Searching Similar Segments over Textual Event Sequences

Liang Tang Tao Li Shu-Ching Chen
School of Computer Science
Florida International University
Miami, Florida, 33199, U.S.A
{ltang002,taoli,chens}@cs.fiu.edu

Shunzhi Zhu
School of Computer & Information Engineering
Xiamen University of Technology
Xiamen, P.R. China
szzhu@xmut.edu.cn

ABSTRACT

Sequential data is prevalent in many scientific and commercial applications such as bioinformatics, system security and networking. Similarity search has been widely studied for symbolic and time series data in which each data object is a symbol or numeric value. Textual event sequences are sequences of events, where each object is a message describing an event. For example, system logs are typical textual event sequences and each event is a textual message recording internal system operations, statuses, configuration modifications or execution errors. Similar segments of an event sequence reveal similar system behaviors in the past which are helpful for system administrators to diagnose system problems. Existing search indexing for textual data only focuses on unordered data. Substring matching methods are able to efficiently find matched segments over a sequence, however, their sequences are single values rather than texts. In this paper, we propose a method, *suffix matrix*, for efficiently searching similar segments over textual event sequences. It provides an integration of two disparate techniques: locality-sensitive hashing and suffix arrays. This method also supports the k-dissimilar segment search. A k-dissimilar segment is a segment that has at most k dissimilar events to the query sequence. By using *random sequence mask* proposed in this paper, this method can have a high probability to reach all k-dissimilar segments without increasing much search cost. We conduct experiments on real system log data and the experimental results show that our proposed method outperforms alternative methods using existing techniques.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods

General Terms

Algorithms, Experimentation

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505762.

Keywords

Textual Sequence; Similarity Search; Log Event

1. INTRODUCTION

Sequential data is prevalent in many real-world applications such as bioinformatics, system security and networking. Similarity search is one of the most fundamental techniques in sequential data management. A lot of efficient approaches are designed for searching over symbolic sequences or time series data, such as DNA sequences, stock prices, network packets and video streams. A textual event sequence is a sequence of events, where each event is a plain text or message. For example, in system management, most system logs are textual event sequences which describe the corresponding system behaviors, such as the starting and stopping of services, detection of network connections, software configuration modifications, and execution errors [24, 22, 29, 30, 36, 37]. System administrators utilize the event logs to understand system behaviors. Similar system events reveal potential similar system behaviors in history which help administrators to diagnose system problems. For example, four log messages collected from a supercomputer [4] in Sandia National Laboratories are listed below:

```
- 1131564688 2005.11.09 en257 Nov 9 11:31:28 en257/en257
ntpd[1978]:  ntpd exiting on signal 15
- 1131564689 2005.11.09 en257 Nov 9 11:31:29 en257/en257
ntpd:  failed
- 1131564689 2005.11.09 en257 Nov 9 11:31:29 en257/en257
ntpd:  ntpd shutdown failed
- 1131564689 2005.11.09 en257 Nov 9 11:31:29 en257/en257
ntpd:  ntpd startup failed
```

The four log messages describe a failure in restarting of the `ntpd` (Network Time Protocol daemon). The system administrators need to first know the reason why the `ntpd` could not restart and then come up with a solution to resolve this problem. A typical approach is to compare the current four log messages with the historical `ntpd` restarting logs and see what is the difference with them. Then the administrators can find out which steps or parameters might cause this failure. To retrieve the relevant historical log messages, the four log messages can be used as a query to search over the historical event logs. However, the size of the entire historical logs is usually very large, so it is not efficient to go through all event messages. For example, IBM Tivoli Monitoring 6.x [1] usually generates over 100G bytes system events for just one month from 600 windows servers. Searching over such

a large scale event sequence is challenging and the searching index is necessary for speeding up this process. Current system management tools and software can only search a single event by keywords or relational query conditions [1, 3, 2]. However, a system behavior is usually described by several continuous event messages not just one single event, as shown in the above `ntpd` example. In addition, the number of event messages for a system behavior is not a fixed number, so it is hard to decide what is the appropriate segment length for building the index.

Existing search indexing methods for textual data and sequential data can be summarized into two different categories. In our problem, however, each of them has its own limitation. For the textual data, the locality-sensitive hashing (LSH) [14] with the Min-Hash [10] function is a common scheme. But these LSH based methods only focus on unordered data [14, 7, 27]. In a textual event sequence, the order information cannot be ignored since different orders indicate different execution flows of the system. For sequential data, the segment search problem is a substring matching problem. Most existing methods are hash index based, suffix tree based, suffix arrays based or BOWTIE based [15, 23, 18, 5, 19, 9]. These methods can keep the order information of elements, but their sequence elements are single values rather than texts. Their search targets are the matched substrings. In our problem, the similar segments are not necessary to be matched substrings.

1.1 Contributions

To combine both advantages of the locality-sensitive hashing (LSH) and suffix arrays, this paper proposes a method, *suffix matrix*, to search similar segments over textual sequences. This method first creates a set of independent hash functions and maps the textual sequence into a set of hash-value sequences. Then, it constructs a suffix matrix where each row is a suffix array generated from a hash-value sequence. By using binary search over this suffix matrix, this method is able to find out similar segments with a high probability. Meanwhile, it can reduce the collision probability with dissimilar segments. This method can also search k-dissimilar segments. A k-dissimilar segment is a segment which has at most k dissimilar events to the query sequence. By using the *random sequence mask* proposed in this paper, *suffix matrix* is able to maintain a high probability to reach all k-dissimilar segments without increasing much search costs. *suffix matrix* is a systematic integration of LSH and suffix arrays. We conduct experiments on real system log data. The experimental results show that *suffix matrix* outperforms the straightforward combinations of existing techniques.

The rest of the paper is organized as follows: In Section 2, we formulate the problem of this paper. Then, we discuss the straightforward potential solutions using existing methods. Section 3 presents our *suffix matrix* method and theoretically analyzes its performance. In Section 4, we conduct experiments on real system log data and present the empirical results. Section 5 describes the related work. Finally, Section 6 concludes this paper.

2. PROBLEM FORMULATION

Let $S = e_1e_2...e_n$ be a sequence of n event messages, where e_i denotes the i-th event, $i = 1, 2, ..., n$. $|S|$ denotes the length of sequence S, which is the number of events in S. $\mathcal{E}$ denotes the universe of events. $sim(e_i, e_j)$ is a similarity function which measures the similarity between event e_i and event e_j, where $e_i \in \mathcal{E}$, $e_j \in \mathcal{E}$. In this paper, Jaccard coefficient [28] with 2-shingling [11] is utilized as the similarity function $sim(\cdot, \cdot)$ because each event is a textual message.

