Indexing Weighted Sequences in Large Databases

Haixun Wang[1] Chang-Shing Perng[1] Wei Fan[1] Sanghyun Park[2] Philip S. Yu[1]

[1]IBM T.J.Watson Research
Hawthorne, NY 10532
{haixun,perng,weifan,psyu}@us.ibm.com

[2]Dept. of Computer Science & Engineering, POSTECH
Pohang, Korea
sanghyun@postech.ac.kr

Abstract

We present an index structure for managing weighted-sequences in large databases. A weighted-sequence is defined as a two-dimensional structure where each element in the sequence is associated with a weight. A series of network events, for instance, is a weighted-sequence in that each event has a timestamp. Querying a large sequence database by events' occurrence patterns is a first step towards understanding the temporal causal relationships among the events. The index structure proposed in this paper enables us to efficiently retrieve from the database all subsequences, possibly non-contiguous, that match a given query sequence both by events and by weights. The index method also takes into consideration the non-uniform frequency distribution of events in the sequence data. In addition, our method finds a broad range of applications in indexing scientific data consisting of multiple numerical columns for discovery of correlations among these columns. For instance, indexing a DNA micro-array that records expression levels of genes under different conditions enables us to search for genes whose responses to various experimental perturbations follow a given pattern. We demonstrate, using real-world data sets, that our method is effective and efficient.

1 Introduction

Fast sequence indexing is essential to many applications, including time series analysis, multimedia database management, network intrusion detection, etc. Recently, the field of molecular genetics has received increasing attention and is widely recognized as being one of the key technologies of this century. In this paper, we propose an efficient algorithm for indexing weighted-sequences. It directly applies to indexing and retrieval of event sequences where each event has a timestamp. We also explore how this technique can be applied to indexing scientific data sets consisting of numerical columns for discovery of correlations among these columns.

We motivate our work with applications in two important areas, event management systems and DNA micro-array analysis.

TIMESTAMPED EVENT SEQUENCES. Consider the domain of event management for complex networks, where events, or messages, are generated when special conditions arise. Each event, as well as the environment in which it occurs, is logged into a database. Among all attributes of the data set, such as Host, Severity, etc., we focus on two columns, Event and Timestamp (Table 1).

Event	Timestamp
⋮	⋮
CiscoDCDLinkUp	19:08:01
MLMSocketClose	19:08:07
MLMStatusUp	19:08:21
⋮	⋮
MiddleLayerManagerUp	19:08:37
CiscoDCDLinkUp	19:08:39
⋮	⋮

Table 1. A Sequence of Events

Given a large data set of event sequences, a typical type of query we would like to answer efficiently is illustrated by the following example.

Example 1. *Event Sequence Matching*
Find all occurrences of CiscoDCDLinkUp that are followed by MLMStatusUp that are followed, in turn, by CiscoDCDLinkUp, under the constraint that the interval between the first two events is about 20 ± 2 *seconds, and the interval between the 1st and 3rd events is about* 40 ± 3 *seconds.*

Answering such queries efficiently is important to understanding temporal causal relationships among events, which often provide actionable insights for determining problems in system management.

A query can involve any number of events, and each event has an *approximate weight*, which, in this case, is the elapsed time between the occurrence of the event and

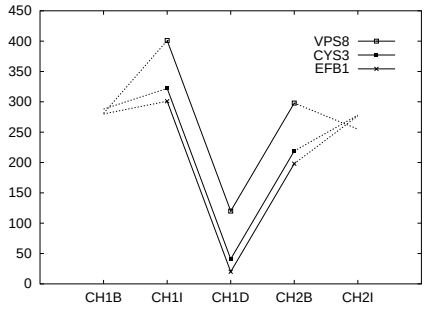


Figure 1. Expression levels of VPS8, CYS3, EFB1 rise and fall coherently in samples CH1I, CH1D, CH2B

the occurrence of the first event (CiscoDCDLinkUp) in the query sequence. There are two characteristic issues in event sequences. i) In real life datasets, more often than not, certain events occur more frequently than others. This phenomenon may affect query performance and hence, the design of the indexing algorithm. ii) It is unlikely that two causally related events are separated by a very large time gap. In other words, queries on events that occur within a same time period should be our focus of study.

GENE EXPRESSION DATA ANALYSIS. Scientific data sets usually consist of many numerical columns. One such example is the gene expression data. DNA micro-arrays are an important breakthrough in experimental molecular biology, for they provide a powerful tool in exploring gene expression on a genome-wide scale. By quantifying the relative abundance of thousands of mRNA transcripts simultaneously, researchers can discover new functional relationships among a group of genes [5, 7, 8].

Investigations show that more often than not, several genes contribute to a disease, which motivates researchers to identify genes whose expression levels rise and fall coherently under a subset of conditions, that is, they exhibit fluctuation of a similar shape when conditions change [5, 7, 8]. Figure 1 shows that three genes, VPS8, CYS3, and EFB1, respond to certain environmental changes coherently.

It is widely believed that we will be facing an explosion of gene expression data that may dwarf even the human sequencing projects [1, 4]. Management of such data is becoming one of the major bottlenecks in the utilization of the micro-array technology. In this paper, we study the *query-by-pattern* problem on such datasets. Queries such as the following are essential in discovering gene correlations [5, 7] from large scale DNA micro-array data.

Example 2. *Query-by-pattern in DNA Micro-arrays*
Find all genes whose expression level in sample CH1I is about 100 ± 5 *units higher than that in CH2B,* 280 ± 10 *units higher than that in CH1D, and* 75 ± 7 *units higher than that in CH2I.*

The query can also be expressed in SQL:

```
SELECT *
FROM dna-array
WHERE  95 ≤ (CH1I - CH2B) ≤ 105
   AND270 ≤ (CH1I - CH1D) ≤ 290
   AND  68 ≤ (CH1I - CH2I) ≤ 82
```

Database indexing techniques, such as B-tree and R-tree, are based on values of a single column or multiple columns, and do not support querying by pattern correlations. A DNA micro-array can have hundreds of columns, and the WHERE clause in a query can contain any subset of them. Indexing on each and every single column or a limited number of combinations of them can usually do no better than scanning the entire data set.

OUR CONTRIBUTIONS. This paper presents a novel problem with a wide range of potential applications.

- We introduce the weighted-sequence model. In addition to modeling timestamped sequence data, it finds applications in a much wider range of scenarios, for example, fast retrieval of objects that conform to a numerical pattern from scientific datasets.
- We present a method to convert the numerical pattern problem (Example 2) to the problem of weighted-subsequence matching. A relational table of numerical columns is treated as a set of weighted sequences.
- We present a solution to the frequency distribution problem in event sequences. We transform both the sequence data and the query so that sequence matching always starts with the least frequent symbol, which has the highest selectivity. Thus, we are able to reduce disk accesses when events are distributed non-uniformly (e.g. Zipf distribution).
- We present an index structure that enables us to perform weighted subsequence matching orders of magnitude faster than other approaches, including linear scan or the B^+-Tree index.

