficient than "Tree Kernel".

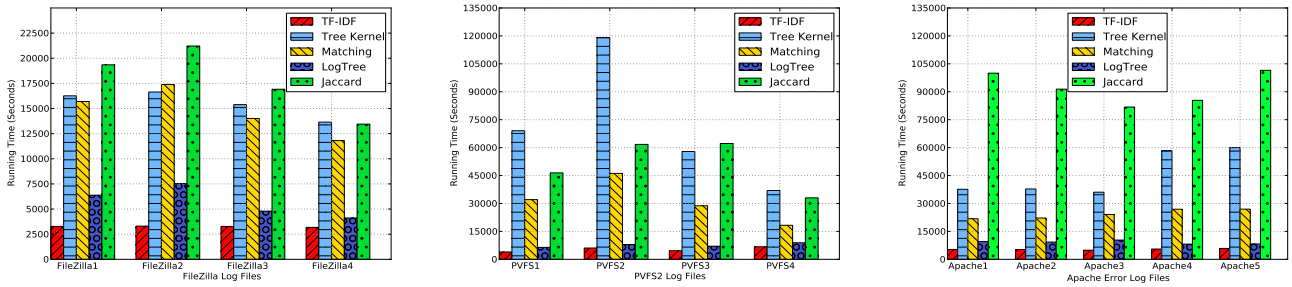
E. The Efficiency of Event Generation

We records the running time of each clustering algorithm on the log data. Due to the space limitation, we only show the running time of K-Medoids algorithm on FileZilla log, PVFS2 log, and Apache error log in Figure 6a, 6b and 6c. The running time is the average number of 5 runs. In the implementation, we build the similarity matrix of each pair of log messages at the beginning, whose time complexity is $O(N^2)$ where N is the number of samples. Thus, the majority of the running time is used for building the similarity matrix. As for "LogTree", the threshold of Message Segment Table is $f_{min} = 0.00001$. The parameter choice depends on the size of the main memory. **Note that the running time of LogTree includes the time for building MST.**

Figure 6 shows that the vector space model based text clustering, "TF-IDF", is the most efficient approach. The reason is that, the sparse vector is a compact representation of the log message. The cosine similarity of two sparse vectors can be obtained in one pass. The vector transformation can be achieved in a linear time complexity by using a hash table. Furthermore, the cosine similarity of vectors do not consider the structural information of two log messages.

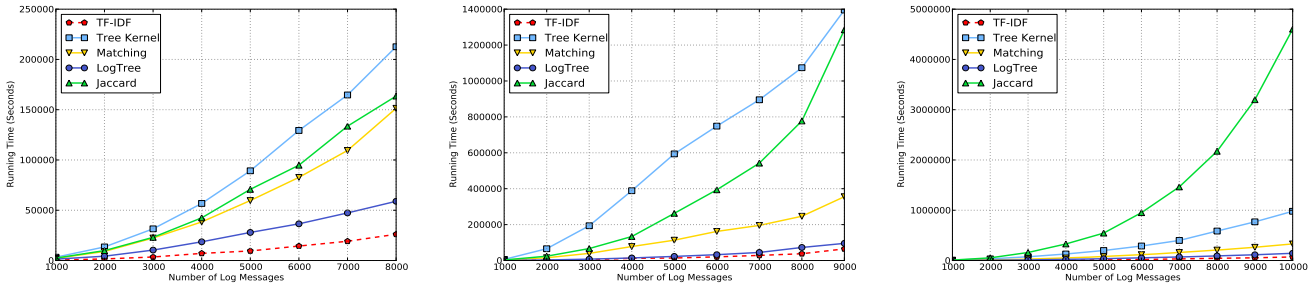
Our proposed approach, "LogTree", is in the second place in Figure 6. With the help of the Message Segment Table, it can save a lot of computation to obtain the similarity of two tree nodes. However, in order to consider the structural information of the log message, the similarity function F_C still has to find the most matched node in each level of the tree. So it cannot be completed in one pass as the cosine similarity.