DEFINITION 1. **(Segment)** *Given a sequence of events $S = e_1...e_n$, a segment of S is a sequence $L = e_{m+1}e_{m+2}...e_{m+l}$, where l is the length of L, $l \leq n$, and $0 \leq m \leq n - l$.*

The problem of this paper is formally stated as follows.

PROBLEM 1. **(Problem Statement)** *Given an event sequence S and a query event sequence Q, find all segments with length $|Q|$ in S which are similar to Q.*

Similar segments are defined based on the event similarity. Given two segments $L_1 = e_{11}e_{12}...e_{1l}$, $L_2 = e_{21}e_{22}...e_{2l}$, we consider the number of dissimilar events in L_1 and L_2. If the number of dissimilar event pairs is at most k, then L_1 and L_2 are similar. This definition is also called k-dissimilar:

$$N_{dissim}(L_1, L_2, \delta) = \sum_{i=1}^{l} z_i \leq k,$$

where

$$z_i = \begin{cases} 1, & sim(e_{1i}, e_{2i}) < \delta \\ 0, & \text{otherwise} \end{cases},$$

and δ is a user-defined threshold for the event similarity. The k-dissimilar corresponds to the well-known k-mismatch or k-error in the subsequence matching problem [20].

2.1 Potential Solutions by LSH

The locality-sensitive hashing (LSH) [14] with the Min-Hash [10] function is a common scheme for the similarity search over texts and documents. LSH is a straightforward solution for our problem. We can consider each segment as a small "document" by concatenating its event messages. Figure 1 shows a textual event sequence $S = e_1e_2...e_{i+1}e_{i+2}...$, where e_i is a textual event. In this sequence, every 4 adjacent event messages are seen as a "document", such as L_{i+1}, L_{i+2} and so on. The traditional LSH with the Min-Hash function can be utilized on these small "documents" to speed up the similar search. This solution is called LSH-DOC as a baseline method in this paper. However, this solution ignores the order information of events, because the similarity score obtained by the Min-Hash does not consider the order of elements in each "document".

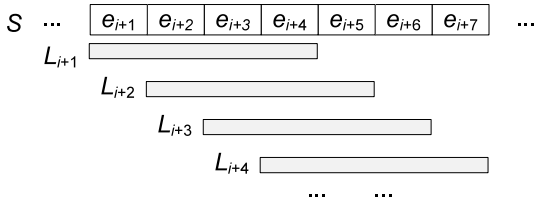


Figure 1: An Example of LSH-DOC

To preserve the order information, we can distribute the hash functions to individual regions of segments. For example, the length of the indexed segment is 4, and we have 40 hash functions. We assign every 10 hash functions to every event in the segment. Then, each hash function can only

be used to index the events from one region of the segment. Figure 2 shows a sequence S with several segments L_{i+1},..., L_{i+4}, where p_1,..., p_4 are 4 regions of each segment and each region contains one event. Every p_j has 10 hash functions to compute the hash values of the contained event, $j = 1, ..., 4$. If the hash signatures of two segments are identical, it is probably that every region's events are similar. Thus, the order information is preserved. This solution is called LSH-SEP as another baseline method in this paper.

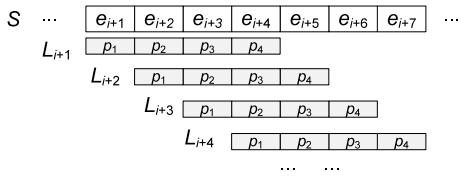


Figure 2: An Example of LSH-SEP

k-dissimilar segments are two segments which contain at most k dissimilar events inside. To search the k-dissimilar segments, a common approach is to split the query sequence Q into $k+1$ non-overlapping segments. If a segment L has at most k dissimilar events to Q, then there must be one segment of Q which has no dissimilar event with its corresponding region of L. Then, we can use any search method for exact similar segments to search the k-dissimilar segments. This idea is applied in many biological sequence matching algorithms [5]. But there is a drawback for the two previous potential solutions: *they all assume that the length of indexed segments l is equal to the length of query sequence $|Q|$.* The query sequence Q is given by the user at runtime, so $|Q|$ is not fixed. However, if we do not know the length of the query sequence Q in advance, we cannot determine the appropriate segment length l for building the index. If $l > |Q|$, none of the similar segments could be retrieved correctly. If $l < |Q|$, we have to split Q into shorter subsegments of length l, and then query those shorter subsegments instead of Q. Although all correct similar segments can be retrieved, the search cost would be large, because the subsegments of Q are shorter than Q and the number of retrieve candidates is thus larger [20]. Figure 3 shows an example

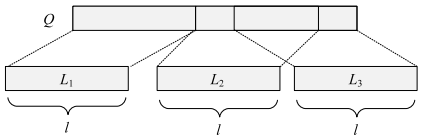


Figure 3: An example of $l < |Q|$

for the case $l < |Q|$. Since the length of indexed segments is l and less than $|Q|$, LSH-DOC and LSH-SEP have to split Q into subsegments L_1, L_2 and L_3, $|L_i| = l$, $i = 1, .., 3$. Then, LSH-DOC and LSH-SEP use the three subsegments to query the segment candidates. If a segment candidate is similar to Q, its corresponding region must be similar to a subsegment L_i, but not vice versa. Therefore, the acquired candidates for L_i must be more than those for Q. Scanning a large number of candidates is time-consuming. Therefore, the optimal case is $l = |Q|$. But $|Q|$ is not fixed at runtime.

3. ALGORITHMS

In this section, we present the suffix matrix method and discuss its theoretical performance.

3.1 Suffix Matrix

3.1.1 Suffix Matrix Indexing

Let h be a hash function from LSH family. h maps an event to an integer, $h : \mathcal{E} \to \mathcal{Z}_h$, where $\mathcal{E}$ is the universe of textual events, and $\mathcal{Z}_h$ is the universe of hash values. In suffix matrix, Min-Hash [10] is the hash function. By taking a Min-Hash function h, a textual event sequence $S = e_1...e_n$ is mapped into a sequence of hash values $h(S) = h(e_1)...h(e_n)$. Suppose we have m independent hash functions, we can have m distinct hash value sequences. Then, we create m suffix arrays from the m hash value sequences respectively. The suffix matrix of S is constructed by the m suffix arrays, where each row is a suffix array.