The rest of the paper is organized as follows. Section 2 reviews related work. We formalize the weighted-subsequence matching problem in Section 3. In Section 4, we show that queries over numerical tables can be expressed through weighted-subsequence matching. Section 5 presents a compact index structure and an algorithm framework. In Section 6, we study how to improve query performance by taking advantage of symbol's frequency distribution. Experiments and results are reported in Section 7.

2 Related Work

There has been much research on indexing substrings. A suffix tree [15] is a very useful data structure that embodies a compact index to all the distinct, non-empty substrings of a given string. Suffix arrays [14] and PAT-arrays [10] also provide fast searches on text databases.

The above index structures, however, are not adequate to solve the problems mentioned in the previous section, because they only provide fast accesses for searching contiguous subsequences in a string database. More specifically, in string matching, the relative positions of two elements in a string is also used to embody the distance between them, while in Example 1, the distance between two elements is expressed explicitly by another dimension, the weight.

Similarity based subsequence matching [9, 16] has been a research focus for applications such as time series databases. The basic idea is to map each data sequence into a small set of multidimensional rectangles in the feature space. Traditional spatial access methods [11, 3] are then used to index and retrieve these rectangles. Here, retrieval is based on similarity of the time-series within a continuous time interval. The method can not be applied to solve the weighted-sequence problem since the pattern to retrieve is usually a non-contiguous subsequence in the original sequence.

Recently, the problem of *exact* matching for multidimensional strings is proposed in [12]. Strings are mapped to real numbers based on their lexical order. Later, these multidimensional points are indexed using R-trees [11]. This technique works efficiently for queries such as "find a person whose name begins with Sri and telephone number begins with 973". Our concern in this paper, however, is to find objects that match a given pattern instead of exact values.

To the best of our knowledge, there has been little research in fast retrieval of numerical patterns in relational tables. The above mentioned techniques can not be applied directly to solve this problem, largely because they only handle one-dimensional series. On the other hand, much research has been devoted to find *frequent* patterns in large databases [18, 2]. These methods typically scan a data set multiple times in order to find patterns whose occurrence level is beyond a threshold. Recent advances in biology, such as the DNA micro-array technique, also fuel this need. As a result, several clustering-based algorithms [6, 21] have been introduced to attack this problem.

3 Weighted-Sequences

In this section, we define some terminologies and formalize the problem we intend to solve.

Definition 1. *Weighted-Sequence*
A weighted-sequence is a sequence of (symbol, weight) pairs.

$$\mathcal{T} = \langle (a_1, w_1), (a_2, w_2), \cdots, (a_n, w_n) \rangle$$

Here, each a_i is a symbol, and $w_i \in \mathbb{R}$.

In this paper, we focus on sequences where weights are in ascending order, i.e., $w_i \leq w_j$ for $i < j$. Event sequences, for instance, are generated with increasing timestamps. We shall see later that numerical tables can be reduced to sequences in the ordered form as well. For the rest of the paper, we use weighted-sequences to denote sequences with ascending weights.

We introduce some notations based on the above definition.

$\mathcal{T}$	a weighted-sequence
$\mathcal{T}_i$	the i-th item in sequence $\mathcal{T}$
$s(\mathcal{T}_i)$	symbol of the i-th item
$w(\mathcal{T}_i)$	weight of the i-th item
A	symbol set, $A = \bigcup_i \{s(\mathcal{T}_i)\}$
$\lvert\mathcal{T}\rvert$	length (number of items) of $\mathcal{T}$
$\lVert\mathcal{T}\rVert$	weight range of $\mathcal{T}$, $\lVert\mathcal{T}\rVert = w(\mathcal{T}_{\lvert\mathcal{T}\rvert}) - w(\mathcal{T}_1)$
$\mathcal{T}' \subset \mathcal{T}$	$\mathcal{T}'$ is a (non-contiguous) subsequence of $\mathcal{T}$
ξ	window size

Note that $\lVert\mathcal{T}\rVert$ is the weight difference between the first and last elements of $\mathcal{T}$. If $\mathcal{T}$ is a timestamped event sequence, then $\lVert\mathcal{T}\rVert$ is the time span between the first and the last event that compose the sequence, while $\lvert\mathcal{T}\rvert$ is the number of events occurred during that period. A subsequence of $\mathcal{T}$, possibly non-contiguous, is derived from $\mathcal{T}$ by discarding some of its items. We use $\mathcal{T}' \subset \mathcal{T}$ to indicate that $\mathcal{T}'$ is a (non-contiguous) subsequence of $\mathcal{T}$.

A query sequence $\mathcal{Q} = \langle (b_1, 0), \cdots, (b_m, w_m) \rangle$ is a special weighted-sequence in that the weight of the first item in the sequence, $w(\mathcal{Q}_1)$, is 0.

Definition 2. *Weighted-Sequence Matching*
A query sequence $\mathcal{Q}$ matches sequence $\mathcal{T}$ if there exists a (non-contiguous) subsequence $\mathcal{T}' \subset \mathcal{T}$ such that $\lvert\mathcal{Q}\rvert = \lvert\mathcal{T}'\rvert$, $s(\mathcal{Q}_i) = s(\mathcal{T}'_i)$, and $w(\mathcal{Q}_i) = w(\mathcal{T}'_i) - w(\mathcal{T}'_1)$, $\forall i \in 1, \cdots, \lvert\mathcal{Q}\rvert$.

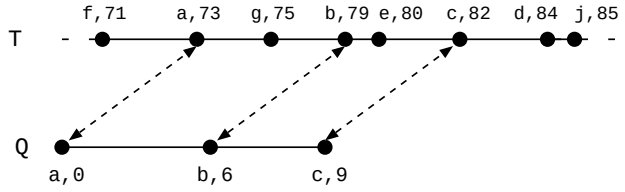


Figure 2. Query $\mathcal{Q} = \langle (a, 0), (b, 6), (c, 9) \rangle$ matches a non-contiguous subsequence of $\mathcal{T}$, $\langle (a, 73), (b, 79), (c, 82) \rangle$

An example of weighted-sequences matching is shown in Figure 2, where a query sequence $\langle(a,0),(b,6),(c,9)\rangle$ matches a weighted-subsequence of $\mathcal{T}$.

Sequence matching in Definition 2 requires that the weight difference between any two items in the matched subsequence is exactly the same as that of the corresponding items in the query sequence. This restriction can be relaxed to allow approximate matching.

Definition 3. *Approximate Matching of Weighted-Sequences Given a sequence $\mathcal{T}$, a query sequence $\mathcal{Q}$, and tolerance $e_i \geq 0, i \in 1,\cdots,|\mathcal{Q}|$, we say $\mathcal{Q}$ approximately matches $\mathcal{T}$ if there exists a (non-contiguous) subsequence $\mathcal{T}' \subset \mathcal{T}$ such that $|\mathcal{Q}| = |\mathcal{T}'|$, $s(\mathcal{Q}_i) = s(\mathcal{T}'_i)$, and $|w(\mathcal{Q}_i) - (w(\mathcal{T}'_i) - w(\mathcal{T}'_1))| \leq e_i, \forall i \in 1,\cdots,|\mathcal{Q}|$.*

The tolerances given in a query must not disturb the order of the elements in the sequence, otherwise ambiguity might arise. In other words, we require $w(\mathcal{Q}_i) + e_i < w(\mathcal{Q}_{i+1}) - e_{i+1}$, which guarantees that $\mathcal{Q}_i$ precedes $\mathcal{Q}_{i+1}$ in the presence of the tolerances.

