How to Improve Efficiency of Analysis of Sequential Data?[1]

Witold Andrzejewski[2], Zbyszko Królikowski[2], Tadeusz Morzy[2]

Abstract: In order to extract useful knowledge from large databases of sales data, data mining algorithms (the so-called market basket analysis) are used. Unfortunately, these algorithms, depending on data and parameters, may generate a large number of patterns. Analysis of these results is performed by the user and involves executing a lot of queries on complex data types that are not well supported by commercially available database management systems. To increase efficiency of analysis of data mining results, new index structures need to be developed. In this paper we propose the indexing scheme for non-timestamped sequences of sets, which supports set subsequence queries. Experimental evaluation of the index proves the feasibility and benefit of the index in query processing.

Keywords: data mining, indexing, market basket analysis

1. Introduction

Analysis of large volumes of data (such as sales data) is impossible to do "by hand". To solve this problem, a large number of different techniques for *knowledge discovery in databases* (also known as *data mining*) have been developed. The purpose of these techniques is to discover new, useful, correct and understandable patterns in large databases. Such patterns are very useful in many applications, such as: science, medicine, finances and marketing.

Many different types of data may be analysed using data mining algorithms including: stock prices, cash register data or web server logs. Particularly interesting is the analysis of sales data also known as *market basket analysis*. Through market basket analysis one may obtain frequent itemsets, association rules or sequential patterns. Unfortunately, very often, the number of discovered patterns is very large, and thus they need to be stored in a separate database for further analysis. Such analysis involves searching for either itemsets or sequential patterns which are in some way related to the user specified set or sequence of sets. Possible relations include sequence or set containment and sequence or set similarity. All of the aforementioned patterns have a complex structure. Frequent itemsets are sets of categorical data, association rules are represented by two itemsets and sequential patterns are sequences of itemsets. Such complex data types, although possible to store, are not well supported in commercial database systems. Thus, the search for sets and sequences is very costly.

[1] The paper is sponsored by The Polish Ministry of Science and Higher Education, grant no. N206 011 32/1221.

[2] Institute of Computing Science, Poznan University of Technology, Piotrowo 2, 60-965 Poznań
e-mail: {wandrzejewski,zkrolikowski,tmorzy}@cs.put.poznan.pl

Concluding, there is evidently a need to design efficient, possibly general, indexing schemes for sets and sequences of sets. Several indexing schemes for sequences and sets have been proposed so far. Most of indexes for sequences were designed either for time series (see Agrawal et al., 1993 and Faloutsos et al., 1994) or sequences of atomic values (see Wang et al, 2003 and Mamoulis et al., 2004). Several indexes for sets were proposed as well in Morzy et al., 2003, Aggarwal et al., 1999, Andrzejewski et al., 2003, Ishikawa et al., 1993, Hellerstein et al., 1994, Deppisch, 1986, Helmer et al., 1999, Goczyła, 1997 and Faloutsos et al., 1984.

However the aformentioned solutions could only be used either to index sets or to index sequences of non-complex data. Almost nothing has been done with regard to more general indexing of sets and sequences of sets. According to our knowledge, the only other general solutions developed so far were proposed by us in Andrzejewski et al., 2005 and Andrzejewski et al., 2006 (two papers).

The original contribution of this paper is the proposal of a new indexing scheme capable of efficient retrieval of sets and sequences of non-timestamped sets based on sequence containment. We present the physical structure of the index and develop algorithms for query processing. The index has a very simple structure and may be easily implemented over existing database management systems.

2. Related work

Most of research on indexing of sequential data is focused on three distinct areas: indexing of time series, indexing of strings (DNA and protein sequences), and indexing of 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 (see Keogh et al., 2001), compression-based metrics (see Keogh et al., 2004) and dynamic time warping metrics (see Vlachos et al., 2003)). Often, a technique for reduction of the dimensionality of the problem is employed, such as discrete Fourier transform see Agrawal et al., 1993 and Faloutsos et al., 1994). 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 (see Levenshtein, 1965), and index structures are built on suffix tree (see Ukkonen, 1995 and Weiner, 1973) or suffix array (see Manber, 1990).