DEFINITION 2. **(Suffix Matrix)** *Given a sequence of events $S = e_1...e_n$ and a set of independent hash functions $H = \{h_1, ..., h_m\}$, let $h_i(S)$ be the sequence of hash values, i.e., $h_i(S) = h_i(e_1)...h_i(e_n)$. The suffix matrix of S is $\mathbf{M}_{S,m} = [A_1^T, ..., A_m^T]^T$, where A_i^T is the suffix array of $h_i(S)$ and $i = 1, ..., m$.*

We illustrate the suffix matrix by an example as follows:

EXAMPLE 1. *Let S be a sequence of events, $S = e_1e_2e_3e_4$. H is a set of independent hash functions for events, $H = \{h_1, h_2, h_3\}$. For each event and hash function, the computed the hash value is shown in Table 1.*

Table 1: An Example of Hash Value Table

Event	e_1	e_2	e_3	e_4
h_1	0	2	1	0
h_2	3	0	3	1
h_3	1	2	2	0

Let $h_i(S)$ denote the i-th row of Table 1. By sorting the suffixes in each row of Table 1, we could get the suffix matrix $\mathbf{M}_{S,m}$ below.

$$\mathbf{M}_{S,m} = \begin{bmatrix} 3 & 0 & 2 & 1 \\ 1 & 3 & 0 & 2 \\ 3 & 0 & 2 & 1 \end{bmatrix}.$$

For instance, the first row of $\mathbf{M}_{S,m}$: 3021, is the suffix array of $h_1(S) = 0210$.

There are a lot of efficient algorithms for constructing the suffix arrays [15, 23, 18]. The simplest algorithm is sorting all suffixes of the sequence with a time complexity $O(n \log n)$. Thus, the time complexity of constructing the suffix matrix $\mathbf{M}_{S,m}$ is $O(mn \log n)$, where n is the length of the historical sequence and m is the number of hash functions.

3.1.2 Searching over Suffix Matrix

Similar to the traditional LSH, the search algorithm based on a suffix matrix consists of two steps. The first step is to acquire the candidate segments. Those candidates are potential similar segments to the query sequence. The second step is to filter the candidates by computing their exact similarity scores. Since the second step is straightforward and is the same as the traditional LSH, we only present the first step of the search algorithm.

Given a set of independent hash functions $H = \{h_1, ..., h_m\}$ and a query sequence $Q = e_{q1}e_{q2}...e_{qn}$, let $\mathbf{Q}_H = [h_i(e_{qj})]_{m \times n}$, $\mathbf{M}_{S,m}(i)$ and $\mathbf{Q}_H(i)$ denote the i-th rows of $\mathbf{M}_{S,m}$ and $\mathbf{Q}_H$

respectively, $i = 1, ..., m$, $j = 1, ..., n$. Since $\mathbf{M}_{S,m}(i)$ is a suffix array, we obtain these entries that matched with $\mathbf{Q}_H(i)$ by a binary search. $\mathbf{M}_{S,m}$ has m rows, we apply m binary searches to retrieve m entry sets. If one segment appears at least r times in the m sets, then this segment is considered to be a candidate. Parameters r and m will be discussed at a later stage of this section.

Algorithm 1 states the candidates search algorithm. $h(i)$ is the i-th hash function in H. Q_{h_i} is the hash-value sequence of Q mapped by h_i. SA_i is the i-th row of the suffix matrix $\mathbf{M}_{S,m}$, and $SA_i[l]$ is the suffix at position l in SA_i. $CompareAt(Q_{h_i}, SA_i[l])$ is a subroutine to compare the order of two suffixes Q_{h_i} and $SA_i[l]$ for the binary search. If Q_{h_i} is greater than $SA_i[l]$, it returns 1; if Q_{h_i} is smaller than $SA_i[l]$, it returns -1; otherwise, it returns 0. $Extract(Q_{h_i}, SA_i, pos)$ is a subroutine to extract the segments candidates from the position pos. Since H has m hash functions, $C[L]$ records the number of times that the segment L is extracted in the m iterations. The final candidates are only those segments which are extracted for at least r times. The time cost of this algorithm will be discussed in Section 3.3.

Algorithm 1 SearchCandidates (Q, δ)

Parameter: Q : query sequence, δ: threshold of event similarity;
Result: $\mathcal{C}$: segment candidates.

```
Create a counting map C
for i = 1 to |H| do
    Q_{h_i} ← h_i(Q)
    SA_i ← M_{S,m}(i)
    left ← 0, right ← |SA_i| − 1
    if CompareAt(Q_{h_i}, SA_i[left]) < 0 then
        continue
    end if
    if CompareAt(Q_{h_i}, SA_i[right]) > 0 then
        continue
    end if
    pos ← −1
    // Binary search
    while right − left > 1 do
        mid ← ⌊(left + right)/2⌋
        ret ← CompareAt(Q_{h_i}, SA_i[mid])
        if ret < 0 then
            right ← mid
        else if ret > 0 then
            left ← mid
        else
            pos ← mid
            break
        end if
    end while
    if pos = −1 then
        pos ← right
    end if
    // Extract segment candidates
    for L ∈ Extract(Q_{h_i}, SA_i, pos) do
        C[L] ← C[L] + 1
    end for
end for
for L ∈ C do
    if C[L] < r then
        del C[L]
    end if
end for
```

If a segment L of S is returned by the Algorithm 1, we call L is ***reached*** by this algorithm. We illustrate how the binary search works for one hash function $h_i \in H$ by the following example.

Example 2. *Given an event sequence S with a hash function $h_i \in H$, we compute the hash value sequence $h_i(S)$ shown in Table 2. Let the query sequence be Q, and $h_i(Q) =$ 31, where each digit represents a hash value. The sorted*

Table 2: Hash Value Sequence $h_i(S)$

$h_i(S)$	5	3	1	4	3	1	0
Position	0	1	2	3	4	5	6

Table 3: Sorted Suffixes of $h_i(S)$

Index	Position	Hashed Suffix
0	6	0
1	5	10
2	2	14310
3	4	310
4	1	314310
5	3	4319
6	0	5314310

suffixes of $h_i(S)$ are shown in Table 3. We use $h_i(Q) = 31$ to search all matched suffixes in Table 3. In Algorithm 1, by using the binary search, we could find the matched suffix : 310*. Then, the Extract subroutine probes the neighborhood of suffix* 310*, to find all matched suffixes with $h_i(Q)$. Finally, the two segments at position 4 and 1 are extracted. If the two segments are extracted for at least r independent hash functions, then the two segments are the final candidates returned by the Algorithm 1.*

Lemma 1. *Given an event sequence S and a query event sequence Q, L is a segment of S, $|L| = |Q|$, δ_1 and δ_2 are two thresholds for similar events, $0 \le \delta_2 < \delta_1 \le 1$, then:*

- *if $N_{dissim}(L, Q, \delta_1) = 0$, then the probability that L is reached by Algorithm 1 is at least $F(m-r; m, 1-\delta_1^{|Q|})$;*
- *if $N_{dissim}(L, Q, \delta_2) \ge k$, $1 \le k \le |Q|$, then the probability that L is reached by Algorithm 1 is at most $F(m-r; m, 1-\delta_2^k)$,*

where $F(\cdot; n, p)$ is the cumulative distribution function of Binomial distribution $B(n, p)$, and r is a parameter for Algorithm 1.