4 Reducing Numerical Tables to Weighted-Sequences

The purpose of introducing weighted-subsequence matching is two-fold. First of all, it models problems in event management systems. More importantly, it sheds light on a general topic with a wider range of applications, i.e., querying relational tables with numerical columns, such as the DNA micro-array presented in Example 2.

Our approach is to reduce the latter problem into the former one. Each record in a numerical relation can be transformed into a weighted-sequence. For instance, expression levels of gene VPS8 under condition {*CH1I, CH1B, CH1D, CH2I, CH2B*} can be represented by a list of column-value pairs,[1] which is shown in Table 2.

Then, we sort the column-value pairs in the list by their values in ascending order to derive a sequence with ascending weights, which has column names as symbols, and expression levels as weights. Concatenating all the weighted-sequences thus derived from each gene record, and delimiting them by a special item, for instance, `(NULL, 0)`, we have transformed the entire table into a long sequence.

We call the resulting long sequence (shown in Table 2) a *generalized* weighted-sequence, as weights of its items are only sorted locally between the delimiters.

The problem of querying relational tables with numerical columns is now equivalent to (approximate) weighted-subsequence matching. For instance, the DNA micro-array query of Example 2 can be paraphrased into the following after the array is transformed into a sequence:

[1] For presentation simplicity, we use only 5 columns of the Yeast gene array. A widely available Yeast micro-array [19] has 17 columns.

Example 3. *Querying DNA Micro-array by Sequence Matching*
Let $\mathcal{T}$ be the generalized weighted-sequence converted from the Yeast DNA micro-array. Find all subsequences of $\mathcal{T}$ that match query

$$\mathcal{Q} = \langle(\texttt{CH1D},0),(\texttt{CH2B},180),(\texttt{CH2I},205),(\texttt{CH1I},280)\rangle$$

with tolerance $e_1 = 0$, $e_2 = 5$, $e_3 = 7$, and $e_4 = 10$.

Thus, we have unified the two problems by a single model. However, in micro-array queries such as Example 3, each symbol (column name) appears at most once. This is not necessarily true in querying event sequences, for instance, in Example 1, event CiscoDCDLinkUp appears twice. Also, symbols in event sequences often distribute non-uniformly, while all symbols occur at the same rate in micro-array sequences.

5 The Iso-Depth Index

In this section, we describe an index structure for weighted-sequence matching. We take into consideration symbols' frequency distribution in the sequence.

5.1 Overview

We propose an index structure called *Iso-Depth Index* to support fast accesses of (non-contiguous) subsequences that match a query sequence.

The iso-depth structure embodies a compact index to all the distinct, non-empty sequences whose weight range is less than ξ, a window size provided by the user. For event sequences, we choose a ξ such that two events separated by a gap longer than ξ are rarely correlated. For scientific data sets, there is no need for a moving window to cross the `(NULL,0)` record boundary. Therefore, the window size is no larger than the value range of the scientific dataset. However, if we choose a ξ less than the value range, then there is a possibility that we need to break down a query into multiple sub-queries. We do not discuss the details of handling queries longer than window size in this paper due to lack of space.

Weights in sequences are usually represented by real numbers. We discretize them into a number of equi-width or equi-depth units, depending on their distribution and the application. In the rest of the paper, we assume that the weights in the sequences are already discretized into equi-width units. The queries, as well as the tolerances associated with them, are discretized in the same way.

During the index construction, a trie is employed as an intermediary structure to facilitate the building of the iso-depth index. The trie is not used during query processing. Various approaches to build tries or suffix trees in linear time have been developed. Ukkonen [20], for

Gene VPS8: (CH1I, 401), (CH1B, 281), (CH1D, 120), (CH2I, 275), (CH2B, 298)

$\Downarrow$

Sort by value: $\langle$(CH1D, 120), (CH2I, 275), (CH1B, 281), (CH2B, 298), (CH1I, 401)$\rangle$

$\Downarrow$

Concatenate all records: $\langle \underbrace{(\texttt{CH1D}, 120), \cdots, (\texttt{CH1I}, 401)}_{1st\ record}, (\texttt{NULL}, 0), \underbrace{(\texttt{CH1D}, 109), \cdots, (\texttt{CH2I}, 580)}_{2nd\ record}, (\texttt{NULL}, 0), \cdots \rangle$

Table 2. Reducing Numerical Tables to Weighted-Sequences

instance, developed a linear-time, on-line suffix tree construction algorithm. We do not address the details of building suffix trees in this paper. The suffix tree, however, only supports efficient matching of contiguous substrings. If a query string contains gaps, for instance, `abc*****def`, where * stands for 'don't care' we need to traverse the subtree under `c` for up to 5 levels to find all occurrences of `d` there. The proposed iso-depth structure enables us to *jump* to such `d`'s right away without traversing the subtree. Thus, the iso-depth index provides the capability to support efficient matching of non-contiguous subsequences.

The index structure and the query process presented in this section do not take into account the impact of symbol's frequency distribution. The problem is discussed in detail in Section 6.

5.2 Building the Index Structure

We examine an example before we present the details of the index structure.

Example 4. *Let a sequence database $\mathcal{D}$ be composed of the following symbol/weight pairs. Let window size $\xi = 16$.*

$$\mathcal{D} = (b,6),(d,9),(a,11),(d,14),(a,17),(c,18),(b,23),(c,25),(d,28),(a,29),(c,30)$$

Since queries are constrained by the windows size, we only need to index subsequences $\mathcal{T} \subset \mathcal{D}$ where $\|\mathcal{T}\| < \xi$. We create a moving window of size ξ over $\mathcal{D}$. As we move the window along $\mathcal{D}$, we find the following subsequences in the window:

$\mathcal{T}$	$\|\mathcal{T}\|$
(b, 6), (d, 9), (a, 11), (d, 14), (a, 17), (c, 18)	12
(d, 9), (a, 11), (d, 14), (a, 17), (c, 18), (b, 23)	14
(a, 11), (d, 14), (a, 17), (c, 18), (b, 23), (c, 25)	14
(d, 14), (a, 17), (c, 18), (b, 23), (c, 25), (d, 28), (a, 29)	15
...	...

Next, we apply function f defined below on each subsequence $\mathcal{T}$ in the above table and encode it into a one dimensional sequence $\mathcal{S}$:

$$f(\langle \mathcal{T}_1, \cdots, \mathcal{T}_k \rangle) = \langle \mathcal{S}_1, \cdots, \mathcal{S}_k \rangle \quad (1)$$

where:

$$\mathcal{S}_i = \begin{cases} s(\mathcal{T}_i)_0 & : \ i = 1 \\ s(\mathcal{T}_i)_{w(\mathcal{T}_i) - w(\mathcal{T}_{i-1})} & : \ i > 1 \end{cases}$$

The resulting one-dimensional sequences are shown below:

$f(\mathcal{T})$
$b_0, d_3, a_2, d_3, a_3, c_1$
$d_0, a_2, d_3, a_3, c_1, b_5$
$a_0, d_3, a_3, c_1, b_5, c_2$
$d_0, a_3, c_1, b_5, c_2, d_3, a_1$
...

