AISS: An Index for Non-timestamped Set Subsequence Queries

Witold Andrzejewski and Tadeusz Morzy

Institute of Computing Science
Poznan University of Technology
Piotrowo 2, 60-965 Poznan, Poland
{wandrzejewski, tmorzy}@cs.put.poznan.pl

Abstract. In many recent applications of database management systems data may be stored in user defined complex data types (such as sequences). However, efficient querying of such data is not supported by commercially available database management systems and therefore efficient indexing schemes for complex data types need to be developed. In this paper we focus primarily on the indexing of non-timestamped sequences of sets of categorical data, specifically indexing for set subsequence queries. We address both: logical structure and implementation issues of such indexes. Our main contributions are threefold. First, we specify the logical structure of the index and we propose algorithms for set subsequence query execution, which utilize the index structure. Second, we provide the proposition for the implementation of such index, which uses means available in all of the "of the shelf" database management systems. Finally, we experimentally evaluate the performance of the index.

1 Introduction

Many of todays commercially available database management systems allow users to define complex datatypes, such as sets, sequences or strings. One of the most important complex datatypes is the sequence. Sequences are very convienient in modelling such objects as protein sequences, DNA chains (sequences of atomic values of a small alphabet), time series (composed of real values), and Web server logs (composed of events). Purchases made by customers in stores are also sequential. Here, sequences are composed of sets of products bought by the customer, which are ordered by the date of purchase. The problem of indexing and querying sequences has recently received a lot of attention [7,12].

Although modern database management systems provide users with the means to create sequences, they do not support efficient querying of this data type. To illustrate the problem, let us consider the following example. We are given the database of four sequences of sets (for example database of sequences of purchases) shown on Table 1 and the sequence $\mathcal{Q} = \langle\{2\}, \{1\}\rangle$. The problem is to find all sequences from the database such that they contain the sequence $\mathcal{Q}$. By sequence containment we mean that the sequence $\mathcal{Q}$ is contained within

A Min Tjoa and J. Trujillo (Eds.): DaWaK 2006, LNCS 4081, pp. 503–512, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Table 1. Running example database

Id	Sequence
1	$\mathcal{S}^1 = \langle\{1,2\},\{1,2\},\{1,6\}\rangle$
2	$\mathcal{S}^2 = \langle\{2,7\},\{1,3\},\{1,5\}\rangle$
3	$\mathcal{S}^3 = \langle\{2,8\},\{1,3\},\{1,4\}\rangle$
4	$\mathcal{S}^4 = \langle\{1,3\},\{3,9\},\{1,6\}\rangle$

the sequence $\mathcal{S}$ IFF it may be obtained by removing of some of the items from the sequence $\mathcal{S}$. Here, sequence $\mathcal{Q}$ is contained within sequences $\mathcal{S}^1$, $\mathcal{S}^2$ and $\mathcal{S}^3$, but not in $\mathcal{S}^4$. Such queries are very common in many database application domains, such as: market basket sequence mining, web log mining and mining results analysis. As can be easily seen the problem is difficult. If there is no indexing structure for the database, then, in order to answer the query, we need to read all sequences from database one by one, and for each such sequence check whether it contains the given sequence $\mathcal{Q}$. Unfortunately, for large databases, brute force solution may be very costly.

Concluding, there is evidently a need to research efficient, possibly general, indexing schemes for sequences. Several indexing schemes for sequences have been proposed so far. Most of them were designed either for time series [1,4] or sequences of atomic values [14,9]. Almost nothing has been done with regard to indexing sequences of sets. According to out knowledge, the only index for sequences of sets developed so far was proposed by us in [3]. However, this index was designed for sequences of timestamped sets and its main task was to support a very special case of set subsequence queries, where sets were also timestamped.

The original contribution of this paper is the proposal of a new indexing scheme capable of efficient retrieval of sequences of non-timestamped sets. Moreover, the index also supports retrieval of multisets. We present the physical structure of the index and we develop algorithms for query processing. We also present algorithms for incremental set/sequence insertion, deletion and update in the index. The index has a very simple structure and may be easily implemented over existing database management systems.

