Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 177 (2007) 4859–4876
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

A multi-dimensional indexing approach for timestamped event sequence matching

Sanghyun Park [a,*], Jung-Im Won [b], Jee-Hee Yoon [c], Sang-Wook Kim [b]

[a] *Department of Computer Science, Yonsei University, Seoul, Republic of Korea*
[b] *College of Information and Communications, Hanyang University, Republic of Korea*
[c] *Division of Information Engineering and Telecommunications, Hallym University, Republic of Korea*

Received 1 August 2006; received in revised form 30 April 2007; accepted 23 June 2007

Abstract

This paper addresses the problem of timestamped event sequence matching, a new type of similar sequence matching that retrieves the occurrences of interesting patterns from timestamped sequence databases. The sequential-scan-based method, the trie-based method, and the method based on the iso-depth index are well-known approaches to this problem. In this paper, we point out their shortcomings, and propose a new method that effectively overcomes these shortcomings. The proposed method employs an R^*-tree, a widely accepted multi-dimensional index structure that efficiently supports timestamped event sequence matching. To build the R^*-tree, this method extracts time windows from every item in a timestamped event sequence and represents them as rectangles in n-dimensional space by considering the first and last occurring times of each event type. Here, n is the total number of disparate event types that may occur in a target application. To resolve the dimensionality curse in the case when n is large, we suggest an algorithm for reducing the dimensionality by grouping the event types. Our sequence matching method based on the R^*-tree performs with two steps. First, it efficiently identifies a small number of candidates by searching the R^*-tree. Second, it picks out true answers from the set of candidates. We prove its robustness formally, and also show its effectiveness via extensive experiments.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Sequence database; Event sequence; Timestamped event sequence matching; Similar sequence matching; Multi-dimensional index

1. Introduction

A *sequence database* is a set of data sequences, each of which comprises an ordered list of symbols or values [1]. *Similar sequence matching* is an operation that finds sequences similar to that of a query sequence from a sequence database [1,2,7,12]. Similar sequence matching is classified into two categories as follows [7,12]:

[*] Corresponding author. Tel.: +82 2 2123 5714; fax: +82 2 365 2579.
E-mail addresses: sanghyun@cs.yonsei.ac.kr (S. Park), jiwon@hanyang.ac.kr (J.-I. Won), jhyoon@hallym.ac.kr (J.-H. Yoon), wook@hanyang.ac.kr (S.-W. Kim).

0020-0255/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2007.06.020

Event	Timestamp
:	:
CiscoDCDLinkUp	19:08:01
MLMSocketClose	19:08:07
MLMStatusUp	19:08:20
:	:
MiddleLayerManagerUp	19:08:37
TCPConnectionClose	19:08:39
:	:

Fig. 1. Example of a timestamped event sequence in a network environment.

- **Whole matching**: Given N data sequences $S_1, \ldots, S_N$ and a query sequence Q, we find all the data sequences S_i that are similar to Q. Here, we note that the data and query sequences should be of the same length.
- **Subsequence matching**: Given N data sequences $S_1, \ldots, S_N$ and a query sequence Q, we find all the data sequences S_i, whose subsequences are similar to Q, and the offsets in S_i of those subsequences. Here, the data and query sequences are allowed to be of arbitrary lengths.

Since subsequence matching is a generalization of whole matching, it is applicable to real applications wider than whole matching. During the past decades, there have been many research efforts on similar sequence matching in sequence databases. Based on these results, many useful techniques for similar sequence matching have been proposed [1,2,5,7,9,11,14–16,18,21–23].

Wang et al. [19] defined a new type of a similar sequence matching problem that deals with timestamped event sequences. It is called *timestamped event sequence matching*. To exemplify this problem, let us examine a domain of event management in a network environment. Here, *items*, each of which is a pair of (event type, timestamp), are sequentially added into a log file in their chronological order[1] whenever pre-determined events arise. In this situation, a sequence is an ordered list of items. Fig. 1 shows an example of a sequence in a log file.

In such environments, the queries to identify the temporal relationship among events are frequently issued as follows [19].

Query 1. Find all occurrences of CiscoDCDLinkUp that are followed by MLMStatusUp within 20 ± 2 s as well as TCPConnectionClose within 40 ± 3 s.

In Ref. [19], a sequence of items as in Fig. 1 is called a *timestamped event sequence* S. Also, Query 1 above is a query sequence Q of $\langle$(CiscoDCDLinkUp, 0), (MLMStatusUp, 20), (TCPConnectionClose, 40)$\rangle$. The timestamped event sequence matching problem is to find subsequences similar to Q from a timestamped event sequence S. Here, the tolerance is ± 2 for the time interval between CiscoDCDLinkUp and MLMStatusUp and ± 3 for the time interval between CiscoDCDLinkUp and TCPConnectionClose.

The timestamped event sequence matching has two unique properties in comparison with the original similar sequence matching.

- **Property 1**: The timestamp gap among events in a query sequence and the order of events are important to determine the degree of similarity.
- **Property 2**: A subsequence X of a data sequence S is defined as a sequence obtained by removing some items from S. That is, X could be a list of *non-contiguous* items in S.

Previous methods proposed for the original similar sequence matching are not directly applicable to this new problem since they do not consider these two properties. Ref. [19] tackled this problem and proposed

[1] It is rare but permitted that more than one item have the same timestamp in a sequence.

a method that employs a new index structure called an *iso-depth index*. This method uses the concept of a *time window*, a time interval of a fixed size. It extracts *contiguous* subsequences of a predetermined time window size from every possible position of items in a timestamped event sequence, and builds a trie [8] on them. Then, it constructs an iso-depth index by referencing the trie via a depth-first traversal.

The iso-depth index consists of two parts: a set of iso-depth arrays and a set of offset lists. In the trie built as above, there are a set of groups of nodes that contain the events of the same type, which also occur after the same time interval from the starting point of the time window. Each iso-depth array contains multiple entries, each of which has a range of IDs for those nodes belonging to the same group. Also, each offset list is in a one-to-one correspondence with a leaf node in a trie, and has the starting offsets, within a sequence, of the time windows that match the pattern in the path from the root to that leaf node.

The event subsequence matching based on the iso-depth index proceeds as follows. First, it extracts all the items from a query sequence. For each item, it finds the entries matched with the item by searching the iso-depth arrays. For each node belonging to the entries matched with all the items specified in a query sequence, it searches for the starting points of the subsequences within a timestamped event sequence which are matched with a query sequence. This method based on the iso-depth index effectively finds non-contiguous timestamped event subsequences.

In this paper, however, we point out two problems with this approach:

(1) This method performs well when a query specifies the exact values for the time intervals between the first and the following events in a query sequence, i.e., when the tolerance is 0 since it accesses only one iso-depth array from the disk for each event within a query sequence. When a query specifies the value ranges for those time intervals, i.e., when the tolerance is larger than 0 it has to access multiple iso-depth arrays from the disk. Thus, the performance of the method becomes worse as the tolerance gets larger.
(2) When constructing the iso-depth index from a trie, the method assigns IDs (starting with 0, which is for the root) to nodes of the trie in the depth-first traversal fashion. Also, it sorts the entries within an iso-depth array according to those IDs. Thus, such IDs are not changeable once the iso-depth index has been built [19]. Therefore, the method does not allow further insertions and deletions of nodes after constructing a trie [19]. To alleviate this problem, we recommended building a separate trie for newly arrived events. Due to these characteristics, the method is not appropriate for dynamic situations where events continuously arrive into a sequence over time.

Specifically, Ref. [19] evaluated performance for only the cases of the queries for the tolerance of 0. In real applications, however, most queries issued in this form would not find subsequences matched to a query sequence. Even with a small tolerance, this problem would happen. Also, the proper tolerance would differ depending on the types of the subsequent events in a query sequence.