Subscripts of symbols in a one-dimensional sequence, as a matter of fact, represent intervals (weight differences) between two adjacent symbols in the original weighted sequence. Let $f(\mathcal{T}) = S$. The weight range of $\mathcal{T}$ is the sum the subscripts in S, i.e., $\|\mathcal{T}\| = \sum_{v_i \in S} i$.

The encoded sequences have an expanded symbol set. Take the 3rd encoded sequence for example. In $f(\mathcal{T}) = \langle a_0, d_3, a_3, c_1, b_5, c_2 \rangle$, a_0 and a_3 are two different, independent symbols. We insert $f(\mathcal{T})$ into a trie by following the arcs in $f(\mathcal{T})$, as shown in Figure 3. Each node in the trie has an *offset list*. Assuming the insertion of $f(\mathcal{T})$ leads to node u, which is pointed to by arc c_2, we append to the offset list of u the position of $\mathcal{T}$ in the original sequence $\mathcal{D}$, which in this case, is 3, since $\mathcal{T}$ appears in the 3rd window of $\mathcal{D}$.

After all encoded sequences are inserted, we assign sequential IDs (starting with 0, which is assigned to the root) to the tree nodes in the depth-first traversal order. In addition, we also record for each node the largest ID of its descendents. More specifically, each node v is assigned a pair of labels, (v_s, v_m), where v_s is the ID of node v, and v_m is the largest ID of v's descendent nodes. Based on the numbering, the ID of any descendent of v is between v_s and v_m. Similar labeling scheme is used for indexing XML documents [13].

For a given node v, let $v_\mathcal{P}$ be the path descending from the root to v. We use $\|v_\mathcal{P}\|$, or simply $\|v\|$, to denote the distance between the root and v. $\|v\|$ can be derived by simply summing up the subscripts of the symbols in sequence $v_\mathcal{P}$. As an example, for nodes x and y in Figure 3, we have $\|x\| = 6$ and $\|y\| = 12$.

Next, we create iso-depth links for each (x, d) pair, where x is a symbol, and $d = 1, \ldots, \xi$. As we visit the nodes in depth-first order, we append each node to an iso-depth link. Assuming arc x_k points to node v, we append v to the iso-depth link for pair $(x, \|v\|)$. Thus, an iso-depth link is composed of nodes that have the same distance from the root. As shown in Figure 3, each node v, represented by pair (v_s, v_m), appears in only one iso-depth link. The iso-depth links shown in the figure are for illustrative purpose only; instead of linked lists, they are realized by consecutive buffers or B^+-Trees for efficient accesses (Section 5.3).

The iso-depth links have the following property.

Property 1. *The Iso-depth Property*

1. *Nodes in an iso-depth link are sorted by their IDs in ascending order;*
2. *A node's descendents that appear in an iso-depth link are contiguous in that link. More formally, let $v \cdots w \cdots u$ be three nodes in an iso-depth link, in that order. If r is an ancestor of both v and u, then r is also an ancestor of w.*

Proof.

1. Nodes are appended during the depth-first traversal when node IDs of increasing values are generated.
2. Since r is an ancestor of both v and u, we have $r_s < v_s \leq r_m$ and $r_s < u_s \leq r_m$. From $v_s < w_s < u_s$, we get $r_s < w_s < r_m$, which means r is an ancestor of w.

□

Input: $\mathcal{D}$: weighted-sequence, ξ: window size
Output: F: index of $\mathcal{D}$

for *all sequences $\mathcal{T}$ in moving window of size ξ* **do**
  insert $f(\mathcal{T})$ into a trie;
make a depth-first traversal of the tree;
for *each node v encountered in the traversal* **do**
  label node v by (v_s, v_m);
  let a_k be the tree arc that points to v;
  append (v_s, v_m) to iso-depth list $(a, \|v\|)$;
index file F contains two parts:
▷ iso-depth links (as consecutive buffers), where a node v is represented by a pair (v_s, v_m);
▷ offset list $L[0...m]$ for node $0...m$, where m is the largest node ID.

Algorithm 1: Index Construction

Algorithm 1 summarizes the index construction procedure. The construction of the iso-depth index has time complexity $O(n)$. The well-known Ukkonen algorithm [20] builds suffix tree in linear time. The construction of the trie used for iso-depth indexing is less time consuming because the length of the subsequences inserted into the trie is constrained by ξ, the window size. Thus, a brute-force algorithm [15] can construct the trie in linear time. For large datasets, we construct the trie with limited main memory, and merge the trie to a disk-resident tree structure periodically.

5.3 Secondary Storage Management

In Figure 3, for presentation simplicity, iso-depth links are depicted as linked lists. In reality, the (v_s, v_m) pairs in an iso-depth link are stored consecutively in an array. Since v_s are in ascending order (Property 1), by storing them consecutively in an array we can use binary search to locate nodes whose IDs are within a given range.

The secondary index is composed of two major parts: i) arrays of (v_s, v_m) pairs for iso-depth links; and ii) offset lists of nodes. As shown in Figure 4, the iso-depth arrays are organized in ascending order of `(symbol, depth)`, the offset lists in ascending order of node IDs. Both of the structures are one-dimensional buffers, which are straightforward to implement for disk paging. Note that *we do not store the tree structure (parent-child links) in the index*. We show in Section 5.4 that the index structure in Figure 4 contain complete information for efficient subsequence matching. The index has a header structure which contains disk offsets (block IDs) to the iso-depth arrays and the offset lists. This information is made memory-resident when the index is initialized.

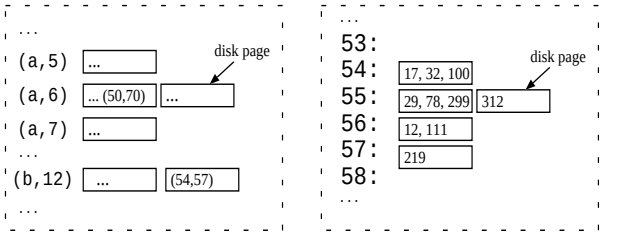


Figure 4. Secondary index

The space taken by the secondary index is linearly proportional to the data sequence. As shown in Figure 4, the index is composed of two parts: Part I, the iso-depth arrays, and Part II, the offset lists. Let $\mathcal{D}$ be a data sequence of length n. Since each trie node appears once and only once in the iso-depth links, the total number of entries in Part I equals the total number of nodes in the trie, or $O(\xi n)$ for the worst case (if none of the nodes are shared by any two subsequences). On the other hand, there are exactly n offsets stored in Part II. Thus, the space is linearly proportional to n.