The rest of the paper is organized as follows. Section 2 contains an overview of the related work. In Section 3 we introduce basic definitions used throughout the paper. We present our index in Section 4. Experimental evaluation of the index is presented in Section 5. Finally, the paper concludes in Section 6 with a summary and a future work agenda.

2 Related Work

Most of research on indexing of sequential data is focused on three distinct types of sequences: time series, strings, and web logs.

Indexes proposed for time series support searching for similar or exact subsequences by exploiting the fact, that the elements of the indexed sequences are numbers. This is reflected both in index structure and in similarity metrics.

Popular similarity metrics include Minkowski distance [16], compression-based metrics [6], and dynamic time warping metrics [13]. Often, a technique for reduction of the dimensionality of the problem is employed [1].

String indexes usually support searching for subsequences based on identity or similarity to a given query sequence. Most common distance measure for similarity queries is the Levenshtein distance [8], and index structures are built on suffix tree [15] or suffix array [10].

Indexing of web logs is often based on indexing of sequences of timestamped categorical data. Among the proposed solutions, one may mention: SEQ family of indexes which use transformation of the original problem into the well-researched problem of indexing of sets [11], ISO-Depth index [14] which is based on a trie structure and SEQ-Join index [9] which uses a set of relational tables and a set of B^+-tree indexes.

Recently, works on sequences of categorical data were extended to sequences of sets. The Generalized ISO-Depth Index proposed in [3] supports timestamped set subsequence queries and timestamped set subsequence similarity queries. Construction of the index involves storing all of the sequences in a trie structure and numbering the nodes in depth first search order. Final index is obtained from such trie structure.

3 Basic Definitions and Problem Formulation

Let $I = \{i_1, i_2, \ldots, i_n\}$ denote the set of *items*. A non-empty set of items is called an *itemset*. We define a *sequence* as an ordered list of itemsets and denote it: $\mathcal{S} = \langle s_1, s_2, \ldots, s_n \rangle$, where s_i, $i \in \langle 1, n \rangle$ are itemsets. Each itemset in the sequence is called an *element* of a sequence. Each element s_i of a sequence $\mathcal{S}$ is denoted as $\{x_1, x_2, \ldots, x_n\}$, where x_i, $i \in \langle 1, n \rangle$ are items. Given the item i and a sequence $\mathcal{S}$ we say that the item i is *contained* within the sequence $\mathcal{S}$, denoted $i \vdash \mathcal{S}$, if there exists any element in the sequence such that it contains the given item. Given a sequence $\mathcal{S}$ and an item i, we define $n(i, \mathcal{S})$ as a number of elements in a sequence $\mathcal{S}$ containing the item i. Given sequences $\mathcal{S}$ and $\mathcal{T}$, the sequence $\mathcal{T}$ is a *subsequence* of $\mathcal{S}$, denoted $\mathcal{T} \sqsubseteq \mathcal{S}$, if the sequence $\mathcal{T}$ may be obtained from sequence $\mathcal{S}$ by removal of some of items from the elements and removal of any empty elements which may appear. Conversely, we say that the sequence $\mathcal{S}$ *contains* the sequence $\mathcal{T}$ and that $\mathcal{S}$ is a *supersequence* of $\mathcal{T}$.

In order to present the structure of the proposed index, additional notions and definitions are needed. A *multiset* is an itemset where items may appear more then once. We denote multisets as $S_M = \{(x_1 : n_1), (x_2 : n_2), \ldots, (x_m : n_m)\}$, where x_i, $i \in \langle 1, m \rangle$ are items, and n_i are *counters* which denote how many times the items x_i appear in the multiset. We omit such items, that their counters are equal to zero (i.e. they do not appear in the multiset). Given the S_M and T_M multisets, the T_M multiset is a *subset* of the multiset S_M, denoted $T_M \subseteq S_M$ if the multiset T_M may be obtained from multiset S_M by removal of some of the items.