Proof. Let's first consider the case $N_{dissim}(L, Q, \delta_1) = 0$, which indicates every corresponding events in L and Q are similar and the similarity is at least δ_1. The hash function h_i belongs to the LSH family, so we have $Pr(h_i(e_1) = h_i(e_2)) = sim(e_1, e_2) \ge \delta_1$. L and Q have $|Q|$ events, so for one hash function, the probability that hash values of all those events are identical is at least $\delta_1^{|Q|}$. Once those hash values are identical, L must be found by a binary search over one suffix array in $\mathbf{M}_{S,m}$. Hence, for one suffix array, the probability of L being found is $\delta_1^{|Q|}$. $\mathbf{M}_{S,m}$ has m suffix arrays. The number of those suffix arrays that L is found follows the Binomial distribution $B(m, \delta_1^{|Q|})$. Then, the probability that there are at least r suffix arrays that L is reached is $1 - F(r; m, \delta_1^{|Q|}) = F(m-r; m, 1-\delta_1^{|Q|})$. The second case that $N_{dissim}(L, Q, \delta_2) \ge k$ indicates there are at least dissimilar k events and their similarities are less than δ_2. The probability that hash values of all those events in L and Q are identical is at most δ_2^k. The proof for this case is analogous to that of the first case. □

Lemma 1 is to ensure that if a segment L is similar to the query sequence Q, then it is very likely to be reached by our algorithm; if L is dissimilar to the query sequence Q, then it is very unlikely to be reached. The probabilities shown in this lemma are the false negative probability and the false positive probability. The choice of r controls the tradeoff between the probabilities. F-measure is a combined measurement for the two factors [26]. The optimal r is the one that maximizes the F-measure score. Since r can only be an integer, we can enumerate all possible values of r from 1 to m to find the optimal r.

However, this algorithm cannot handle the case that if there are two dissimilar events inside L and Q. The algorithm narrows down the search space step by step according to each element of Q. A dissimilar event in Q would lead the algorithm to incorrect following steps.

3.2 Randomly Masked Suffix Matrix

Figure 4 shows an example of a query sequence Q and a segment L. There is only one dissimilar event pair between Q "1133" and L "1933", which is the second one, '9' in L with '1' in Q. Clearly, the traditional binary search cannot find "1933" by using "1133" as the query. To overcome

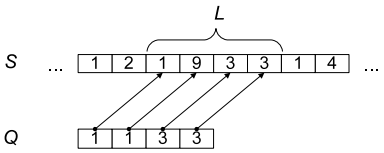


Figure 4: Dissimilar Events in Segments

this problem, a straightforward idea is to skip the dissimilar event between Q and L. However, the dissimilar event can be any event inside L. We do not know which event is the dissimilar event to skip before knowing Q. If two similar segments are allowed to have at most k dissimilar events, the search problem is called the k-dissimilar search. Our proposed method is summarized as follows:

Offline Step:

1. Apply f min-hash functions on the given textual sequence to convert it into f hash-valued sequences.
2. Generate f random sequence masks and apply them to the f hash-valued sequences (one to one).
3. Sort the f masked sequences to f suffix arrays and store them with the random sequence masks to disk files.

Online Step:

1. Apply the f min-hash functions on the given query sequence to convert it into f hash-valued sequences.
2. Load the f random sequence masks and apply them to the f hash-valued query sequences.
3. Invoke f binary searches by using the f masked query sequences over the f suffix arrays and find segment candidates that has been extracted at least r times.

3.2.1 Random Sequence Mask

A sequence mask is a sequence of bits. If these bits are randomly and independently generated, this sequence mask is a random sequence mask.

DEFINITION 3. *A random sequence mask is a sequence of random bits in which each bit follows Bernoulli distribution with parameter θ: $P(bit = 1) = \theta$, $P(bit = 0) = 1 - \theta$, where $0.5 \leq \theta < 1$.*

Figure 5 shows a hash-value sequence $h(S)$ and two random sequence masks: M_1 and M_2. $M_i(h(S))$ is the masked sequence by *AND* operator: $h(S)$ *AND* M_i, where $i = 1, 2$. White cells indicate the events that are kept in $M_i(h(S))$, and dark cells indicate those events to skip. The optimal

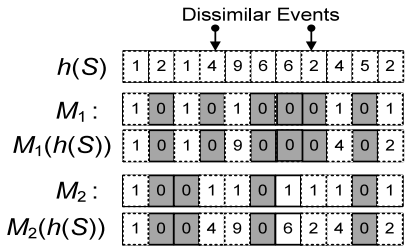


Figure 5: Random Sequence Mask

mask is the one such that all dissimilar events are located in the dark cells. In other words, the optimal mask is able to skip all dissimilar events. We call this kind of random sequence masks as the *perfect* sequence masks. In Figure 5, there are 2 dissimilar events in S: the 4th event and the 8th event. M_1 skips the 4th event and the 8th event in their masked sequences, so M_1 is a *perfect* sequence mask. Once we have a *perfect* sequence mask, previous search algorithms can be applied on those masked hash value sequences without considering dissimilar events.

LEMMA 2. *Given an event sequence S, a query sequence Q, and f independent random sequence masks with parameter θ, let L be a segment of S, $|Q| = |L|$. If the number of dissimilar event pairs of L and Q is k, then the probability that there are at least m perfect sequence masks is at least $F(f - m; f, 1 - (1 - \theta)^k)$, where F is the cumulative probability function of Binomial distribution.*

PROOF. Since each bit in each mask follows the Bernoulli distribution with parameter θ, the probability that the corresponding bit of one dissimilar even is 0 is $1 - \theta$ in one mask. Then, the probability that all corresponding bits of k dissimilar events are 0 is $(1 - \theta)^k$ in one mask. Hence, the probability that one random sequence mask is a *perfect* sequence mask is $(1-\theta)^k$. Then, $F(f-m; f, 1-(1-\theta)^k)$ is the probability for this case happens m times in f independent random sequence masks. □

3.2.2 Randomly Masked Suffix Matrix

A randomly masked suffix matrix is a suffix matrix, where each suffix array is masked by a random sequence mask. We use $\mathbf{M}_{S,f,\theta}$ to denote a randomly masked suffix matrix, where S is the event sequence to index, f is the number of independent LSH hash functions, and θ is the parameter for each random sequence mask. Note that, $\mathbf{M}_{S,f,\theta}$ still consists of f rows by $n = |S|$ columns.

LEMMA 3. *Given an event sequence S, a randomly masked suffix matrix $\mathbf{M}_{S,f,\theta}$ of S and a query sequence Q, L is a segment of S, $|L| = |Q|$. If the number of dissimilar events between L and Q is at most k, then the probability that L is*

reached by Algorithm 1 is at least

$$Pr_{reach} \geq \sum_{m=r}^{f} F(f-m; f, 1-(1-\theta)^k) \cdot F(m-r; m, 1-\delta^{|Q|\cdot\theta}),$$

where δ and r are parameters of Algorithm 1.