For these reasons, we adopt a new query model that specifies time intervals as forms of value ranges. Each interval is for the allowed time gap between the first and the subsequent events in a query sequence. These ranges are set according to the characteristics of applications as well as occurring events. As the range gets larger, Problem (1) of the previous method becomes more serious. In this research, we identified that Problems (1) and (2) of the previous method are mainly due to the structural characteristics of the iso-depth index. Based on this identification, we propose a novel method that replaces it with a multi-dimensional index. Our method places a time window on every item in a timestamped event sequence. We refer to the part of the event sequence covered by this time window as a *data window*. Next, we represent each data window as a rectangle over n-dimensional space. Here, the dimensionality of n represents the number of possible event types in a given application. Also, the range for each dimension of a rectangle specifies the first and last occurring times of its corresponding event. Because there are a significant number of rectangles extracted from an event sequence, we build an R^*-tree, a widely-accepted multi-dimensional index, for indexing them.

The subsequence matching based on the R^*-tree performs as follows. First, given a query sequence, it forms a rectangle that also reflects the occurring time ranges to all the event types. We refer to the rectangle as a *query rectangle*. By searching the R^*-tree, it retrieves data rectangles that are overlapped with the query rectangle. We call these rectangles *candidate data rectangles*. Finally, for candidate data rectangles, it accesses their corresponding data windows within an event sequence from the disk, and eliminates *false alarms* [1,7] by examining whether they are true answers. When the number of events, n, is large, the problem of the *dimensionality curse* [20] appears, and thus makes the search performance of the R^*-tree degrade significantly [4]. To solve this problem, we propose an effective approach that reduces the number of dimensions via grouping event types.

This paper is organized as follows. Section 2 defines the terminology and notations used throughout subsequent sections, and formulates the problem we are going to solve in this paper. Section 3 briefly reviews the prior methods for the problem of timestamped event sequence matching. Section 4 proposes a novel method for the problem in detail. Section 5 verifies the superiority of the proposed method via performance evaluation with extensive experiments. Finally, Section 6 summarizes and concludes the paper.

2. Problem definition

In this section, we formulate the problem of subsequence matching on timestamped event sequences. Before proceeding, we define the terminology and notations necessary for further presentation.

Definition 1 (*Timestamped event sequence T*). *T*, *a timestamped event sequence*, is a list of pairs of $(e(T_i), ts(T_i))$ $(1 \leqslant i \leqslant n)$ defined as follows:

$$T = \langle (e(T_1), ts(T_1)), (e(T_2), ts(T_2)), \ldots, (e(T_n), ts(T_n)) \rangle$$

where $e(T_i)$ is an event type and $ts(T_i)$ is the timestamp at which $e(T_i)$ has occurred. Also, a pair $(e(T_i), ts(T_i))$ is referred to as the ith item of T and is denoted as T_i. The number of items in T is denoted as $|T|$. The time interval between the first and last events, i.e., $ts(T_n) - ts(T_1)$, is denoted as $\|T\|$.

Hereafter, we call a timestamped event sequence an event sequence for short. Also, we safely assume that all items are listed in ascending order of their timestamps, i.e., $ts(T_i) \leqslant ts(T_j)$ for $i < j$. We can simply handle the situations where this assumption does not hold by sorting the items within an event sequence according to their timestamps. To exemplify the following definitions, we employ $T_{\text{example}}(= \langle (a,0), (b,3), (c,5), (d,6), (e,11), (f,32) \rangle)$.

Definition 2 (*Non-contiguous subsequence T'*). If two event sequences $T(=\langle T_1, T_2, \ldots, T_n \rangle)$ and $T'(= \langle T'_1, T'_2, \ldots, T'_k \rangle)$ satisfy the condition below, T' is referred to as a *non-contiguous subsequence* of T.

- For all $j(= 1, 2, \ldots, k)$, there exists at least one list of subscripts of T, $\langle i_1, i_2, \ldots, i_k \rangle$ that makes $T'_j = T_{i_j}$ satisfied, where $i_j < i_{j+1}$.

Intuitively, T' is a sequence obtained by eliminating some items from T. Hereafter, for simplicity, we call a non-contiguous subsequence a subsequence. $T'_{\text{example}}(= \langle (b,3), (d,6), (e,11) \rangle)$ is one of the non-contiguous subsequences of T_{example} since T'_{example} and T_{example} satisfy the above conditions.

Definition 3 (*Query pattern QP*). *QP*, a *query pattern*, consists of an event list *EL* and a range list *RL* defined below. Each element e_i in *EL* implies the ith event, and each pair $(\min_i, \max_i)$ in *RL* is the time range within which e_i has to occur following the first event.

$$QP.EL = \langle e_1, e_2, \ldots, e_k \rangle$$
$$QP.RL = \langle (\min_1, \max_1), (\min_2, \max_2), \ldots, (\min_k, \max_k) \rangle$$

Here, $\min_1 = \max_1 = 0$ for all queries and $\max_i \leqslant \min_{i+1}$ holds for all $i(= 1, 2, \ldots, k-1)$ [19].

$|QP.EL|$ is the number of events $(=k)$ in *QP.EL*, and $\|QP\|$ is the time interval covered by *QP*, which is $\max_k - \min_1$. Also, $QP.EL_i$ is the ith event in *QP.EL*. We can form an example query pattern as follows: $QP_{\text{example}}.EL = \langle b, d, e \rangle$, $QP_{\text{example}}.RL = \langle (0,0), (2,4), (7,10) \rangle$.

Definition 4 (*Time window and data window*). For effective indexing, we fix the time unit for indexing to the maximum $\|QP\|$ among query patterns used in a target application. We define this unit as a *time window TW*, and denote its size as $\|TW\|$. Also, we define the contiguous subsequences of size $\|TW\|$ extracted from every position of items in an event sequence as *data windows DW*. If we set $\|TW\|$ equal to 10, we can obtain the following six time windows from T_{example}: $TW_1 = \langle (a,0),(b,3),(c,5),(d,6)\rangle$, $TW_2 = \langle (b,3),(c,5),(d,6),(e,11)\rangle$, $TW_3 = \langle (c,5),(d,6),(e,11)\rangle$, $TW_4 = \langle (d,6),(e,11)\rangle$, $TW_5 = \langle (e,11)\rangle$, and $TW_6 = \langle (f,32)\rangle$.

Definition 5 (*Matching of event sequence T' and query pattern QP*). T' and QP are considered to *match* with each other if the following three conditions are all satisfied.

- **Condition 1**: $|T'| = |QP.EL|$.
- **Condition 2**: $e(T'_i) = QP.EL_i$ for all $i (= 1, 2, \ldots, |T'|)$.
- **Condition 3**: $\min_i \leqslant ts(T'_i) - ts(T'_1) \leqslant \max_i$ for all $i (= 1, 2, \ldots, |T'|)$.

For instance, T'_{example} and QP_{example} are said to match each other since the above three conditions are all satisfied.

Definition 6 (*Timestamped event sequence matching*). *Timestamped event sequence matching*, or event sequence matching for short, is the problem of finding the locations of all the subsequences T' that are matched to a query pattern QP from an event sequence T in the disk. Given the example query pattern QP_{example}, the event sequence matching finds the location of T'_{example} from T_{example}.

Here, we define the event sequence matching problem to target just one event sequence in a disk. In real applications, however, multiple event sequences in a database could exist. We can simply handle this situation in two ways: First, we connect all the event sequences into a long one, and then perform event sequence matching. Second, we perform event sequence matching on all the event sequences independently, and then merge all the results into one. So, from now on in this paper, we do not address the case of event sequence matching on multiple event sequences. Table 1 summarizes the notations used throughout the paper.

3. Related work

This section briefly reviews prior methods for processing timestamped event sequence matching. We first describe the sequential-scan-based method and the trie-based method. Then, we explain the method based on the iso-depth index and identify its drawbacks.