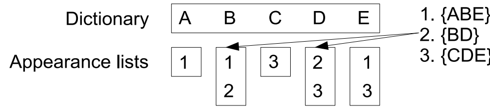

Fig. 1. Basic Inverted File Index structure

We define a *database*, denoted $\mathcal{DB}$, as a set of either sequences or multisets, called *database entries*. Each database entry in the database has a unique *identifier*. Without the loss of generality we assume those identifiers to be consecutive, positive integers. A database sequence identified by the number i is denoted $\mathcal{S}^i$, whereas a database multiset is denoted S_M^i. Given the *query sequence* $\mathcal{Q}$, the *set subsequence query* retrieves a set of identifiers of all sequences from the database, such that they contain the query sequence, i.e. $\{i : \mathcal{S}^i \in \mathcal{DB} \wedge \mathcal{Q} \sqsubseteq \mathcal{S}^i\}$. Given the *query multiset* Q_M, the *subset query* retrieves a set of identifiers of all multisets from the database such, that the multiset Q_M is their subset, i.e. $\{i : S_M^i \in \mathcal{DB} \wedge Q_M \subseteq S_M^i\}$.

4 The AISS Index

Now, we will proceed to the presentation of our index for sequences of non-timestamped sets. The idea of the index is based on the well known Inverted File Index [5]. The new structure allows to search for supersequences of sequences of sets and supersets of multisets.

The general idea for the index is as follows. We transform sequences from database to multisets by discarding data about which items belong to which itemsets and about the order of the itemsets. Next, we store these multisets in the structure based on the idea of the Inverted File Index. To perform a query, we transform the query sequence to a multiset, and retrieve all the supersets of such multiset from the index. Because we discard some of the data, additional verification phase is needed to prune false positives.

Basic Inverted File structure, which is used for indexing itemsets, is composed of two parts: *dictionary* and *appearance lists*. The dictionary is the list of all the items that appear at least once in the database. Each item has an appearance list associated with it. Given the item i, the appearance list associated with item i lists identifiers of all the sets from database, that contain that item. Structure of the basic Inverted File Index is shown on Figure 1.

In order to be able to store multisets in the above presented structure we propose a straightforward modification. We alter appearance lists, so that they store counters which show how many times the item appears in the set as well as identifiers. Notice, that such modification allows us to store full information about multisets. In order to be able to store sequences of sets in such index, we use a *sequence to multiset transformation* which is introduced by the Definition 1.

Algorithm 1. AISS index creation.

1. Build a dictionary by scanning a database and retrieving all distinct items stored in the database.
2. For each of the sequences $\mathcal{S}^j \in \mathcal{DB}$ or for each of the multisets $S_M^j \in \mathcal{DB}$ perform the following steps:
   (a) if $\mathcal{DB}$ is a database of sequences, perform the following transformation: $S_M^j = T(\mathcal{S}^j)$.
   (b) For each of the pairs $(x_i : n_i) \in S_M^j$ create an entry (j, n_i) in the appearance list associated with the item x_i.

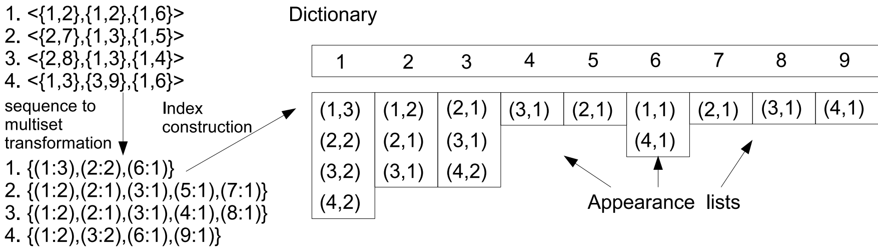

Fig. 2. AISS Index for exemplary database

Definition 1. *Sequence to multiset transformation.*

Sequence is transformed to a multiset by creating a multiset, that contains all the items from the sequence. Formally,

$$T(\mathcal{S}) = \{(x_i : n_i) : x_i \vdash \mathcal{S} \wedge n_i = n(x_i, \mathcal{S})\} \tag{1}$$

Example 1. We want to transform the sequence $\mathcal{S}^1 = \langle\{1, 2\}, \{1, 2\}, \{1, 6\}\rangle$ from the running example database to a multiset. In this sequence, item 1 occurs three times, item 2 occurs two times, and item 6 occurs one time. Therefore the multiset should contain three items 1, two items 2 and a single item 6.

$$S_M^1 = T(\mathcal{S}^1) = T(\langle\{1, 2\}, \{1, 2\}, \{1, 6\}\rangle) = \{(1 : 3), (2 : 2), (6 : 1)\}$$

The steps for AISS index creation, which utilizes a sequence to multiset transformation, are shown by algorithm 1. It is easy to notice that both steps of the algorithm 1 can be performed during a single database scan. The process of building the AISS index for the running example database is presented on Figure 2.