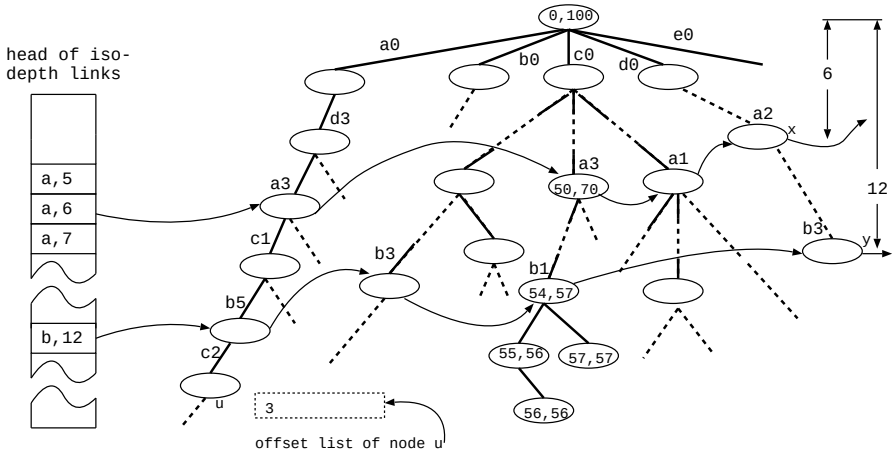


Figure 3. Each node v is represented by a pair (v_s, v_m), where v_s is the ID of v, and v_m is the largest ID under v. Iso-depth links are for illustrative purpose only.

The index construction algorithm and the secondary storage scheme assume that we are managing static datasets. To support dynamic sequence insertions, we need to modify the numbering method. One option is to use prefix paths (starting from the root node) as the labels for the tree nodes. Also, we shall use B^+-Trees instead of consecutive buffers in order to allow dynamic insertion of nodes to the iso-depth links.

5.4 Subsequence Matching

In this section, we demonstrate how to find non-contiguous subsequence matches using the iso-depth index structure.

Suppose we have a query sequence $\mathcal{Q} = \langle (c, 0), (a, 6), (b, 12) \rangle$. We start with node $(c, 0)$, which has only one pair of labels (Figure 3). Let us assume the label is $[20, 200]$, meaning sequences starting with symbol c are indexed by nodes from 20 to 200. Then, we consult iso-depth buffer $(a, 6)$, which contains all the nodes of a that are 6 units away from the root. However, we are only interested in those nodes that are descendents of $(c, 0)$. According to the iso-depth property, those descendents are contiguous in the iso-depth link and their ID $\in [20, 200]$. Since the nodes in the buffer are organized in ascending order of their IDs, the search is carried out as a range query in log time. Suppose we find three nodes, $u = [42, 61]$, $v = [88, 99]$, and $w = [102, 120]$, in that range. Then, we repeat the process for each of the three nodes in iso-depth buffer $(b, 12)$. Assume in the iso-depth buffer of $(b, 12)$, node x is a descendent of node u, node y a descendent of node v, and none are descendents of node w. We now have matched all the symbols in $\mathcal{Q}$, and the offset lists of nodes x, y and their descendents contain offsets for the query sequence $\mathcal{Q}$. Assuming $x = [53, 55]$ and $y = [97, 98]$, we find in Part II the offset lists of nodes 53, 54, 55, 97, and 98. These are the offsets in the data sequence where subsequence $\mathcal{Q}$ occurs.

For approximate matching, we might need to consult multiple iso-depth buffers for each symbol in the query. For instance, let's assume the above query $\mathcal{Q}$ comes with error tolerance $e_1 = 1, e_2 = 1$. To match the 2nd symbol, instead of consulting iso-depth array $(a, 6)$ alone, we need to consult iso-depth array (a, j), where $6 - e_1 \leq j \leq 6 + e_1$, or more specifically, $(a, 5), (a, 6)$ and $(a, 7)$, to find nodes whose ID $\in [20, 200]$. We repeat this process for the rest of the symbols in the query.

Algorithm 2 presents the outline of searching a given weighted subsequence in an index file. It first consults iso-depth links, then it returns offsets in the offset lists. It shows that iso-depth links contain complete information for subsequence matching.

6 Symbols' Frequency Distribution

The index structure discussed in the previous section does not take into account the occurrence frequency of different symbols. In real-life event logs of network management systems, certain events occur much more frequently than others. A close study reveals that the frequency of the events follows a Zipf-like distribution. Zipf's law states that the frequency count of the r-th ranked event is inversely proportional to the rank:

$$freq \sim r^{-b}$$

with exponent b close to unity.

This phenomenon poses a problem for weighted-subsequence matching. Rare events are usually of more interest and they occur frequently in queries. Imagine we have a query $\mathcal{Q} = \langle (a, 0), (b, 10), (c, 20) \rangle$, and a, b are the most common events in the data set, while c is the least common

Input: $\mathcal{Q}$: query sequence, $e_1, ..., e_{|\mathcal{Q}|}$: tolerance
Output: offsets in $\mathcal{D}$ where $\mathcal{Q}$ occur

Let $\mathcal{Q} = \langle (q_1, 0), \cdots, (q_i, w_i), \cdots \rangle$;
$v \leftarrow$ root's child node under arc q_1;
$search(v, 1)$;

Function $search(v, i)$
if $i < |\mathcal{Q}|$ **then**
  $i \leftarrow i + 1$;
  for *each iso-depth link I*=$(q_i, j), w_i - e_i \leq j \leq w_i + e_i$ **do**
    /* *Perform binary search in* I *to find nodes* $\in [v_s, v_m]$ */
    for *each node* $r \in I$ *whose ID* $\in [v_s, v_m]$ **do**
      $search(r, i)$;
    end
  end
else
  output $L[v_s ... v_m]$, offset lists of node v and all nodes under v;
end

Algorithm 2: Subsequence Matching

one. Starting with event a, we often need to examine a large amount of nodes, although only few of them finally lead to c. It is more desirable if subsequence matching starts with the least frequent symbol.

In this section, we show that the problem can be solved by preprocessing the sequences before they are inserted into the trie, and preprocessing the query sequences before we start sequence matching.

Let A be the symbol set and let $rank(a)$ denote the (reverse) frequency rank of symbol $a \in A$, i.e., the least frequent symbol is ranked 0, the most frequent symbol $|A| - 1$. We convert sequence $\mathcal{T}$ to sequence $\mathcal{T}'$ by mapping element $(x, w) \in \mathcal{T}$ to element $(x, w') \in \mathcal{T}'$, where $w' = rank(x) \times 2\xi + w$, and we sort the elements in $\mathcal{T}'$ in ascending order by their new weights. Intuitively, elements in a window of size ξ are distributed to a window of size no larger than $2\xi|A|$ so that in the new window less frequent symbols always precede more frequent ones. The reason why we multiply 2ξ to $rank(x)$ will become clear when we prove the Sequence Reordering Property.

We place a moving window of size $2\xi|A|$ on $\mathcal{T}'$ and index the sequence in the window. Assume the following sequence is in a window on $\mathcal{T}'$:

$$V = \langle (x, w'), \cdots, (y, u'), \cdots \rangle$$

Consider (x, w'), the 1st element of the above sequence, and any other element $(y, u') \in V$. If the two elements are more than ξ apart in the original sequence $\mathcal{T}$, that is, $|u - w| \geq \xi$, we remove (y, u') from V. This is because our queries are restricted by length ξ, so there is no need to index elements more than ξ apart from each other. After filtering out such elements, we insert the sequence of the remaining elements into the trie.