Table 1
Notations

Notation	Meaning
T	A timestamped event sequence
T_i	The ith item of T
$e(T_i)$	The event type of the ith item of T
$ts(T_i)$	The timestamp of the ith item of T
$\|T\|$	The number of items in T
$\|\|T\|\|$	The time interval covered by $T (= ts(T_{\|T\|}) - ts(T_1))$
T'	A subsequence of T
QP	A query pattern
$\|\|QP\|\|$	The time interval covered by QP
$QP.EL$	An event list of QP
$\|QP.EL\|$	The number of events in the event list of QP
$QP.EL_i$	The ith event in the event list of QP
$QP.RL$	A range list of QP
TW	A time window
$\|\|TW\|\|$	The size of a time window
DW	A data window

3.1. Sequential-scan-based method

As a straightforward method for event sequence matching, we sequentially read all the items in an event sequence accessed from the disk and find subsequences matched with a query pattern. We call this method a *sequential-scan-based method*. It is the simplest approach, however, it accesses the entire event sequence from the disk. Also, it requires much CPU overhead since it compares every possible non-contiguous subsequence within an event sequence with the query pattern. Thus, this method is not appropriate in a large database environment.

3.2. Trie-based method

The trie is a useful data structure designed to find contiguous subsequences that are exactly the same as a query sequence [8]. For event sequence matching, the trie-based method first constructs a trie on all the data windows extracted from every position of items within an event sequence. Unlike the case of ordinary strings, pairs of ⟨event type, timestamp⟩ appear in an event sequence. So, in this case, the trie should regard each pair as one symbol. For this, Ref. [19] applies a function f defined below to each subsequence T and encodes it into a one-dimensional sequence S:

$$\begin{aligned}&\mathsf{f}(\langle T_1, T_2, \ldots, T_k\rangle) = (S_1, S_2, \ldots, S_k)\\&\text{where } S_i = e(T_i)_0 \quad \text{when } i = 1\\&\qquad\quad S_i = e(T_i)_{ts(T_i)-ts(T_{i-1})} \quad \text{when } i > 1\end{aligned}$$

For example, if we extract data windows with $\|TW\| = 4$ from an original sequence $\langle(e, 41), (a, 45), (d, 47), (c, 48), (a, 49), (b, 50)\rangle$, we obtain six data windows of $\langle e0, a4\rangle$, $\langle a0, d2, c1, a1\rangle$, $\langle d0, c1, a1, b1\rangle$, $\langle c0, a1, b1\rangle$, $\langle a0, b1\rangle$, $\langle b0\rangle$. Then, the method builds a trie on these data windows. In the leaf nodes, it also stores the offsets where data windows start within an event sequence.

In timestamped event sequence matching, it first visits the root node of the trie, and then traverses the paths matched with a query pattern QP in a depth-first fashion. We note that all the non-contiguous subsequences in each path should be compared with the query pattern. In this way, when a subsequence in a path from the root to an internal node V is matched with a query pattern, it returns the offsets that are stored in all the leaf nodes under the node V.

This method performs effectively when tolerances for events are specified as a constant rather than a range. However, when they are given as a form of ranges, the performance degrades considerably since it accesses the same path in a trie multiple times. Also, the trie is not appropriate to integrate itself with DBMS seamlessly since it has some difficulty in paginating its structure [17,19].

On the other hand, the indexing method proposed in this paper employs an R^*-tree as a basic index structure which has been successfully integrated with DBMSs. Furthermore, when tolerances for time intervals in a user query become larger than 0, the proposed method just enlarges the corresponding query rectangle and thus has no problem accessing the same data pages more than once.

3.3. Iso-depth index based method

The iso-depth index [19] is a different representation of a trie, and successfully overcomes drawbacks of the trie in event sequence matching. The iso-depth index is composed of two parts: a set of iso-depth arrays and a set of offset lists. Its construction from a given trie is summarized as follows. Each node V in a trie is assigned a pair (V_s, V_m), where V_s is the ID of a node V, and V_m is the largest one among IDs of V's descendent nodes. Because the IDs for nodes in a trie are given sequentially in a depth-first fashion, the ID of any descendent of V always has a value between V_s and V_m. Next, for every pair of an event type x and a time interval $d(= 1, 2, \ldots, \|TW\|)$, its iso-depth array is constructed. The array is for the cases where an event of a type x appears after the first event within time d in data windows, and it stores (V_s, V_m) of the nodes corresponding to such events as its entries. Also, the information in all the leaf nodes is stored in a set of offset lists.

Figs. 2 and 3 show a process of constructing an iso-depth index on an event sequence $T(=\langle(e,41),(a,45),(d,47),(c,48),(a,49),(b,50)\rangle)$. Fig. 2 depicts linking of nodes in a trie that belong to a same entry (x,d) in an iso-depth index after assigning IDs to all the nodes in a trie. For instance, the entry of $(a,4)$ indicates the situation where the event of type a follows the first event after the time interval 4, and thus it points to the nodes of (5, 5) and (15, 15) in the trie. Fig. 3 shows the iso-depth arrays and offset lists built from Fig. 2. They are stored in disk as a form of a sequential file or a B^{+}-tree index.

The subsequence matching based on the iso-depth index performs quite similarly to the one based on the trie. However, the big difference is that it can find the nodes which are matched with the events within a query pattern QP directly from the iso-depth arrays without the trie traversal. We note that a node related to an event in QP should be a descendant of the node related to its previous event in QP. With this approach, when the final nodes matched with QP are identified, the offset information in the leaf nodes can be obtained by examining the corresponding offset lists.

This method performs best when the time intervals between the first and subsequent events in a query pattern are given as exact values, i.e., the tolerance of 0. When they are given as value ranges, however, it performs poorly since a larger number of entries in the iso-depth index need to be examined. Also, due to the characteristics of the iso-depth index, it is not properly applicable to a dynamic environment where events continuously arrive to a sequence.

Compared to the method based on the iso-depth index, the proposed method is less affected by the number of events and tolerance values specified in a query pattern, as demonstrated in Section 5. In addition, when

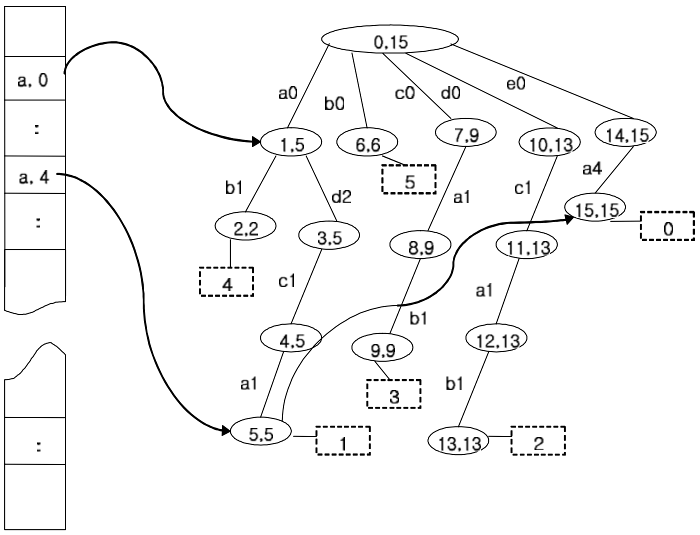

Fig. 2. Linking of nodes in a trie that belong to a same entry (x,d) in an iso-depth index.

(x, d)	(V_s, V_m)
(a, 0)	(1, 5)
(a, 1)	(8, 9)
(a, 2)	(12, 13)
(a, 4)	(5, 5), (15, 15)
(b, 0)	(6, 6)
(b, 1)	(2, 2)
⋮	⋮

V_s	Offset
2	4
5	1
6	5
9	3
13	2
15	0

Fig. 3. Iso-depth arrays and offset lists built from Fig. 2.

new events arrive at the system, we just insert new data rectangles to an existing R^*-tree. Therefore, our method is relatively well suited to a dynamic environment.