This probability combines the two previous probabilities in Lemma 1 and Lemma 2. m becomes a hidden variable, which is the number of *perfect* sequence masks. By considering all possible m, this lemma is proved. Here the expected number of kept events in every $|Q|$ events by one random sequence mask is $|Q| \cdot \theta$.

3.3 Analytical Search Cost

Given an event sequence S and its randomly masked suffix matrix $\mathbf{M}_{S,f,\theta}$, $n = |S|$, the cost of acquiring candidates mainly depends on the number of binary search on suffixes. Recall that $\mathbf{M}_{S,f,\theta}$ is f by n. Each row of it is a suffix array. f binary searches must be executed. Each binary search cost is $\log n$. The total cost of acquiring candidates is $f \log n$.

The cost of filtering candidates mainly depends on the number of candidates acquired. Let $\mathcal{Z}_h$ denote the universe of hash values. Given an event sequence S and a set of hash functions H, $\mathcal{Z}_{H,S}$ denotes the set of hash values output by each hash function in H with each event in S. $\mathcal{Z}_{H,S} \subseteq \mathcal{Z}_h$, because some hash value may not appear in the sequence S. In average, each event in S has $Z = |\mathcal{Z}_{H,S}|$ distinct hash values. Let Q be the query sequence. For each suffix array in $\mathbf{M}_{S,f,\theta}$, the average number of acquired candidates is:

$$N_{Candidates} = \frac{n}{Z^{|Q|\cdot\theta}}.$$

The total number of acquired candidates is at most $f \cdot N_{Candidate}$. A hash table is used to merge the f sets of candidates into one set. Its cost is $f \cdot N_{Candidate}$. To sum up the two parts, given an interleaved suffix matrix $\mathbf{M}_{S,f,\theta}$ and a query sequence Q, the total search cost is

$$Cost_{search} = f \cdot (\log n + \frac{n}{Z^{|Q|\cdot\theta}}).$$

Why the potential solutions are not efficient?
For potential solutions (i.e., LSH-DOC and LSH-SEP) and suffix matrix, the second part of cost is the major cost of the search. Here we only consider the number of acquired candidates to compare the analytical search cost. The average number of acquired candidates by LSH-DOC and LSH-SEP is at least:

$$N'_{Candidates} = \frac{n}{Z^{|Q|/(k+1)}}.$$

When $|Q|\cdot\theta \geq \log_Z f + |Q|/(k+1)$, $f \cdot N_{Candidate} \leq N'_{Candidates}$. Z depends on the number of 2-shinglings, which is approximated to the square of the vocabulary size of log messages. Hence, Z is a huge number, $\log_Z f$ can be ignored. Since $\theta \geq 0.5$, $k \geq 1$, we always have $|Q| \cdot \theta \geq |Q|/(k+1)$. Therefore, the acquired candidates of suffix matrix are less than or equal to those of LSH-DOC and LSH-SEP.

3.4 Offline Parameter Choice

The parameters f and θ balances the search costs and search result accuracy. These two parameters are decided in the offline step before building the suffix matrix. Let $Cost_{max}$ be the search cost budget, the parameter choosing problem is to maximize Pr_{reach} subject to $Cost_{search} \leq Cost_{max}$. A practical issue is that the suffix matrix is constructed in the offline phase, but $|Q|$ and δ can only be known in the online phase. A simple approach to find out the optimal f and θ is looking at the historical queries to estimate $|Q|$ and δ. This procedure can be seen as a *training* procedure. Once the two offline parameters are obtain, other parameters are found by solving the maximization problem. The objective function Pr_{reach} is not convex, but it can be solved by the enumeration method since all tuning parameters are small integers.

The next question is how to determine $Cost_{max}$. We can choose $Cost_{max}$ according to the average search cost curve. Figure 6 shows a curve about the analytical search cost and the probability Pr_{reach}, where $m = \lfloor Cost_{search}/(\log n + \frac{n}{|\mathcal{Z}_{H,S}|^{|Q|\cdot\theta}}) \rfloor$. According to this curve, we suggest users to choose $Cost_{max}$ between 100 and 200, because larger search costs would not significantly improve the accuracy any more.

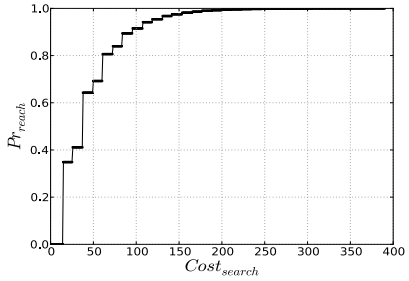


Figure 6: Average Search Cost Curve ($n = 100K, |\mathcal{Z}_{H,S}| = 16, \theta = 0.5, |Q| = 10, \delta = 0.8, k = 2$)

3.5 Scalability

The time complexity of the offline suffix matrix construction is $O(n \log n)$. The online search is $O(\log n)$. The only problem for scaling suffix matrix when the memory cost exceeds the limitation. In this case, the suffix matrix can be stored in the external memory or a distributed system.

4. EVALUATION

In this section, we conduct experiments on real system event logs to evaluate our proposed method.

4.1 Experimental Platform

We implement LSH-DOC, LSH-SEP and our method in Java 1.6. Table 4 summarizes our experimental machine.

Table 4: Experimental Machine

OS	CPU	JRE	JVM Heap Size
Linux 2.6.18	Intel Xeon(R) @ 2.5GHz, 8 core, 64bits	J2SE 1.6	2G

4.2 Data Collection

Our experimental system logs are collected from two different real systems. Apache HTTP error logs are collected from the server machines in the computer lab of a research center and have about 236,055 log messages. Logs of ThunderBird [4] are collected from a supercomputer in Sandia National Lab. The first 350,000 log messages from the ThunderBird system logs are used for this evaluation.

Testing Queries
Each query sequence is a segment randomly picked from

the event sequence. Table 5 lists detailed information about the 6 groups, where $|Q|$ indicates the length of the query sequences. The true results for each query are obtained by the *brute-force* method, which scans through every segment of the sequence one by one to find all true results.

Table 5: Testing Query Groups

Group	Num. of Queries	$\|Q\|$	k	δ
TG1	100	6	1	0.8
TG2	100	12	3	0.65
TG3	100	18	5	0.6
TG4	100	24	7	0.5
TG5	100	30	9	0.5
TG6	100	36	11	0.5

4.3 Baseline Methods

We compare our method with baseline methods LSH-DOC, LSH-SEP stated in Section 2.1. The two methods are both LSH based methods applying to the sequential data. In order to handle the k-dissimilar approximation queries, the indexed segment length l for LSH-DOC and LSH-SEP can be at most $|Q|/(k+1) = 3$, so we set $l = 3$.