We demonstrate the above process by an example. Suppose we have a total of 3 symbols, a, b, and c, with descending frequency rates: $rank(a) = 2$, $rank(b) = 1$, and $rank(c) = 0$. We are given the following data sequence $\mathcal{T}$, with window size $\xi = 20$.

$$\mathcal{T} = \langle (a, 1), (b, 8), (c, 19), (b, 48), (a, 66) \rangle$$

We derive $\mathcal{T}'$ from $\mathcal{T}$ by changing (x, u) to $(x, rank(x) \times 2\xi + u)$. We order elements in $\mathcal{T}'$ by their new weights. (The old weights are left as subscripts for presentation purpose.)

$$\mathcal{T}' = \langle (c, 19_{19}), (b, 48_8), (a, 81_1), (b, 88_{48}), (a, 146_{66}) \rangle$$

We place a moving window of size $2|A|\xi = 120$ on $\mathcal{T}'$. Table 3 lists each sequence in the moving window.

Underlined elements in the above table are removed. For instance, $(b, 88_{48})$ in V_2 is removed because its distance (in the original sequence $\mathcal{T}$) to the first element of V_2, $(b, 48_8)$, is $|48 - 8| = 40$, which is larger than $\xi = 20$.

Given a query $\mathcal{Q}$, we convert it to $\mathcal{Q}'$ using the same process. For instance, let $\mathcal{Q} = \langle (a, 0), (b, 7), (c, 18) \rangle$. After deriving $\mathcal{Q}' = \langle (c, 18), (b, 47), (a, 80) \rangle$, we search for $f(\mathcal{Q}') = c_0, b_{29}, a_{62}$ in the trie. One match is found since c_0, b_{29}, a_{33} has been inserted.

By using the sequence reordering process, we always start sequence matching by the least frequent symbol in the query. It has the potential of saving lots of disk accesses since $c \rightarrow b$ is much rarer than $a \rightarrow b$, given that a and b occur more frequently than c.

More formally, the trie constructed for $\mathcal{T}'$ has the following property:

Property 2. *Sequence Reordering Property*

1. *Elements of the inserted sequences are ordered by ascending symbol frequency rate.*
2. *The trie for* $\mathcal{T}'$ *indexes all the subsequences (with length* $< \xi$*) in* $\mathcal{T}$.

Proof.

1. Assume $V = \langle (x, w'), \cdots, (y, u'), \cdots, (z, v'), \cdots \rangle$ is an inserted sequence. We have $|u - w| < \xi$ and $|v - w| < \xi$, from which we get $v - u < 2\xi$. Since V is a weighted sequence, we have $u' \leq v'$, and

$$\begin{aligned} rank(y) * 2\xi + u &\leq rank(z) * 2\xi + v \\ rank(y) - rank(z) &\leq \frac{v - u}{2\xi} < 1 \end{aligned}$$

   which means symbols in V are in ascending order of frequency rate.

	moving window of size 120	after removing elements	after applying f()
V_1	$\langle (c, 19_{19}), (b, 48_8), (a, 81_1), \underline{(b, 88_{48})} \rangle$	$\langle (c, 19), (b, 48), (a, 81) \rangle$	c_0, b_{29}, a_{33}
V_2	$\langle (b, 48_8), (a, 81_1), \underline{(b, 88_{48})}, \underline{(a, 146_{66})} \rangle$	$\langle (b, 48), (a, 81) \rangle$	b_0, a_{33}
V_3	$\langle (a, 81_1), \underline{(b, 88_{48})}, \underline{(a, 146_{66})} \rangle$	$\langle (a, 81) \rangle$	a_0
V_4	$\langle (b, 88_{48}), \underline{(a, 146_{66})} \rangle$	$\langle (b, 88), (a, 146) \rangle$	b_0, a_{58}

Table 3. Sequence in moving windows

2. Assume V is a subsequence of $\mathcal{T}$, $||V|| < \xi$, and V' is converted from V by the process described above. Let $(x, u'), (y, v')$ be any two of the elements of V'. We have

$$\begin{aligned} v' - u' &= (rank(y) * 2\xi + v) - (rank(x) * 2\xi + u) \\ &\leq 2\xi(|A| - 1) + v - u < 2\xi|A| \end{aligned}$$

which means V' is inside a moving window of size $2\xi|A|$. Since $||V|| < \xi$, any two elements of V are less than ξ apart, so none of them will be removed before they are inserted into the trie.

□

Since reordering does not increase the length of the data sequence ($|\mathcal{T}| = |\mathcal{T}'|$), the number of subsequences inserted into the trie is the same. However, the average number of elements in the subsequences might be different. As a matter of fact, two elements in subsequence V inside a moving window can be 2ξ apart in the original sequence (both elements are at most ξ apart from the first element of V). This has the potential to double the size of the trie. However, reordering also increase the chances of path sharing. The experiments in Section 7 reveal that reordering creates no significant change in the size of the trie.

7 Experiments

We experimented our index structure on both synthetic and real life data sets. The algorithm is implemented in C on a Linux machine with a 700 MHz CPU and 256 MB main memory.

7.1 Data Sets

Each element in a synthetic data sequence is represented by a pair of integers (symbol, weight), thus, the size of a dataset comes to $8n$ bytes, where n is the number of elements it has. In our experiments, we use disk page size $B = 2K$ bytes. We also used event sequences generated by a production network.

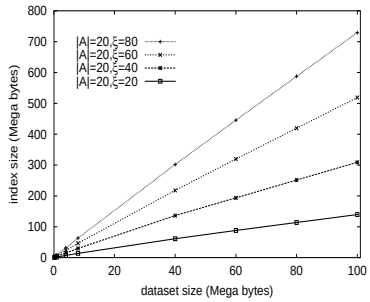
SYNTHETIC DATA In addition to symbol size $|A|$, sequence length $|\mathcal{D}|$, the synthetic data generator also simulates the distribution of symbols and weights in the sequence. Symbols are randomly generated; however, the occurrence rate of different symbols follows either a uniform or a Zipf distribution (with parameter $b = 1$). We generate nondecreasing weights from zero in such a way that the number of elements in a unit window follows either uniform or Poisson distribution. A sample dataset named `D100K-A40-Zipf-P10`, for instance, indicates that the synthetic sequence has 100K elements, 40 different symbols in Zipf distribution, and the weight difference of two adjacent elements in the sequence follows a Poisson distribution with parameter $\lambda = 10$.

EVENT MANAGEMENT DATA The data sets we use are taken from a production computer network at a financial service company. One data set (NETVIEW) [17] has six attributes: `Timestamp`, `EventType`, `Host`, `Severity`, `Interestingness`, and `DayOfWeek`. We are concerned with attribute `Timestamp` as well as attribute `EventType`, which has 241 distinctive values. The second data set (TEC) [17] has attributes `Timestamp`, `EventType`, `Source`, `Severity`, `Host`, and `DayOfYear`. In TEC, there are 75 distinctive values of `EventType` and 16 distinctive types of `Source`. It is often interesting to differentiate same type of events from different sources, and this is realized by combining `EventType` and `Source` to produce $75 \times 16 = 1200$ symbols.