4. Proposed method

This section proposes a new indexing method that overcomes the problems of the aforementioned approaches, and suggests an algorithm that uses the proposed indexing method for event sequence matching.

4.1. Indexing

The proposed index construction algorithm is given in Algorithm 1. To support efficient event sequence matching, we first build an empty R^*-tree (Line 1). The dimensionality of the tree is equal to the total number of disparate event types that can occur in a target application. Next, we extract a time window at every possible position of an event sequence (Line 3). The maximum one among the time intervals covered by query patterns in a target application is set as the maximum size of a time window, ξ. The time window TW starting at the ith position of T is produced by Algorithm 2. After generating TW, we construct a rectangle from TW (Line 4) and insert it into the R^*-tree using i as its identifier (Line 5). Finally, we return the R^*-tree containing all rectangles (Line 6).

Algorithm 1. Index construction

```
   Input  : Event sequence $T$, maximum size of time window $\xi$
   Output : $R^*$-tree $I$
1 $I$ := createEmptyRstarTree();
2 for ($i = 1$; $i \leqslant |T|$; $i$++) do
3     $TW$ := extractTimeWindow($T$, $i$, $\xi$);
4     $DR$ := constructDataRectangle($TW$);
5     insertRectangle($I$, $DR$, $i$);
6 return $I$;
```

Let $E = \{E_1, E_2, \ldots, E_n\}$ denote a set of n distinct event types. The rectangle from the time window TW is expressed as $([\min_1, \max_1], [\min_2, \max_2], \ldots, [\min_n, \max_n])$. Here, $\min_i (i = 1, 2, \ldots, n)$ denotes the difference between $ts(TW_1)$ and $first\_ts(E_i)$ where $ts(TW_1)$ is the timestamp of the first item of TW and $first\_ts(E_i)$ is the timestamp of the first occurrence of event type E_i in TW. Similarly, $\max_i$ denotes the difference between $ts(TW_1)$ and $last\_ts(E_i)$ where $last\_ts(E_i)$ is the timestamp of the last occurrence of event type E_i in TW. We assign ξ to both $\min_i$ and $\max_i$ when there are no instances of event type E_i in TW.

Algorithm 2. Extraction of time window at the ith position

```
   Input  : Event sequence $T$, position $i$, maximum size of time window $\xi$
   Output : Time window $TW$
1 $TW$ := createEmptyWindow();
2 insertItem($TW$, $T_i$);
3 for ($j = i + 1$; $j \leqslant |T|$; $j + +$) do
4     if ($ts(T_j) - ts(T_i) \leqslant \xi$) then
5         insertItem($TW$, $T_j$);
6     else
7         break;
8 return $TW$;
```

Example 1. Let us consider an application where the total number of disparate event types is 5 (i.e., $n = 5$) and the size of the time window is 20 s (i.e., $\xi = 20$). If the time window TW beginning at the 100th position on an event sequence is $\langle(E_3,12),(E_4,15),(E_3,21),(E_4,22),(E_5,30)\rangle$, then the rectangle ([20, 20], [20, 20], [9, 9], [3, 10], [18, 18]) is constructed and then inserted into the R^*-tree with 100 as its identifier. The detailed procedure to construct the rectangle is as follows:

(1) Since there are no occurrences of event types E_1 and E_2 in TW, the size of the time window (i.e., ξ (=20)) is assigned to $\min_1$, $\max_1$, $\min_2$, and $\max_2$.
(2) There is only one occurrence of event type E_3 after the occurrence of the first event of TW and their time interval is 9(=21 − 12). Therefore, 9 is assigned to both $\min_3$ and $\max_3$.
(3) Just like the case of event type E_3, there is only one occurrence of event type E_5 after the occurrence of the first event of TW. Since their time interval is 18(=30 − 12), both $\min_5$ and $\max_5$ are assigned 18.
(4) There are two occurrences of event type E_4 after the occurrence of the first event of TW; one is after 3(=15 − 12) s and the other is after 10(=22 − 12) s. Therefore, 3 and 10 are assigned to $\min_4$ and $\max_4$, respectively.

In practice, we maintain multiple R^*-trees in order to reduce the search space before starting an index search. More specifically, we insert the rectangle for the time window TW into an R^*-tree I_j when the first item of TW is of event type E_j. Therefore, the proposed index structure consists of n R^*-trees, $I_1, I_2, \ldots, I_n$, and an index table (see Fig. 4). The index table consists of n entries, each of which stores the physical location of the corresponding R^*-tree. Since each entry of the index table occupies sizeof(eventType) + sizeof(pathName) + sizeof(fileName), which is less than 1 Kbyte in most cases, the total size of the index table is not larger than n Kbytes. Therefore, even in an extreme case where n is 1000, the index table requires less than 1 Mbyte and thus the entire index table can be kept in main memory.

4.2. Event sequence matching

As explained in Section 2, a query pattern QP with k events has the following format: $QP.EL = \langle e_1, e_2, \ldots, e_k\rangle$, $QP.RL = \langle[\min_1,\max_1],[\min_2,\max_2],\ldots,[\min_k,\max_k]\rangle$. This query pattern requires that the events should occur in the order of $e_1, e_2, \ldots, e_k$, and there should be an event e_i within the time range of $[\min_i,\max_i]$ after the occurrence of the first event e_1. The event sequence matching algorithm that uses the proposed index structure is given in Algorithm 3.

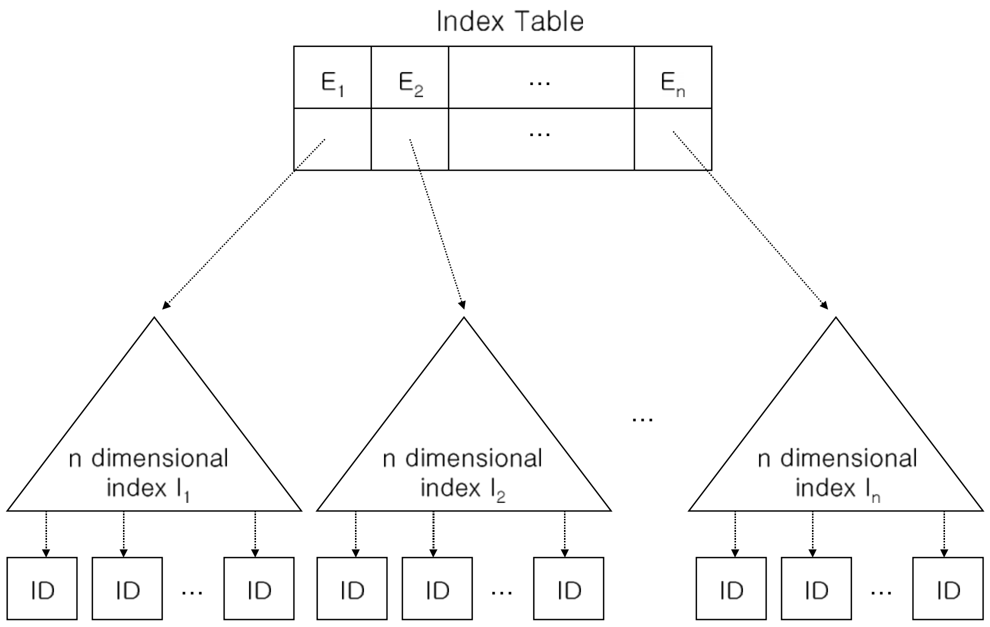

Fig. 4. Overall index structure that consists of an index table and n indexes.

Algorithm 3. Event sequence matching

```
    Input  : Query pattern $QP = (\langle e_1, \ldots, e_k\rangle, \langle[\min_1, \max_1], \ldots, [\min_k, \max_k]\rangle)$,
  Target index $I$, Event sequence $T$
    Output: Set of answers $A$
  1 $A := \{\}$;
  2 $QR :=$ constructQueryRectangle($QP$);
  3 $C :=$ findOverlappingRectangles($I$, $QR$);
  4 for (each identifier $id \in C$) do
  5     if (isTrueAnswer($T$, $id$)) then
  6         addAnswer($A$, $id$);
  7 return $A$;
```

We first construct a query rectangle from a given query pattern (Line 2). The minimum and maximum values of each dimension of a query rectangle are obtained from the corresponding time ranges specified in a query pattern.