4.4 Online Searching

4.4.1 Recall and Search Time

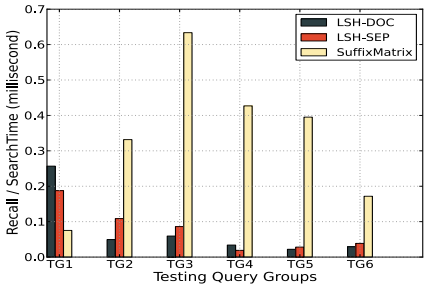
Suffix matrix and LSH based methods all consist of two steps. The first step is to search segment candidates from its index. The second step is filtering acquired candidates by computing their exact similarities. Because of the second step, the precision of the search results is always 1.0. Thus, the quality of results only depends on the recall. By appropriate parameter settings, all the methods can achieve high recalls, but we also consider the associated time cost. For a certain recall, if the search time is smaller, the performance is better. An extreme case is the *brute-force* method that always has the 1.0 recall, but it has to visit all segments of the sequence, so the time cost is huge. We define the recall ratio as a normalized metric for evaluating the goodness of the search results:

$$RecallRatio = \begin{cases} \frac{Recall}{SearchTime}, & Recall \geq recall_{min} \\ 0, & \text{otherwise} \end{cases},$$

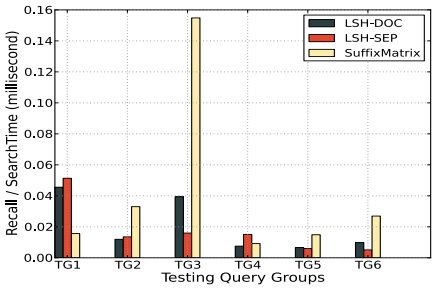
where $recall_{min}$ is a user-specified threshold for the minimum acceptable recall. If the recall is less than $recall_{min}$, the search result is then not acceptable by the user. In our evaluation, $recall_{min} = 0.5$, which means any method should capture at least half of the true results. The unit of the search time is millisecond. *RecallRatio* is expressed as the portion of true results obtained per millisecond. Clearly, *RecallRatio* is higher, the performance is better.

LSH-DOC, LSH-SEP and suffix matrix have different parameters. We vary the value of each parameter in each method, and then select the best performance of each method to compare. LSH-DOC and LSH-SEP have two parameters to set, which are the length of hash vectors b and the number of hash tables t. b varies from 5 to 35. t varies from 2 to 25. We also consider the different number of buckets for LSH-DOC and LSH-SEP. Due to the Java heap size limitation, the number of hash buckets is fixed to be 8000. For suffix matrix, r is chosen according to Section 3.1.2. f and m vary from 2 to 30. θ varies from 0.5 to 1.

Figure 7 shows the *RecallRatios* for each testing group. Overall, suffix matrix achieves the best performance on the two data sets. However, LSH based methods outperform suffix matrix on short queries (TG1). Moreover, in Apache Logs with TG4, LSH-SEP is als better than suffix matrix. To find out the reason why in TG1 suffix matrix performs



(a) ThunderBird Logs



(b) Apache Logs

Figure 7: *RecallRatio* comparison

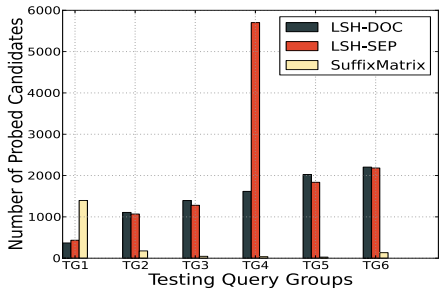
worse than LSH-DOC or LSH-SEP, we record the number of acquired candidates for each method and the number of true results. Figure 8 shows the actual acquired candidates for each testing group with each method. Table 6 shows the numbers of true results for each testing group. From Figure 8, we can see that suffix matrix acquired much more candidates than other methods in TG1. In other words, suffix matrix has a higher collision probability of dissimilar segments in its hashing scheme. To overcome this problem, a common trick in LSH is to make the hash functions be "stricter". For example, there are $d+1$ independent hash functions in LSH family, h_1,...,h_d and h. We can construct a "stricter" hash function $h' = h(h_1(x), h_2(x), ..., h_d(x))$. If two events e_1 and e_2 are not similar, i.e., $sim(e_1, e_2) < \delta$, the collision probability of h_i is $Pr[h_i(e_1) = h_i(e_2)] = sim(e_1, e_2) < \delta$, which can be large if δ is large, $i = 1, ..., d$. But the collision probability of h' is

Table 6: Number of True Results

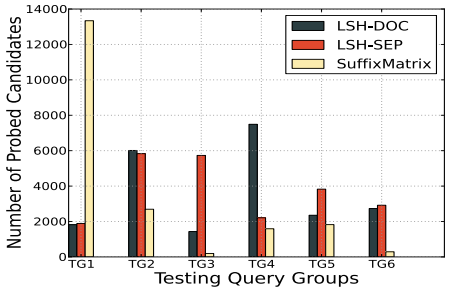
Dataset	TG1	TG2	TG3	TG4	TG5	TG6
ThunderBird Logs	4.12	2.81	27.46	53.24	57.35	7.21
Apache Logs	378.82	669.58	435.94	1139.15	1337.23	990.63

$$\begin{aligned} Pr[h'(e_1) = h'(e_2)] &= \prod_{i=1}^{n} Pr[h_i(e_1) = h_i(e_2))] \\ &= [sim(e_1, e_2)]^d < sim(e_1, e_2). \end{aligned}$$

Figure 9 shows the performance of the suffix matrix by using "stricter" hash functions (denoted as "SuffixMatrix(Strict)") in TG1. Each "stricter" hash function is constructed by 20 independent Min-Hash functions. The testing result shows, "SuffixMatrix(Strict)" outperforms all other methods for both



(a) ThunderBird Logs



(b) Apache Logs

Figure 8: Number of Probed Candidates

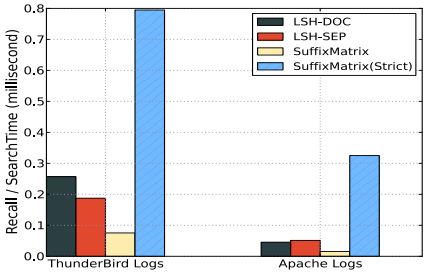


Figure 9: *RecallRatio* **for TG1**

Thunderbird logs and Apache logs in TG1. Table 7 are the parameters and other performance measures of "SuffixMatrix(Strict)". By using "stricter" hash functions, the suffix matrix reduces 90% to 95% of previous candidates. As a result, the search time becomes much smaller than before. The choice of the number of hash functions for a "stricter" hash function, d, is a tuning parameter and determined by the data distribution. Note that the parameters of LSH-DOC and LSH-SEP in this test are already tuned by varying the values of b and t.