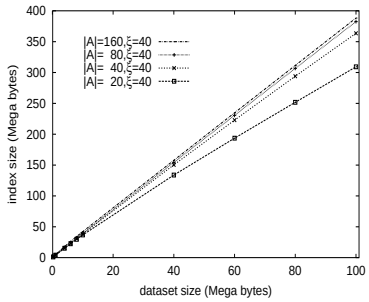
7.2 Performance Analysis

SPACE ANALYSIS. Figure 5 shows, in general, the space requirement of the iso-depth index is linearly proportional to the original sequence size. We first experimented on synthetic data sequence with uniform symbol distribution. The weight difference between two elements in the sequence follows Poisson distribution with parameter $\lambda = 8$. Figure 5(a) shows that the size of the moving window has an impact on the index size. Using a $\xi = 80$ window on a 100M dataset, we find the index/data-size ratio to be 7.2. The ratio drops as the dataset grows larger (the ratio is 7.5 and 7.9 for 40M and 8M datasets respectively), since instead of creating new paths in the trie, later insertions are more likely to share existing ones.

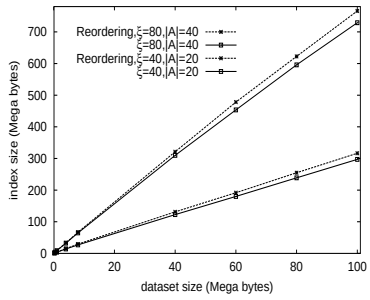
Figure 5(b) shows that $|A|$ affects the index size not as significantly as ξ does. This is especially true when the dataset is not large enough, as a result, not many sequences share paths in the trie. When ξ changes, however, the height of the trie changes proportionally, which results



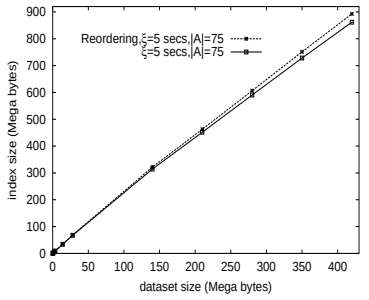
(a) Uniform symbol dist., $|A| = 20$



(b) Uniform symbol dist., $\xi = 40$

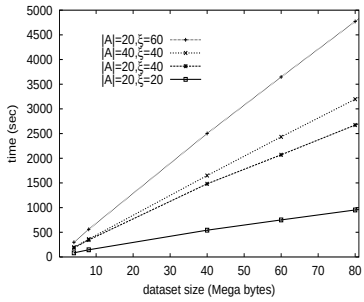


(c) Zipf symbol dist.

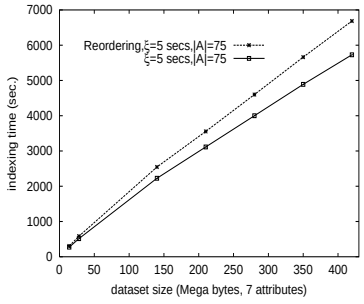


(d) TEC: 75 symbols

Figure 5. Index Size



(a) Uniform symbol dist.



(b) TEC: 75 symbols (Zipf-like dist.)

Figure 6. Construction Time

in significant changes in the index size.

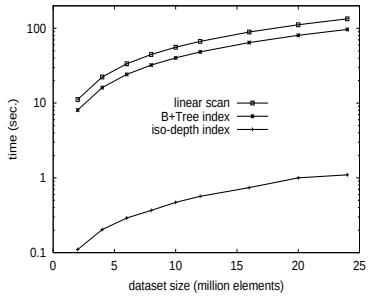
The curves in Figure 5(c) are obtained on synthetic datasets with Zipf-like symbol distribution. They show that sequence reordering increases the index size by less than 10%, a penalty we are willing to take since reordering has the potential to dramatically improve the query performance. We also performed the tests on real life dataset TEC [17] (with up to 3 months' worth of event logs totaling 400M bytes), and Figure 5(d) shows similar results.

INDEX CONSTRUCTION. Figure 6 shows the construction time of iso-depth index. We use a 24M memory buffer to store the intermediary trie. Since each node of the trie takes 12 bytes (a child pointer, a sibling pointer, and a distance-encoded symbol), we merge the memory trie onto the disk trie whenever the buffer overflows (exceeds 2 million nodes). In Figure 6(b) we show the time complexity for building the index for the TEC dataset, which has a Zipf-like symbol frequency distribution. It shows no significant overhead of sequence reordering.

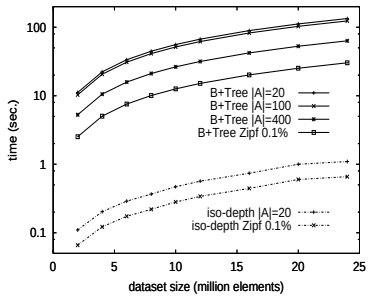
SCALABILITY ANALYSIS. The query time presented in Figure 7 shows that the iso-depth index scales much better than two alternative algorithms. The comparisons are carried out on synthetic datasets, `D?-A200-U-U10`, which range in size from 2 to 25 million elements, with uniform symbol distribution, and the average weight difference between two adjacent symbols is 10.

We issue fixed-form queries (3 evenly separated elements in a window of size $\xi = 45$) against the datasets. The Y axis of Figure 7 represents the search time in log scale (time to output the answer set not included). The alternative algorithms used in the comparisons are i) brute-force linear scan; and ii) B^+-Tree index, which records for every different symbol all of its occurrences in the data so that when given a query, we may jump directly to those positions where the first symbol of the query occurs. To match the remaining symbols in the query, a linear scan is performed within each ξ-window from those positions. On average, there are $\frac{8n}{B}(1-(1-\frac{1}{8n/B})^{\frac{n}{|A|}})$ pages that may contain the first symbol of the query (as some pages may contain more than one occurrences of the first symbol), where n is the data size (number of elements), B the page size (2K bytes). In both cases, the iso-depth index is orders of magnitude faster.

No significant changes are found in the query performance when we enlarge the moving window. However, the number of symbols and their distribution have an impact on the query performance, particularly on the B^+-Tree approach. This is because a larger $|A|$ offers better selectivity. With $|A| = 400$, the B^+-Tree approach is 2 times faster than when $|A| = 20$. We also compared their performance on a synthetic dataset whose three least frequent symbols account for only .1% of the total occurrences. *We ask only those queries that contain at least one of the three least frequent symbols and B^+-Tree is 4 times faster than linear scan.* Iso-depth index, on the other hand, is orders of magnitude faster under all these conditions.

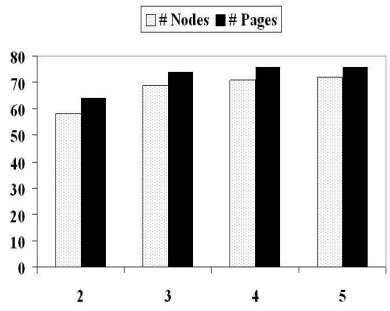


(a) Uniform distribution: D?-A200-U-U10

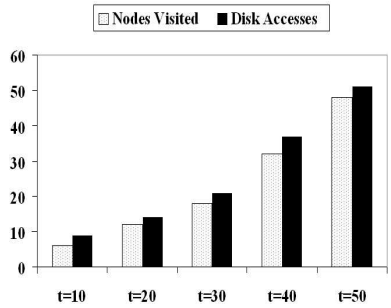


(b) B^+-tree and Iso-depth with different symbol set/distribution

Figure 7. Iso-depth vs. linear scan and B^+-Tree index.