Basic steps for procesing subset queries are given by algorithm 2. It is easy to notice, that the basic algorithm would run faster, if the items in the query multiset were ordered by their frequency of appearance in the database. Therefore, before executing the query, we should calculate, for each item of the multiset, how many multisets in the database contain this item. This of course needs to be done only once, and can be easily updated incrementally after database updates.

Algorithm 2. Subset query algorithm utilizing the AISS index.

Parameter: query multiset $Q_M = \{(x_1 : n_1), (x_2 : n_2), \ldots, (x_l : n_l)\}$

- For each entry (j_1, m_1) from the appearance list of item x_1, if $n_1 < m_1$, do:
  1. $level \leftarrow 2$
  2. While $level <= l$ do:
     (a) Find entry (j_{level}, m_{level}) on appearance list for item x_{level}, such that $j_{level} = j_1$. If such entry does not exist, break the while loop.
     (b) If $(n_{level} < m_{level})$ then $level \leftarrow level + 1$ else break the while loop.
  3. If $level = l + 1$ then the set j_1 contains the query set.

Algorithm 3. Incremental updating of the AISS Index.

1. For each such item, that it appears both in the new and old version of the multiset, correct the counters on the respective appearance lists.
2. For each such item, that it appears only in the new version of the multiset, create appropriate entry on the respective appearance lists, creating an appearance list if necessary. Increase frequency counters of such items by one.
3. For each such item, that it appears only in the old version of the multiset, delete appropriate entry from the respective appearance lists, deleting appearance lists if they become empty. Decrease frequency counters of such items by one.

After that we need to execute the steps of the algorithm 2 starting with the least frequent items.

In order to perform set subsequence queries, two steps need to be added. First, before we start processing the query, we must transform the query sequence to a multiset using a sequence to multiset transformation. Because such transformation loses information about the order of items in the sequence, a verification phase needs to be added, to prune the false positives. In order for verification phase to work efficiently, we must make an assumption that each of the sequences in the database is placed at a single location on disk and may be easily accessed by rowid. During the verification phase, we access all of the sequences that were returned from index and check whether they fulfill the query conditions or not.

Algorithms for incremental updates are also very simple. Due to the lack of space we will present only the algorithm for updates. Algorithms for insertions and deletions may be easily derived from it. In order to update index after modification of the multiset, perform the steps shown by algorithm 3. For databases of sequences of sets update algorithms are almost the same. The only difference is the necessity of transformation of the updated sequence (both old and new version) to the multiset before proceeding.

The performance of the index depends mainly on its physical implementation. In this paper we propose a way of implementing the AISS index, which uses functionality offered by any commercially available database management system. Both, the dictionary and appearance lists may be represented by a B$^+$-tree or a

Table 2. Synthetic data and experiment parameters

Parameter	Exp.1	Exp.2	Exp.3
size of the domain [items]	150000	150000	150000
item distribution	zipfian and uniform		
minimal set size [items]	1	1	5 – 95
maximal set size [items]	30	30	15 – 105
minimal set number [sets]	1	5 – 95	5
maximal set number [sets]	10	15 – 105	15
number of sequences	10000 – 100000	10000	10000
page/node size [bytes]	4096B	4096B	4096B

B*-tree structure. These structures allow very fast mapping of key values to some values associated with them. Let us consider the following key-value pair. Let the key be a pair $(item, id)$, where *item* is some item from the dictionary and *id* is the multisets unique identifier, and let the *value* be a number of appearances of the *item* in the multiset identified by the *id*. If we assume lexicographic order imposed on key pairs, then the groups of consecutive entries in leaves of the B$^+$-tree will form appearance lists. To read an appearance list of the item x_i we just need to locate the first leaf entry, which corresponds to the item x_i, and read consecutive entries, until we find the first entry for the next appearance list. If we need to locate an entry corresponding to the multiset j on the appearance list of the item x_i, we can easily locate it, because the (x_i, j) forms a key. Notice, that such implementation has other advantages: very easy insertion, deletion and modification of entries, as well as "automatic" removal, or insertion of appearance lists (each list only exists, if there is at least one entry from it stored in the tree).