Table 7: "SuffixMatrix(Strict)" for TG1

Dataset	**Parameters**	*Recall*	*SearchTime*	Num. of Probed
ThunderBird Logs	$m = 2, \theta = 0.9$	0.9776	1.23 ms	5.04
Apache Logs	$m = 2, \theta = 0.8$	0.7279	2.24ms	152.75

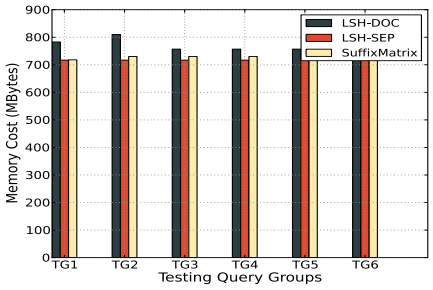
4.4.2 *Parameter and Model Validation*

To verify Lemma 3, we vary each parameter of suffix matrix and test the recall of search results. We randomly sample 100,000 log messages from the ThunderBird logs and randomly pick 100 event segments as the query sequences. The length of each query sequence is 16. Other querying criteria are $k = 5$ and $\delta = 0.5$. Figure 10 shows the recalls by varying each parameter. Figure 10(a) shows that the increase of m will improve the recall. Figure 10(c) verifies that if r becomes larger, the recall will decrease. As for Figure 10(b), since the random sequence masks are randomly generated, the trends of the recall are not stable and a few jumps are in the curves. But generally, the recall curves drop down when we enlarge the θ for the random sequence mask. To sum up, the results shown by Figure 10 can partially verify Lemma 3 proposed in Section 3.

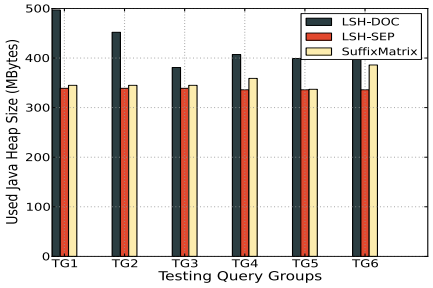
4.5 Offline Indexing

4.5.1 *Space Cost*

Space cost is an important factor for evaluating these methods [9] [15] [13] [20]. If the space cost is too large, the index cannot be loaded into the main memory. To exclude the disk I/O cost for the online searching, we load all event messages and index data into the main memory. The total space cost can be directly measured by the allocated heap memory size in JVM. Note that the allocated memory does not only contain the index, it also includes the original log event messages, 2-shinglings of each event message and the corresponding Java objects information maintained by JVM. We use Java object serialization to compute the exact size of the allocated memory. Figure 11 shows the total used memory size for each testing group. The parameters of each method are the same as in Figure 7. The total space costs for LSH-SEP and suffix matrix are almost the same because they both build the hash index for each event message only once. But LSH-DOC builds the hash indices for each event l times since each event is contained by l continuous segments, where l is the length of the indexed segment and $l = 3$.

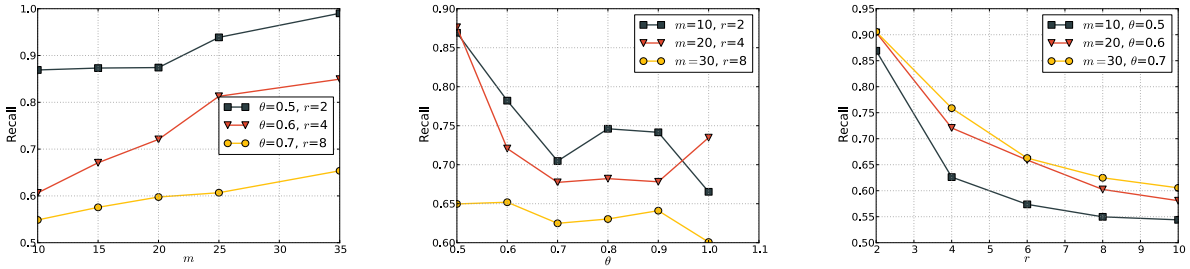


(a) ThunderBird Logs



(b) Apache Logs

Figure 11: Peak Memory Cost

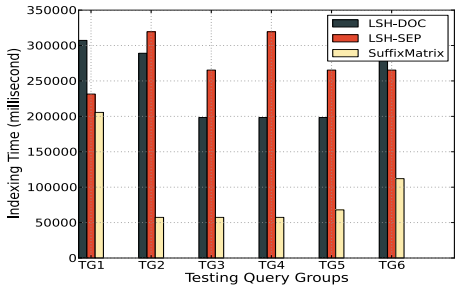


(a) Varying m (b) Varying θ (c) Varying r

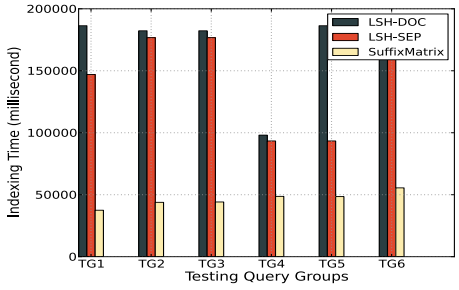
Figure 10: Varying Parameters

4.5.2 *Indexing Time*

Indexing time is the time cost for building the index. Figure 12 shows the indexing time for each method that has the same parameters for Figure 7. The time complexities of LSH-DOC and LSH-SEP are $O(nlbt \cdot c_h)$ and $O(nbt \cdot c_h)$, where n is the number of event messages, l is the indexed segment length, b is the length of the hash vector, t is the number of hash tables, and c_h is the cost of Min-Hash function for one event message. Although for each testing group, the selected LSH-DOC and LSH-SEP may have different b and t, in general LSH-SEP is more efficient than LSH-DOC. The time complexity of suffix matrix for building the index is $O(mn \log n + mn \cdot c_h)$, where m is the number of rows of the suffix matrix. It seems that the time complexity of suffix matrix is bigger than LSH based methods if we only consider n as a variable. However, as shown in Figure 12, suffix matrix is actually the most efficient method in building index. The main reason is $m \ll b \cdot t$. In addition, the time cost of Min-Hash function, c_h, is not small since it has to randomly permute the 2-shinglings of an event message.