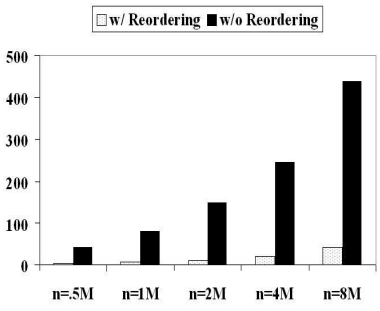


(a) D50000K-A100-U-P10

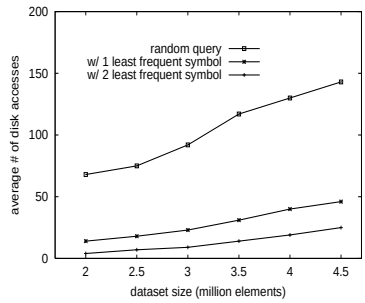


(b) $\langle (x,0),(y,t),(z,59)\rangle$

Figure 8. Different query forms



(a) $\langle (a_0,0),(a_1,t),(a_2,60)\rangle$ on D1000K-A100-U-U10



(b) Subsequence Reordering

Figure 9. Sequence Reordering

DIFFERENT QUERY FORMS. The typical response time of the iso-depth index, as indicated by Figure 7, is around 1 second even for a sequence of 25 million elements (200 M bytes). In order to further analyze the impact of different query forms on the performance, we use number of disk accesses for comparisons. First, we ask queries of different lengths on synthetic dataset D5000K-A100-U-P10 whose weights follow a Poisson distribution with parameter $\lambda = 20$.We generate random queries with 2 to 5 evenly separated events within the span of a moving window ($\xi = 40$). Figure 8(a) shows the average number of node accesses and disk accesses. Since the iso-depth index utilizes the increased selectivity of longer queries, it is robust as the query size becomes larger.

We also study the impact of different query forms on the performance. We generate queries in the form $\langle (x,0),(y,t),(z,59)\rangle$ with a varying t, where x, y, and z are random symbols in dataset D1000K-A100-U-U10, and $\xi = 60$ is the window size. Figure 8(b) presents the results. When t becomes larger, more disk accesses occur as the number of y nodes under x increases dramatically.

In Figure 9(a), we show the benefits of subsequence reordering during indexing. The datasets used in the study are the D?-A100-Zipf-U10 series. The frequency distribution of the symbols follows Zipf's law with exponent $b = 1$. In such datasets, the top 3 frequent symbols account for more than 35% of the data, while the bottom 3 account for less than 0.6%. We issue a fixed query $\langle (a_1,0),(a_5,10),(a_{100},20)\rangle$ against the datasets, where a_i is the i-th ranked symbol on the occurrence frequency list. From Figure 9(a), we find the benefits of reordering obvious. In Figure 9(b), we show the results of a similar test on real life dataset NETVIEW, whose events' occurrence rate follows a Zipf-like distribution. We discretize the time interval such that the average number of events that occur in a window with $\xi = 60$ slots is 8. We issue queries of fixed length, i.e., each query is composed of 4 evenly separated events. The event types in the query are i) randomly uniform, ii) randomly uniform, except for one event whose occurrence rate is among the bottom 3, and iii) randomly uniform, except for two events whose occurrence rate is among the bottom 3. In reality, queries ii) and iii) occur frequently, as we are more interested in rare events. It shows that frequency distribution has a strong impact on query performance. However, the number of disk accesses can be reduced significantly by the reordering method used in the iso-depth index.

8 Conclusion

In this paper, we identified a new challenge in sequence matching: indexing weighted-sequences for efficient retrieval. It generalizes the well-known string matching problem by giving each element in the sequence a weight. The

distance between two elements in a sequence is measured by their weight difference, instead of their relative positions in the sequence. In many applications, such weight differences are of great interest. Event management systems, for instance, record millions of timestamped events on a daily basis. The elapsed time (weight difference) among different events can often provide actionable insights for such systems. Another example is scientific databases, which often have dozens or even hundreds of numerical columns. We showed that querying numerical patterns among these columns is equivalent to the weighted-subsequence matching problem.

We proposed an index structure called iso-depth index for fast retrieval of weighted-subsequences in large datasets. Experimental results show that the index structure achieves orders of magnitude speedup over alternative algorithms based on naive indexing and linear scan. The method is also resilient to non-uniform frequency distribution of elements. The query performance is improved by reordering elements in subsequences (according to their occurrence rate) during indexing so that we always start matching from the least frequent element in the query.

References

[1] Alison Abbott. Bioinformatics institute plans public database for gene expression data. *Nature*, 398:646, 1999.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, 1995.

[3] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] Alvis Brazma, Alan Robinson, Graham Cameron, and Michael Ashburner. One-stop shop for microarray data. *Nature*, 403:699–700, 2000.

[5] P. O. Brown and D. Botstein. Exploring the new world of the genome with DNA microarrays. *Nature Genetics*, 21:33–37, 1999.

[6] Y. Cheng and G. Church. Biclustering of expression data. In *Proc. of 8th International Conference on Intelligent System for Molecular Biology*, 2000.

[7] P. D'haeseleer, S. Liang, and R. Somogyi. Gene expression analysis and genetic network modeling. In *Pacific Symposium on Biocomputing*, 1999.

[8] R. Miki et al. Delineating developmental and metabolic pathways in vivo by expression profiling using the riken set of 18,816 full-length enriched mouse cDNA arrays. In *Proceedings of National Academy of Sciences*, 98, pages 2199–2204, 2001.

[9] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[10] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. In *Information Retrieval: Data Structures and Algorithms*, pages 335–349. Prentice Hall, 1992.

[11] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[12] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *SIGMOD*, pages 403–414, 2000.

[13] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, September 2001.

[14] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal On Computing*, pages 935–948, 1993.

[15] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[16] Chang-Shing Perng, Haixun Wang, Sylvia R. Zhang, and D. Stott Parker. Landmarks: a new model for similarity-based pattern querying in time series databases. In *ICDE*, pages 33–42, 2000.

[17] Chang shing Perng, Haixun Wang, Sheng Ma, and Joseph L Hellerstein. A framework for exploring mining spaces with multiple attributes. In *ICDM*, 2001.

[18] Ramakrishnan Srikant and Rakesh Agrawal. Mining generalized association rules. In *VLDB*, Zurich, Switzerland, September 1995.

[19] S. Tavazoie, J. Hughes, M. Campbell, R. Cho, and G. Church. Yeast micro data set. In *http://arep.med.harvard.edu/biclustering/yeast.matrix*, 2000.

[20] E. Ukkonen. Constructing suffix-trees on-line in linear time. *Algorithms, Software, Architecture: Information Processing*, pages 484–92, 1992.

[21] Haixun Wang, Wei Wang, Jiong Yang, and Philip S. Yu. Clustering by pattern similarity in large data sets. In *SIGMOD*, 2002.