The other three methods, "Tree Kernel", "Matching" and "Jaccard" are slower than the previous two methods. One reason is that, those three methods do not provide a compact representation of the log message in the main memory. For the similarity of every two messages, they all have to access the original messages, requiring more CPU and I/O costs. As for "Tree Kernel", it compares every pair of nodes in the same level and its time complexity $O(mn^3)$ is very large, where m and n are the number of nodes in the two trees respectively [21].



(a) FileZilla logs (b) PVFS2 logs (c) Apache logs

Figure 6: The Efficiency of K-Medoids



(a) FileZilla logs (b) PVFS2 logs (c) Apache logs

Figure 7: Time Scalability of K-Medoids

F. The Scalability of Event Generation

1) Time Scalability: We run all methods on the logs with different sizes to evaluate their time scalability. Figure 7a, 7b and 7c show the scalability results of K-Medoids algorithm with different similarity measurements. The running time is obtained by averaging 5 different runs as mentioned before. This experiment needs more than 2G main memory, so it is conducted in a different machine that we introduced in Section V-A. The results shown in Figure 7a, 7b and 7c are consistent with the efficiency tests in previous subsection. "TF-IDF" is the most efficient approach, and our proposed method,"LogTree", is in the second place, where the threshold for MST $f_{min} = 0.00001$.

2) Space Scalability: The space costs for all methods are identical except for our method "LogTree". For "LogTree", there is an additional message segment table. The message segment table is always maintained in the main memory. Figure 8 shows the space cost of message segment tables, which is the sum of the entries of each level's MST, where $f_{min} = 0.00001$. In this figure, FileZilla log has the largest space cost in MSTs. The reason is that, the diversity of FileZilla log is very low, so MST almost covers all message segments. On the other hand, the diversity of PVFS2 log is high, which covers various kinds of status messages, error, internal operations. Thus, only a few message segments' frequencies are greater than f_{min} and are maintained in the MST.

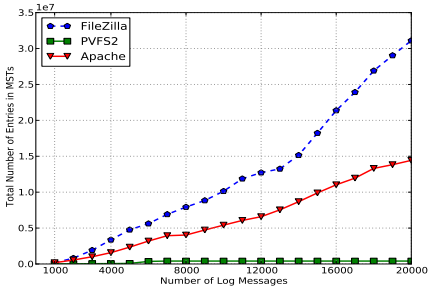


Figure 8: Space Scalability of LogTree.

Every entry of MST is a float number, which occupies 4 bytes. The largest actual memory cost of those MSTs in Figure 8 is $3.2 \times 10^7 \times 4$ = 128M bytes. Comparing to the similarity matrix of log messages built by the clustering algorithm, $20000 \times 20000/2 \times 4$ = 1.6G bytes, the MST's cost can be ignored.

G. A Case Study

We have developed a log analysis toolkit using *Logtree* for events generation from system log data. Figure 9 shows a case study of using our developed toolkit for detecting configuration errors in Apache Web Server. The configuration error is usually cased by human, which is quite different from random TCP transmission failures, or disk read errors. As a result, configuration errors typically lead

to certain patterns. However, the Apache error log file has over 200K log messages. It is difficult to discover those patterns directly from the raw log messages. Figure 9 shows the event timeline window of our toolkit, where the user can easily identify the configuration error in the time frame. This error is related to the permission setting of the HTML file. It causes continuous permission denied errors in a short time. In addition, by using the hierarchical clustering method, *LogTree* provides multi-level views of the events. The user could use the slider to choose a deeper view of events to check detail information about this error.

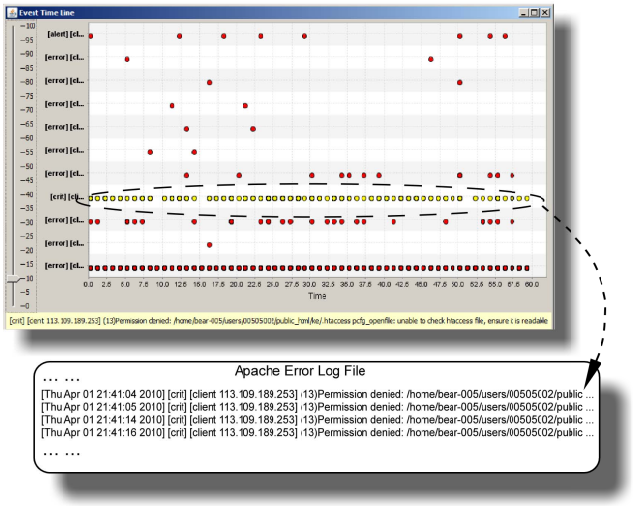


Figure 9: A case study of the Apache HTTP server log.