(a) ThunderBird Logs



(b) Apache Logs

Figure 12: Indexing Time

5. RELATED WORK

The similarity search problem in low-dimensional data spaces has been studied extensively. A number of tree structure based algorithms are devised to support the similarity queries and nearest neighbors queries, such as R-Tree [16], KD-Tree [8] and SR-Tree [17]. These previous algorithms are known to work well in low-dimensional data spaces. But for high-dimensional data spaces, their search time cost or indexing space cost grows to an exponential number of the dimensionality. In textual information retrieval and image processing domains, the descriptor of a data object is usually a high-dimensional vector. Hence, those tree structured based algorithms are not appropriate in these domains. Locality-Sensitive Hashing (LSH) is a randomized approximate algorithm for the similar search in high-dimensional data space [14, 6]. It is applicable for high dimensional data and has been successfully used in image data or textual data. Min-Hash is a widely used hash function for textual data [10], which can quickly estimate the $sim(x, y)$ of x and y. In natural language processing, a w-shingling is a set of unique contiguous subsequences of words/terms in a document. The similarity function $sim(x, y)$ is usually chosen as the Jaccard similarity over the w-shinglings of x and y.

Substring search in sequential data has been studied for years. Suffix tree and suffix array are two typical methods for on-line searching matched substrings over a sequence [35, 23]. By using a binary search over the suffix array, the method can find matched substring in $O(\log n)$, where n is the length of the string. Compressed suffix arrays and BWT-based compressed full-text indices make further efforts to reduce the search time and space cost based on suffix arrays [15, 12]. Time series data is real-valued sequence data. A lot of efficient similarity search methods are proposed and studied for time series data [25, 21]. But their target is a set of data points, rather than a set of segments of the sequence. Moreover, each data point in time series is a real-valued vector, not a textual message or document.

In system management, log and system event analysis is a fundamental method to maintain, diagnose and optimize large production systems [36, 37, 33, 31, 34, 32]. Log event search as a basic functionality is embedded in many system management, log analysis and system monitoring products [1, 3, 2]. Users can input relational query conditions or a set of keywords to query related system event logs in history. This kind of log search has no difference with a traditional database query or a keywords search. Their search target is single events, not continuous subsequence of events.

6. CONCLUSION

This paper proposes *suffix matrix* to search similar segments over large textual sequences. It combines ideas of locality sensitive hashing and suffix arrays to reduce the search space. By using *random sequence mask* proposed in this paper, this method can have a high probability to reach all k-dissimilar segments without increasing much search costs. We conduct experiments on real system log data and experimental results show that our proposed method outperforms alternative methods using existing techniques.

Acknowledgement

The work is supported in part by US National Science Foundation under grants HRD-0833093, CNS-1126619, and IIS-1213026, by Army Research Office under grant number W911NF-12-1-0431, and by the National Natural Science Foundation of China under Grant No. 61070151.

7. REFERENCES

[1] IBM Tivoli : Integrated Service Management software. `http://www-01.ibm.com/software/tivoli/`.

[2] LogLogic: A real-time log analysis and report generation system. `http://www.splunk.com/`.

[3] Splunk: A commerical machine data managment engine. `http://www.splunk.com/`.

[4] ThunderBird: A supercomputer in Sandia National Laboratories. `http://www.cs.sandia.gov/~jrstear/logs/`.

[5] S. Altschul, W. Gish, W. Miller, E. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[6] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of FOCS*, pages 459–468, Berkeley, CA, USA, September 2006.

[7] M. Bawa, T. Condie, and P. Ganesan. LSH Forest: self-tuning indexes for similarity search. In *Proceedings of WWW*, pages 651–660, 2005.

[8] J. L. Bentley. K-d trees for semidynamic point sets. In *Proceedings of the Sixth Annual Symposium on Computational Geometry (SoCG)*, pages 187–197, Berkeley, California, USA, June 1990.

[9] P. Bieganski, J. Riedl, J. V. Carlis, and E. F. Retzel. Generalized suffix trees for biological sequence data: Applications and implementation. In *Proceedings of HICSS*, pages 35–44, Dallas, Texas, USA, May 1994.

[10] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of STOC*, pages 327–336, Dallas, Texas, USA, May 1998.

[11] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks (CN)*, 29(8-13):1157–1166, March 1997.

[12] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[13] M. Ghodsi and M. Pop. Inexact local alignment search over suffix arrays. In *Proceedings of BIBM*, pages 83–87, Washington, DC, USA, September 2009.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of VLDB*, pages 518–529, Edinburgh, Scotland, UK, September 1999.

[15] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[16] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD conference*, pages 47–57, Boston, Massachusetts, USA, June 1984.

[17] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of ACM SIGMOD conference*, pages 369–380, Tucson, Arizona, USA, May 1997.

[18] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2-4):143–156, 2005.

[19] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10, 2009.

[20] Y. Li, A. Terrell, and J. M. Patel. Wham: A high-throughput sequence alignment method. In *Proceedings of SIGMOD*, 2011.

[21] X. Lian and L. Chen. Efficient similarity search over future stream time series. *TKDE*, 20(1):40–54, 2008.

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

[23] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[24] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *Proccedings of ICDM*, pages 959–964, Pisa, Italy, December 2008.

[25] I. Popivanov. Similarity search over time series data using wavelets. In *Proceedings of ICDE*, pages 212–221, 2002.

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

[27] B. Stein. Principles of hash-based text retrieval. In *Proceedings of SIGIR*, pages 527–534, 2007.

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

[29] L. Tang and T. Li. LogTree: A framework for generating system events from raw textual logs. In *Proceedings of ICDM*, pages 491–500, Sydney, Australia, December 2010.

[30] L. Tang, T. Li, and C.-S. Perng. LogSig: generating system events from raw textual logs. In *Proceedings of CIKM*, pages 785–794, 2011.

[31] L. Tang, T. Li, F. Pinel, L. Shwartz, and G. Grabarnik. Optimizing system monitoring configurations for non-actionable alerts. In *Proceedings of IEEE/IFIP NOMS*, pages 34–42, 2012.

[32] L. Tang, T. Li, F. Pinel, L. Shwartz, and G. Grabarnik. An integrated framework for optimizing automatic monitoring systems in large it infrastructures. In *Proceedings of ACM KDD*, 2013.

[33] L. Tang, T. Li, and L. Shwartz. Discovering lag intervals for temporal dependencies. In *Proceedings of ACM KDD*, pages 633–641, 2012.

[34] L. Tang, T. Li, L. Shwartz, and G. Grabarnik. Recommending resolutions for problems identified by monitoring. In *Proceedings of IEEE/IFIP International Symposium on Integrated Network Management*, pages 134–142, 2013.

[35] P. Weiner. Linear pattern matching algorithms. In *Proceedings of FOCS*, pages 1–11, Iowa City, Iowa, USA, September 1973.

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

[37] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Large-scale system problem detection by mining console logs. In *Proceedings of ACM SOSP*, Big Sky, Montana, USA, October 2009.