(1) If there are no time ranges specified for the event type E_i in a query pattern, then the range of the ith dimension becomes $[0, \xi]$.
(2) If there is only one time range specified for the event type E_i in a query pattern, then the minimum and maximum values on that time range become the minimum and maximum values of the ith dimension, respectively.
(3) If there is more than one time range specified for the event type E_i in a query pattern, then the time range with the smallest interval determines the range of the ith dimension.

Example 2. Let us consider the query pattern QP in the application with $n = 5$ and $\xi = 20$ s: $QP.EL = \langle E_3, E_2, E_1, E_2, E_3\rangle$, $QP.RL = \langle[0, 0], [3, 5], [5, 7], [8, 11], [15, 20]\rangle$. This query pattern dictates that 5 events should occur in the order of E_3, E_2, E_1, E_2, and E_3. It also demands that the intervals between the first and second events, between the first and third events, between the first and fourth events, and between the first and fifth events should be within the time ranges of [3, 5], [5, 7], [8, 11], and [15, 20], respectively. The detailed procedure to construct the query rectangle from this query pattern is as follows:

(1) For event type E_1, there is only one constraint, with 5 as the minimum and 7 as the maximum. Therefore, the first dimension of the query rectangle becomes [5, 7].
(2) For event type E_2, there are two constraints, one with [3, 5] and the other with [8, 11]. Between these two constraints, [3, 5] has a smaller interval than [8, 11] (i.e., 5–3 < 11–8) and thus the second dimension of the query rectangle becomes [3, 5].
(3) For event type E_3, there is only a single constraint [15, 20] and therefore the third dimension of the query rectangle is [15, 10].
(4) Since there are no constraints on E_4 and E_5, the ranges of both the fourth and fifth dimensions are $[0, 20] (= [0, \xi])$.

As a result, the query rectangle for the above query pattern QP is represented as ([5, 7], [3, 5], [15, 20], [0, 20], [0, 20]).

After constructing the query rectangle from a query pattern, we search the multi-dimensional index I for the rectangles overlapping with the query rectangle (Line 3). As mentioned in Section 4.1, each query pattern has an associated R^*-tree according to the type of its first event. Therefore, the search is performed only on the R^*-tree dedicated to the type of e_1. Its location is obtained by looking up the index table. The post-processing step begins after obtaining a set of candidate rectangles from the index search. Using the identifier of each candidate rectangle, this step reads the event sequence to verify whether the candidate actually matches the query pattern (Line 5). Only the candidate rectangles that are actually matched with the query pattern are added into the result set (Line 6) and then returned to the user (Line 7). The following theorem shows that the proposed matching algorithm retrieves, without the risk of false dismissals, all the time windows that match a query pattern. Here, a false dismissal [1,7] is defined as missing the time windows that actually match a query pattern.

Theorem 1. *For any time window TW and query pattern QP, if TW matches QP then the rectangle for TW overlaps the rectangle for QP. In a formal manner,* $\forall_{TW,QP}$ (match(TW,QP) $\Rightarrow$ overlap(*rectangle of TW*, *rectangle of QP*)).

Proof. Let DR be the rectangle for time window TW and QR be the rectangle for query pattern QP. DR overlaps QR only when the range of every dimension of DR overlaps the range of the corresponding dimension of QR. Let us consider the ith dimension arbitrarily. The query pattern QP may have no constraints, only one constraint, or more than one constraint on the event type E_i.

- **Case 1 – no constraints on** E_i: When QP does not impose any constraint on E_i, the ith dimension of QR has the range $[0, \xi]$. The ith dimension of DR always has the range contained in $[0, \xi]$. Therefore, their ranges overlap each other.
- **Case 2 – only one constraint on** E_i: Let QP have the sole constraint $[\min_i, \max_i]$ on E_i. If TW matches QP, then TW must have at least one occurrence of E_i within the time range of $[\min_i, \max_i]$ after the occurrence of its first event. In this case, the range $[\min'_i, \max'_i]$ of the ith dimension of DR certainly satisfies the following condition: "$\min'_i \leqslant \max_i$ AND $\max'_i \geqslant \min_i$". Therefore, their ranges overlap each other.
- **Case 3 – more than one constraint on** E_i: In this case, only the constraint with the range of the smallest interval determines the range of the ith dimension of QR. This makes Case 3 become identical to Case 2.

It is now evident that, when TW matches QP, the range of the ith dimension of DR overlaps that of QR in all cases. The proof on the ith dimension can be generalized to all the other dimensions and thus Theorem 1 has been proved. □

4.3. Dimensionality reduction

The proposed index becomes very high dimensional when n is large. To prevent the problem of *dimensionality curse* [20] in this case, we apply *event type grouping* that classifies n event types into a smaller number, say m, of event type groups. The composition of m event type groups from n event types enables the proposed n-dimensional index to be m-dimensional ($m \ll n$). Such event type grouping, however, tends to enlarge the rectangles to be stored in the index, thereby diminishing the filtering effect in searching. Therefore, in this paper, we suggest a systematic algorithm (See Algorithm 4) that performs event type grouping with the least enlargement of the rectangles. For this algorithm, we define a distance metric that measures the degree of enlargement incurred by merging each pair of event types.

Algorithm 4. Event type grouping

```
  Input: Set of n event types $E = \{E_1, E_2, \ldots, E_n\}$, Event sequence $T$,
         Number of desired event type groups $m$
  Output: Set of $m$ event type groups $G$
1 Using $T$, compute the distance of every pair of event types;
2 Construct a complete graph with $n$ nodes and $n(n-1)/2$ edges;
3 Attach a unique label to each node using the integer values from 1 to $n$. The node labeled with the integer $i$
  represents the event type $E_i$;
4 Attach a weight to each edge. The distance between $E_i$ and $E_j$ is used as a weight of the edge connecting
  the node $i$ and the node $j$;
5 while the number of connected components in the graph is less than $m$ do
    Remove from the graph the edge with the largest weight;
6 Extract $G$, a set of $m$ event type groups, from $m$ connected components;
7 Return $G$;
```

Given any two event types E_i and E_j, their distance is determined by the difference between the average size of all the data rectangles before merging E_i and E_j and after merging E_i and E_j. Let $[\min_i, \max_i]$ and $[\min_j, \max_j]$ be the ranges of the ith and jth dimensions of a data rectangle, respectively. If the ith and jth dimensions are merged into one, $[\min_i, \max_i]$ and $[\min_j, \max_j]$ are also merged into a single range $[\mathsf{MIN}(\min_i, \min_j), \mathsf{MAX}(\max_i, \max_j)]$. Given a time window TW, let $\mathsf{Increment}_i(TW, ij)$ and $\mathsf{Increment}_j(TW, ij)$ denote the increments of the ranges of the ith and jth dimensions of TW, respectively, resulting from merging the ith and jth dimensions. Then the distance between E_i and E_j within TW is defined as follows:

$$D(E_i, E_j, TW) = \frac{\mathsf{Increment}_i(TW, ij) + \mathsf{Increment}_j(TW, ij)}{2}$$

By generalizing the above expression to all time windows in the event sequence, we obtain the following expression that computes the distance between two event types (E_i, E_j). Here, $\#timeWindows$ denotes the total number of time windows in the event sequence.

$$D(E_i, E_j) = \sum_{TW} \frac{D(E_i, E_j, TW)}{\#timeWindows}$$

Example 3. Let us compute the distance between E_4 and E_5 within the time window whose data rectangle is ([20, 20], [20, 20], [9, 9], [3, 10], [18, 18]). The range of the 4th dimension is [3, 10]; therefore, its size is $10 - 3 = 7$. The range of the 5th dimension is [18,18]; therefore, its size is $18 - 8 = 0$. When we merge these two dimensions, we have the range of $[\mathsf{MIN}(3, 18), \mathsf{MAX}(10, 18)] = [3, 18]$ for the merged dimension. Therefore, its size in the merged dimension becomes $18 - 3 = 15$. The size increments of the ranges of the 4th and 5th dimensions are $8(=15 - 7)$ and $15(=15 - 0)$, respectively. Therefore, $D(E_4, E_5, TW) = (8 + 15)/2 = 11.5$.