An additional structure, which maps multiset/sequence id to rowid, is needed to locate multisets or sequences on disk. This is especially important for sequences, for which there is additional phase of verification. Such mapping could be easily performed by the second B$^+$-tree. Frequency counters for items should be stored in memory when database is up, and therefore they do not any need special structures to store.

5 Performance Tests

We have performed three experiments testing the impact of the number of sequences of sets in the database, average number of sets in the sequence, average size of sets in the sequence and average length of the query sequence on the index performance. Performance of index was measured as an average time of query execution, including the time of verification phase. Due to lack of competitors, we compare query processing times when using index only to the brute force solution of scanning the whole database. Table 2 summarizes the parameters in experiments.

The first experiment tested the impact of the number of sequences stored in database on the index performance. Figure 3 presents the performance of the

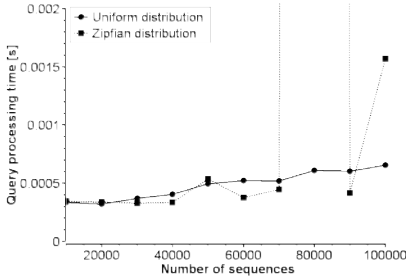

Fig. 3. Average size of sets

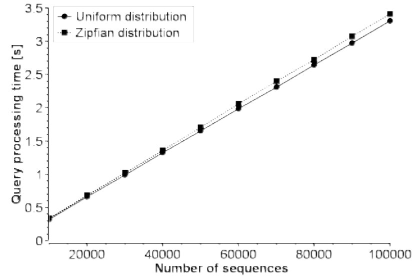

Fig. 4. Average size of sets (no index)

AISS index for zipfian and uniform distributions. Figure 4 presents the same experiments without the index. Analysing Figure 3 one may notice few things. First, query execution time for databases with uniform distribution grows linearly with respect to the number of sequences. Second, for databases with zipfian distribution of items, the trend of growth is also linear, however the query processing times are not "stable". This is caused by random generation of queries. When a short query with frequent items is generated, there is a large set of possible results which need to be verified, and therefore query processing times grow considerably. For example, the peak obtained during querying database of 80000 sequences appeared during processing of a query sequence that contained only a single item. Partial solution to this problem may be based on an observation that when the query sequence contains only a single item, no verification is necessary, as the results obtained from index will be accurate. Queries for databases with uniform distribution of items are more "stable" because there are no such items, that appear in a very large number of sequences, which could be the cause of a long verification phase. When we compare query processing times to those presented on Figure 4 we may notice, that they are three orders of magnitude smaller.

The second experiment tested the impact of the average number of sets in the sequence on the index performance. Figure 5 presents the performance of the AISS index for zipfian and uniform distributions. Figure 6 presents the same experiments without the index. Once again, when analysing Figure 5 one may notice linear dependency of query processing time on average number of elements in the sequence. One may also notice divergence of query processing times for zipfian and uniform distributions. Better performance of the AISS index for zipfian distribution is caused by the optimization described in Section 4, in which we start processing the query with the least frequent item of the query multiset. Once again, when we compare query processing times to those presented on Figure 6 we may notice, that they are three orders of magnitude smaller.

The third experiment tested the impact of the average sizes of sets in the sequences on the index performance. Figure 7 presents the performance of the AISS index for zipfian and uniform distributions. Figure 8 presents the same experiments without the index. Once again we may notice linear dependency of

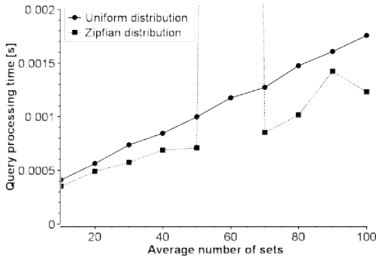

Fig. 5. Average number of sets

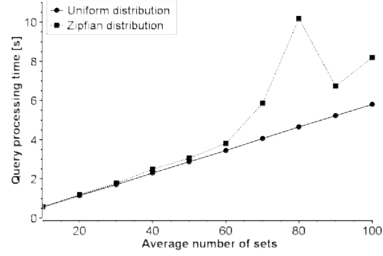

Fig. 6. Average number of sets (no index)