VI. Related Work

The related work about the log data analysis can be broadly summarized into two categories. One category is on system event generation from raw log data [8] [16] [11] and the other category is on analyzing patterns from system events [1] [2] [3] [5] [6].

Our work in this paper belongs to the first category. A word matching similarity measurement is introduced in [8] for clustering the log messages. One problem is that, some types of log messages may not have much common words. [11] develops a 4-steps partitioning method for clustering the log messages based on some characteristics of the log format. However, the methods is not able to handle the situation that one event type with multiple message formats. [16] studies an automatically learning approach to capture the format the log. It can be treated as an assistant tool for many related works in this category.

VII. Conclusion and Future Work

In this paper, we show that traditional methods which only make use of the word level information, are not able to achieve an acceptable accuracy for generating system events from the log data. To address the limitation of existing methods, we propose *LogTree*, a novel and algorithm-independent framework for events creation from system log messages. *LogTree* utilizes the format and structural information in the log message and employs Message Segment Table for effective and efficient system event generation.

As for the future work, we will integrate the automatic log structure learning approach into our framework. Specially, semi-supervised learning techniques can be applied in our framework to capture the structural information of log messages. We hope to identify those structures of log messages from a few labeled samples provided by the users.

References

[1] W. Peng, C. Perng, T. Li, and H. Wang, "Event summarization for system management," in *Proceedings of ACM KDD*, San Jose, California, USA, August 2007, pp. 1028–1032.

[2] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Mining console logs for large-scale system problem detection," in *SysML*, San Diego, CA, USA, December 2008.

[3] J. L. Hellerstein, S. Ma, and C.-S. Perng, "Discovering actionable patterns in event data," *IBM Systems Journal*, vol. 43, no. 3, pp. 475–493, 2002.

[4] J. Stearley, "Towards informatic analysis of syslogs," in *Proceedings of IEEE International Conference on Cluster Computing*, San Diego, California, USA, September 2004, pp. 309–318.

[5] T. Li, F. Liang, S. Ma, and W. Peng, "An integrated framework on mining logs files for computing system management," in *Proceedings of ACM KDD*, Chicago, Illinois, USA, August 2005, pp. 776–781.

[6] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *Proceedings of the 29th International Conference on Distributed Computing Systems (ICDCS'09)*, 2009, pp. 623–630.

[7] "PVFS2 : The state-of-the-art parallel I/O and high performance virtual file system," http://pvfs.org.

[8] M. Aharon, G. Barash, I. Cohen, and E. Mordechai, "One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs," in *Proceedings of ECML/PKDD*, Bled, Slovenia, September 2009, pp. 227–243.

[9] "FileZilla: An open-source and free FTP/SFTP solution," http://filezilla-project.org.

[10] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[11] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of ACM KDD*, Paris, France, June 2009, pp. 1255–1264.

[12] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill, 1984.

[13] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison Wesley, 2005.

[14] "MySQL: The world's most popular open source database," http://www.mysql.com.

[15] C. D. Manning and H. Schuetze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[16] K. Fisher, D. Walker, and K. Q. Zhu, "Incremental learning of system log formats," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 85–90, 2010.

[17] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal on Computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[18] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. K. Sellis, "Clustering xml documents using structural summaries," in *Proceedings of EDBT Workshops*, Heraklion, Crete, Greece, March 2004, pp. 547–556.

[19] "Apache HTTP Server : An Open-Source HTTP Web Server," http://www.apache.org.

[20] A. Moschitti, "Making tree kernels practical for natural language learning," in *Proceedings of EACL*, Trento, Italy, April 2006.

[21] A. Culotta and J. S. Sorensen, "Dependency tree kernels for relation extraction," in *Proceedings of ACL*, Barcelona, Spain, July 2004, pp. 423–429.

[22] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques, 2ed*. Morgan Kaufmann, 2005.