After computing the distance of every pair of event types, we draw a weighted complete graph [13] with n nodes. We attach a unique label to each node using the integer values from 1 to n. The node labeled with the integer i represents the event type E_i. We then attach a weight $D(E_i, E_j)$ to each edge connecting the node i and the node j. We delete from this graph an edge with the largest weight successively until the graph is partitioned into m components. Finally, we have m connected components as event type groups. Note that, although our indexing method employs the proposed grouping method for dimensionality reduction, the other grouping methods such as Ascendant Hierarchical Classification [6], k-means [10] and k-medoids [10] can also be combined with our indexing method for the same purpose.

5. Performance evaluation

This section verifies the superiority of the proposed method through an extensive performance evaluation. Section 5.1 describes the environment for the experiments, and Section 5.2 shows and analyzes the results.

5.1. Experimental environment

Each item in the event sequence T is a pair of *event type* and *timestamp*, and both event type and timestamp require 4-byte integer space each. Therefore, the space required for storing the entire event sequence T is $8|T|$ bytes. For the experiments, we generated synthetic event sequences with various $|T|$ and n values. Within the range of $[1, n]$, the event types were generated according to either a uniform or zipf distribution, and their interarrival times followed an exponential distribution.

The queries with three event items were used as a standard query pattern. $QP.EL_i$, the event types for the query patterns, also followed the uniform distribution, and $QP.RL_i$, the tolerance ranges of the event types, varied according to the purpose of each experiment. We produced 100 queries for an individual experiment, executed each query 100 times, and computed their average elapsed time. The machine for the experiments was a personal computer with a Pentium IV 2 GHz CPU, main memory of 512 MB, and a Windows 2000 operating system.

We compared the performances of the following three methods: (1) the proposed method that employs the R^*-tree [3] whose page size is 1 KB, (2) the sequential-scan-based method, and (3) the method based on the iso-depth index [19].

5.2. Results and analysis

5.2.1. Experiment 1: index size

In the first experiment we compared our method and the iso-depth index in terms of the index size. Regardless of the number of event types, the proposed method produced five event type groups by performing the event type grouping, and then constructed a five-dimensional R^*-tree. On the other hand, the iso-depth index was comprised of the iso-depth arrays and the offset lists stored in a sequential file structure with a B^+-tree index.

Using data items generated according to a uniform distribution, Fig. 5 shows the sizes of the proposed index and the iso-depth index with increasing data set sizes. The X-axis is for the number of items in the event sequence, and the Y-axis is for the index sizes when the event sequence has 2.5 million items (20 Mbytes), 5 million items (40 Mbytes), or 10 million items (80 Mbytes). The size of the time window $\|TW\|$ was fixed at 50, but the number of event types n was 20 at first and then changed to 80.

As clearly shown in the figure, both the proposed index and the iso-depth index grow linearly as the data set becomes larger. It is also evident that the number of event types only slightly affects the size of the proposed index but noticeably affects the size of the iso-depth index. This is because the iso-depth array, whose entry has the structure (event type, timestamp), becomes larger as the number of event types increases. On the contrary, the size of the proposed index is almost constant due to the event type grouping.

Using the data set with 2.5 millon items (20 Mbytes) generated by a zipf distribution, Fig. 6 shows the sizes of the proposed index and the iso-depth index with increasing time window sizes. The number of event types, n, was 20 at first and then changed to 80. The figure indicates that the iso-depth index becomes significantly larger as the time window becomes longer. This is because the number of arrays in the iso-depth index increases as the time window gets longer. The proposed index, however, becomes slightly smaller when the time window gets longer. This can be interpreted as follows: when the time window gets longer, the signatures of adjacent data windows become similar. This makes the data rectangles from adjacent data windows be located closely in the index space and thus enables the R^*-tree to organize MBRs (minimum bounding rectangles) in a more efficient way. The change patterns of the index size caused by the increase in the number of event types are almost the same as those shown in Fig. 5.

5.2.2. Experiment 2: query processing time with various tolerances, data set sizes, and time window sizes

Experiment 2 compared the query processing times of the three methods. First, Fig. 7 shows the query processing times with various tolerance values of query patterns. The data set had 5 million items generated from

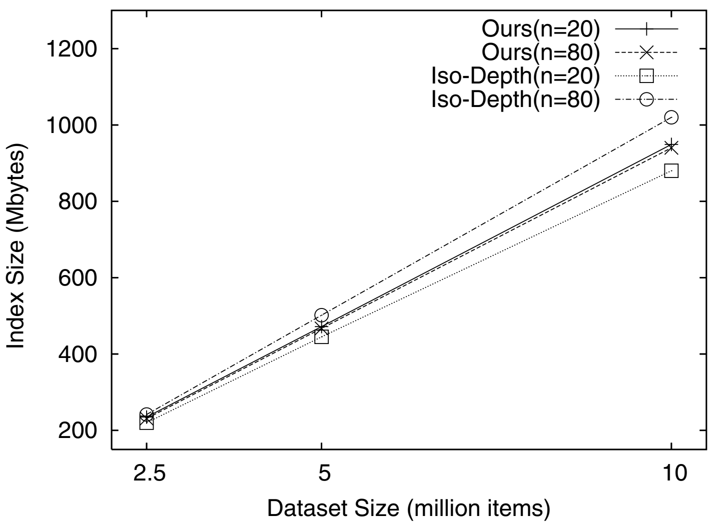

Fig. 5. The sizes of the proposed index and the iso-depth index with increasing data set sizes.

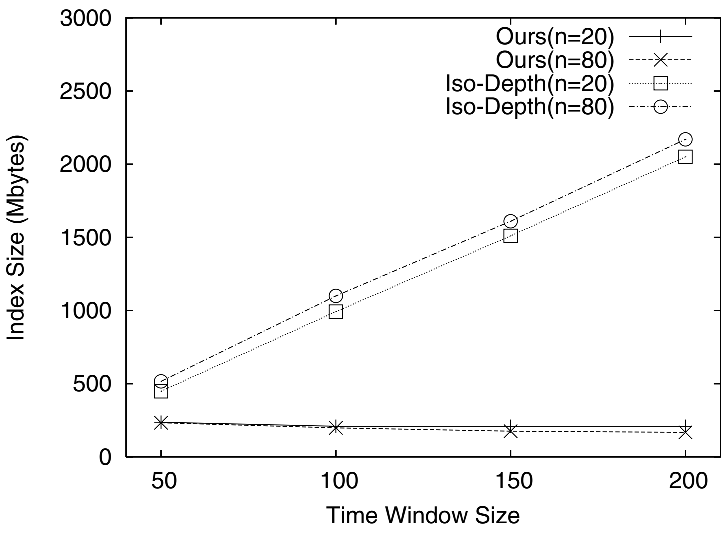

Fig. 6. The sizes of the proposed index and the iso-depth index with increasing time window sizes.