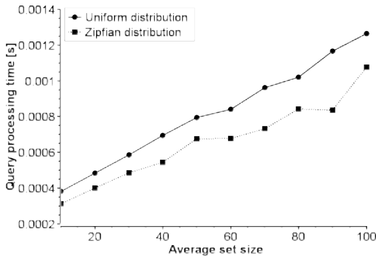

Fig. 7. Average size of sets

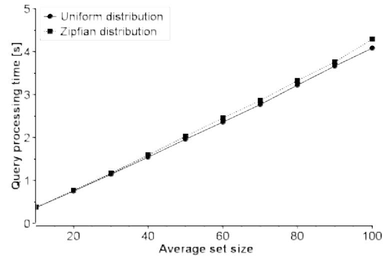

Fig. 8. Average size of sets (no index)

query processing time on average size of sets in the sequence and three orders of magnitude improvent over full scan of database.

During the experiments, the verification phase took from 5% to 50% of the query processing time with the exception of the observed peaks, when it took about 95% of the query processing time.

Let us notice, that theoretically we could create other solutions for indexing sequences of sets based on other solutions for indexing sets. However, as we have experimentally shown in [2] Inverted File Index outperforms all other solutions in subset queries and therefore we know in advance, that other solutions, will not be as good as the one presented in this paper.

6 Conclusions

To the best of authors' knowledge, the AISS index presented in this paper is the only index for sequences of sets supporting set subsequence queries without timestamps. We have presented both: logical and physical structure as well as algorithms for set subsequence query execution and incremental updates. Experiments show that the ratio of speed-up for set subsequence queries is three to four orders of magnitude when compared to brute-force approach.

In the future we plan to perform additional extensive experiments to determine weak points of our index. We also plan to design a new index, which is able to answer set subsequence queries without verification and apply such index to improve speed of sequential pattern mining algorithms.

References

1. R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pages 69–84. Springer-Verlag, 1993.
2. W. Andrzejewski, Z. Królikowski, and M. Morzy. Performance evaluation of hierarchical bitmap index supporting processing of queries on setvalued attributes (polish). *Archiwum Informatyki Teoretycznej i Stosowanej*, 17(4):273–288, 2005.
3. W. Andrzejewski, T. Morzy, and M. Morzy. Indexing of sequences of sets for efficient exact and similar subsequence matching. In *Proceedings of the 20th International Symposium on Computer and Information Sciences*, pages 864–873. Springer-Verlag, 2005.
4. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 419–429. ACM Press, 1994.
5. S. Helmer and G. Moerkotte. A study of four index structures for set-valued attributes of low cardinality, 1999.
6. E. Keogh, S. Lonardi, and C. A. Ratanamahatana. Towards parameter-free data mining. In *KDD '04: Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 206–215. ACM Press, 2004.
7. A. Lerner and D. Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.
8. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademia Nauk SSSR*, 163(4):845–848, 1965.
9. N. Mamoulis and M. L. Yiu. Non-contiguous sequence pattern queries. In *Proceedings of the 9th International Conference on Extending Database Technology*, 2004.
10. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics, 1990.
11. A. Nanopoulos, Y. Manolopoulos, M. Zakrzewicz, and T. Morzy. Indexing web access-logs for pattern queries. In *WIDM '02: Proceedings of the 4th international workshop on Web information and data management*, pages 63–68. ACM Press, 2002.
12. R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *Symposium on Principles of Database Systems*, 2001.
13. M. Vlachos, M. Hadjieleftheriou, D. Gunopulos, and E. Keogh. Indexing multidimensional time-series with support for multiple distance measures. In *ACM KDD*, 2003.
14. H. Wang, C.-S. Perng, W. Fan, S. Park, and P. S. Yu. Indexing weighted-sequences in large databases. In *Proceedings of International Conference on Data Engineering*, 2003.
15. P. Weiner. Linear pattern matching algorithms. In *Proceedings 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.
16. B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 385–394. Morgan Kaufmann Publishers Inc., 2000.