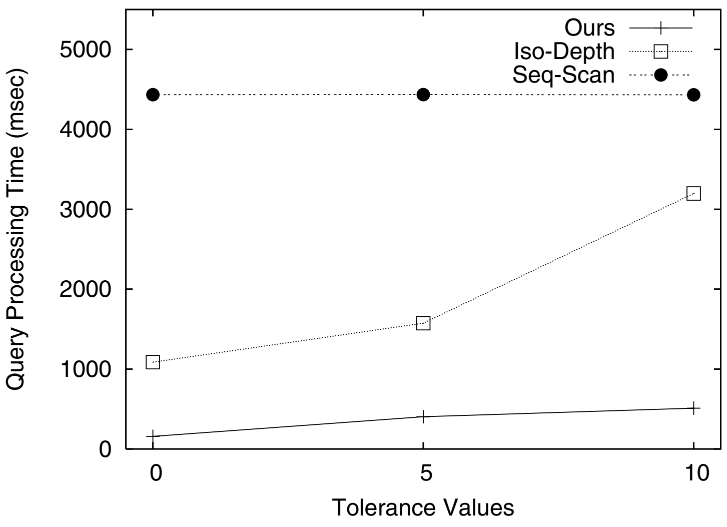

Fig. 7. Query processing times of the three methods with various tolerance values of query patterns.

a uniform distribution. The number of event types was 20 and the size of the time window was 50. The tolerance values were set to 0% (=0), 10% (=5), and 20% (=10) of a time window size. The result is shown in Fig. 7. The *X*-axis represents the tolerance values and the *Y*-axis represents the query processing times.

Since the sequential-scan-based method reads the entire event sequence from the disk in all cases, its query processing time does not change much with the tolerance values. On the contrary, the query processing times of our method and the iso-depth index noticeably increase as the tolerance value grows. This is because both methods have to access more portions of the indexes when the tolerance value becomes larger. As compared to the sequential-scan-based method, our method is about 11 and 8.6 times faster when the tolerance values are 5 and 10, respectively. As compared to the iso-depth index, our method is about 3.9 and 6.3 times faster when the tolerance values are 5 and 10, respectively.

Next, we compared the query processing times of the three methods with various sizes of a data set generated by the uniform distribution. The size of a time window was 50, and the tolerance value of query patterns was 5 (i.e., 10% of a time window size). The number of event types was 20 at first and then changed to 80. The

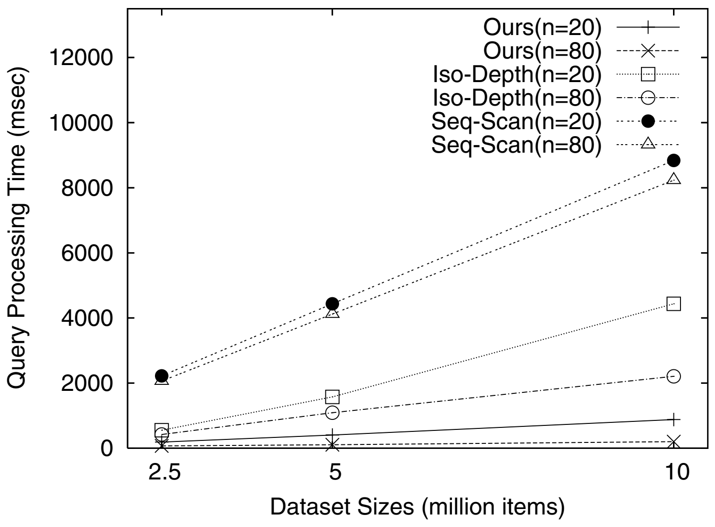

Fig. 8. Query processing times of the three methods with increasing sizes of a data set generated by the uniform distribution.

result is shown in Fig. 8. The *X*-axis shows the number of items in the event sequence and the *Y*-axis indicates the query processing times.

The result shows that the query processing times of all three methods increase linearly with the data set sizes. The query processing time of the sequential-scan-based method does not change much with the number of event types. However, the query processing times of our method and the iso-depth index decrease when the number of event types increases from 20 to 80. This is because both methods generate a smaller number of candidates as the number of event types grows. As compared with the sequential-scan-based method and the iso-depth index, our method performs about 40.9 times and 10.9 times faster, respectively, when the data set has 10 million items and 80 event types.

Last, Fig. 9 shows the query processing times of the three methods with various time window sizes. The data set had 2.5 million items generated with a zipf distribution, and the tolerance values were set to 10% of the time window size. The number of event types was 20 at first and then changed to 80. The figure shows that the query processing times of all three methods increase linearly with the time window size but our method runs much faster than the other two methods. It also reveals that when the number of event types

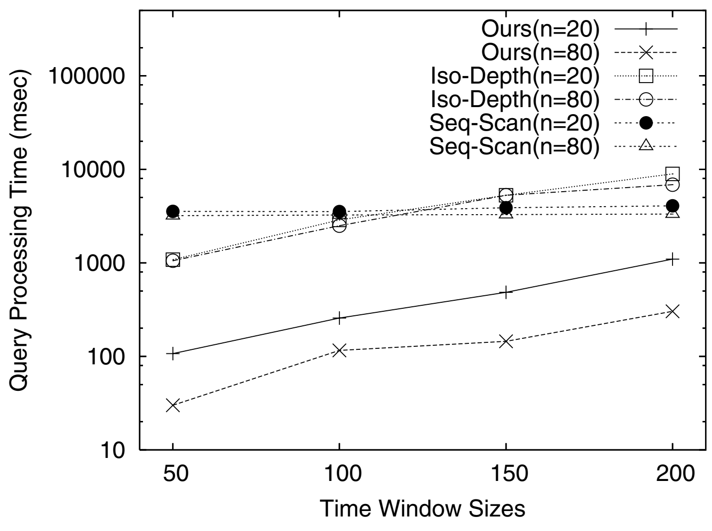

Fig. 9. Query processing times of the three methods with various time window sizes.

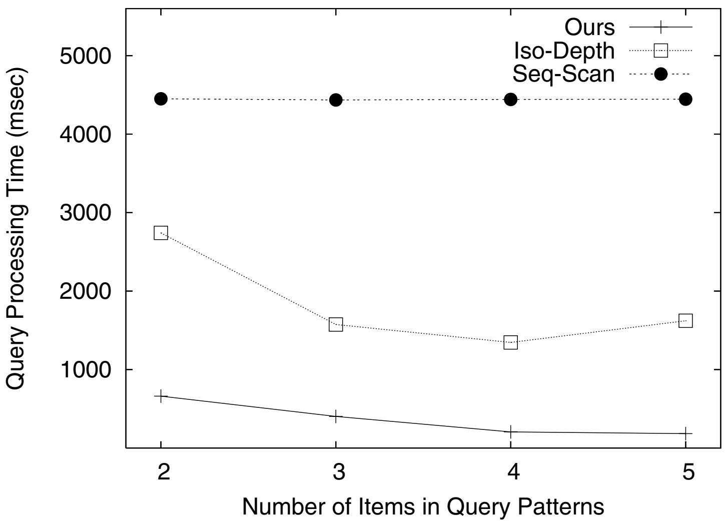

Fig. 10. Query processing times of the three methods with various numbers of event items in query patterns.

increases, the query processing time of the sequential-scan-based method does not change much but that of our method and the iso-depth index decreases. As compared to the sequential-scan-based method and the iso-depth index, our method performs up to about 106.6 times and 36.4 times faster, respectively, when the data set has 80 event types.

5.2.3. Experiment 3: query processing time with various query lengths

In this experiment, we compared the query processing times of the three methods with various numbers of event items in query patterns. The data set had 5 million data items generated with a uniform distribution. The time window size was 50 and the tolerance values were set to 10% (i.e., 5) of the time window size. The number of event types was 20 throughout the experiment. The result is shown in Fig. 10.

The query processing time of the sequential-scan-based method does not change much with the number of events in query patterns. However, the query processing times of our method and the iso-depth index decrease when the number of events in the query patterns increases. This is because both methods produce a smaller number of candidates as query patterns have more events inside. Particularly in our method, the query rectangles become smaller when the query patterns have more events inside, which significantly reduces the number of candidates retrieved by the index search.

5.2.4. Experiment 4: query processing time with extended tolerance ranges

Extending the concept of the tolerance values in query patterns, we compared the query processing times of the three methods with a new type of query QP represented as $QP.EL = \langle e_1, e_2, e_3 \rangle$, $QP.RL = \langle (0,0), (0, \max_2), (0, \max_3) \rangle$. The queries of this type establish the minimum value of all the tolerance ranges as 0 and thus require that an event e_i should occur within the range of $[0, \max_i]$ after the occurrence of the first event. These queries relax the constraint on the time intervals, and therefore enable us to obtain more results from which we may derive important knowledge.

Fig. 11 shows the query processing times of the three methods for this new type of query. The X-axis is for the number of items, generated with a uniform distribution, in the event sequence. The size of the time window was 50, and the number of event types was 20 at first and then changed to 80. This experiment reveals the following: when the number of event types is 20, the performance of the iso-depth index degrades fast as the data set becomes larger. As a result, the iso-depth index becomes slower than the sequential-scan-based method when the number of items in the data set has 10 million items. Unlike the iso-depth index, however, the increasing patterns of query processing times of both the sequential-scan-based method and our method are similar to those shown in Fig. 8. Specifically, the query processing time of our method increases smoothly. In addition, it beats both of the iso-depth index and the sequential-scan-based method.

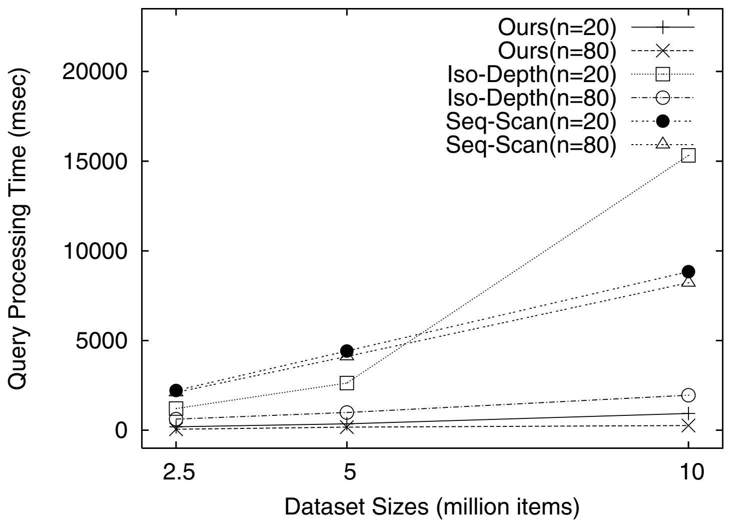

Fig. 11. Query processing times of the three methods with a new type of query QP represented as $QP.EL = \langle e_1, e_2, e_3 \rangle$, $QP.RL = \langle (0,0), (0, \max_2), (0, \max_3) \rangle$.

6. Conclusions

Timestamped event sequence matching is useful for discovering temporal relationships among timestamped events. In this paper, we have proposed a new method for effective processing of timestamped event sequence matching. Our contributions can be summarized as follows:

- We suggest a practical query model that specifies time intervals, each of which is the allowed time gap between the first and the subsequent events, as forms of value ranges.
- We point out two drawbacks of the previous method based on the iso-depth index: (1) performance degradation with range sizes and (2) difficulty in adaptation to a dynamic environment.
- We propose a new method employing the R^*-tree that resolves the two drawbacks.
- We prove the robustness of the proposed method by showing that it guarantees no false dismissals.
- We propose a method of dimensionality reduction for overcoming the problem of the dimensionality curse by grouping event types.
- We show the effectiveness of the proposed method through extensive experiments that compare it with the previous models.

According to the performance results, the proposed method shows a significant speedup by up to a few orders of magnitude in comparison to the previous models. Furthermore, the performance improvement becomes bigger (1) as a tolerance range gets larger, (2) as the number of event types increases, (3) as the size of time window increases, (4) as a data set grows in size, and (5) as a query sequence gets longer.

Acknowledgment

This work was partially supported by the Korea Research Foundation Grant funded from the Korean Government (KRF-2005-041-D00651) and by the research fund of Hanyang University (HY-2005-I).

References

[1] R. Agrawal, C. Faloutsos, A. Swami, Efficient similarity search in sequence databases, in: Proceedings of the International Conference on Foundations of Data Organization and Algorithms, FODO, 1993, pp. 69–84.
[2] R. Agrawal, K. Lin, H.S. Sawhney, K. Shim, Fast similarity search in the presence of noise, scaling, and translation in time-series databases, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, September 1995, pp. 490–501.

[3] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger, The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, 1990, pp. 322–331.
[4] K. Chakrabarti, S. Mehrotra, The hybrid tree: an index structure for high dimensional feature spaces, in: Proceedings of the 15th International Conference on Data Engineering, 1999, pp. 440–447.
[5] K.K.W. Chu, M.H. Wong, Fast time-series searching with scaling and shifting, in: Proceedings of the International Symposium on Principles of Database Systems, ACM PODS, May 1999, pp. 237–248.
[6] E. Diday, G. Govaert, Y. Lechevallier, J. Sidi, Clustering in pattern recognition, in: Digital Image Processing, Kluwer, Edition, 1981, pp. 19–58.
[7] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, Fast subsequence matching in time-series databases, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, May 1994, pp. 419–429.
[8] E. Fredkin, Trie Memory, Communications of the ACM 3 (9) (1960) 490–499.
[9] D.Q. Goldin, P.C. Kanellakis, On similarity queries for time-series data: constraint specification and implementation, in: Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP, September 1995, pp. 137–153.
[10] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Academic Press, 2001.
[11] S.W. Kim, S.H. Park, W.W. Chu, An index-based approach for similarity search supporting time warping in large sequence databases, in: Proceedings of the International Conference on Data Engineering, IEEE ICDE, 2001, pp. 607–614.
[12] Y.S. Moon, K.Y. Whang, W.K. Loh, Duality-based subsequence matching in time-series databases, in: Proceedings of the International Conference on Data Engineering, IEEE ICDE, 2001, pp. 263–272.
[13] S. Sahni, Data Structures, Algorithms, and Applications in Java, McGraw Hill International Edition, 2000.
[14] S. Park, W.W. Chu, J. Yoon, C. Hsu, Efficient searches for similar subsequences of different lengths in sequence databases, in: Proceedings of the International Conference on Data Engineering, IEEE ICDE, 2000, pp. 23–32.
[15] D. Rafiei, A. Mendelzon, Similarity-based queries for time-series data, in: Proceedings of the International Conference on Management of Data, ACM SIGMOD, 1997, pp. 13–24.
[16] D. Rafiei, On similarity-based queries for time series data, in: Proceedings of the International Conference on Data Engineering, IEEE ICDE, 1999, pp. 410–417.
[17] G.A. Stephen, String Searching Algorithms, World Scientific Publishing, 1994.
[18] A. Unal, Y. Saygin, O. Ulusoy, Processing count queries over event streams at multiple time granularities, Information Sciences 76 (14) (2006) 2066–2096.
[19] H. Wang, C. Perng, W. Fan, S. Park, P. Yu, Indexing weighted sequences in large databases, in: Proceedings of the International Conference on Data Engineering, IEEE ICDE, 2003, pp. 63–74.
[20] R. Weber, H.-J. Schek, S. Blott, A quantitative analysis and performance study for similarity search methods in high-dimensional spaces, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 1998, pp. 194–205.
[21] B.K. Yi, H.V. Jagadish, C. Faloutsos, Efficient retrieval of similar time sequences under time warping, in: Proceedings of the International Conference on Data Engineering, IEEE ICDE, 1998, pp. 201–208.
[22] B.K. Yi, C. Faloutsos, Fast time sequence indexing for arbitrary Lp norms, in: Proceedings of the International Conference on Very Large Data Bases, VLDB, 2000, pp. 385–394.
[23] M. Zhou, M.H. Wong, A segment-wise time warping method for time scaling searching, Information Sciences 173 (1–3) (2005) 227–254.