Indexing of web logs data differs significantly from indexing of strings. The main difference is that each element in such a sequence is assigned a timestamp that must be taken into consideration when processing a query. Several different approaches have been considered so far. The first one used a special transformation technique to transform the original problem into the well-researched problem of indexing of sets (see Nanopoulos et al., 2002). Other approaches include ISO-Depth index (see Wang et al., 2003) which is based on a trie structure and SEQ-Join index (see Mamoulis, 2004) 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 Andrzejewski et al. (2005)

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. The SeqTrie index , presented in Andrzejewski et al. (2006), is based on an idea similar to the Generalized Iso-Depth Index, however it was designed to support non-timestamped set subsequence queries. The AISS Index proposed in Andrzejewski et al. (2006) was designed to support non-timestamped set subsequence queries on sequences of sets and subset queries on multisets, and uses a structure based on the inverted file.

3. Basic definitions & 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. We define the *length* of a sequence as a number of sets in the sequence, and denote it $|\mathcal{S}|$. We also define the *size* of the sequence as the number of items in the sequence and denote it $\|\mathcal{S}\|$. Given the item x and a sequence $\mathcal{S}$ we say that the item x is *contained* within the sequence $\mathcal{S}$, denoted $x \in \mathcal{S}$, if there exists any itemset in the sequence such that it contains the given item. 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 removing of some of items from the elements, and removing of empty elements, if such occur. We also say, that if, and only if $\mathcal{T} \sqsubseteq \mathcal{S}$, the sequence $\mathcal{T}$ is *contained* within the sequence $\mathcal{S}$. Conversely, we say that the sequence $\mathcal{S}$ *contains* the sequence $\mathcal{T}$ and that $\mathcal{S}$ is a *supersequence* of $\mathcal{T}$.

We define a *database*, denoted $\mathcal{DB}$, as a set of sequences, called *database sequences*. Each database sequence in the database has a unique *identifier*. Without the loss of generality we assume those identifiers to be positive integers. A database sequence identified by the number id is denoted $\mathcal{S}^{id}$. Let the *support* of the item x, denoted $supp(x)$, be the number of sequences that contain the item. Formally $supp(x) = |\{\mathcal{S}^{id} \in \mathcal{DB} : x \in \mathcal{S}^{id}\}|$. 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. $\{id : \mathcal{S}^{id} \in \mathcal{DB} \wedge \mathcal{Q} \sqsubseteq \mathcal{S}^{id}\}$. Such sets are called *result sets*. Our problem is to design an auxiliary structure (an index) for database tables storing sequences of sets, and an algorithm utilizing this structure, which allows efficient set subsequence query processing.

4. The FIRE Index

In this section we present our new index for sequences of non-timestamped sets. The idea of the index is based on the well known inverted file index. The new index may be used to increase performance of set subsequence queries.

Basic Inverted File structure, which may be used for indexing itemsets, is composed of two parts: *dictionary* and *appearance lists*. The dictionary is the list of all

Table 1. Examples

(a) Exemplary database

Id	Sequence
1.	$\langle\{1,2,3\},\{1,5\},\{4,6\}\rangle$
2.	$\langle\{2,6\},\{1,5\}\rangle$
3.	$\langle\{1,2,3\},\{3\},\{3,4,5\}\rangle$

(b) FIRE index for an exemplary database

Dictionary (items support)					
1 (3)	2 (3)	3 (2)	4 (2)	5 (3)	6 (2)
Appearance lists					
(1,1)	(1,1)	(1,1)	(1,3)	(1,2)	(1,3)
(1,2)	(2,1)	(3,1)	(3,3)	(2,2)	(2,1)
(2,2)	(3,1)	(3,2)		(3,3)	
(3,1)		(3,3)			

the items that appear at least once in the database. Each item has an appearance list associated with it. Given the item x, the appearance list associated with item x lists identifiers of all the sets from database, that contain that item. Inverted file index is particurarly efficient in supporting subset queries. Such queries are performed by reading appearance lists of all of the items from the query set, and finding their intersection.

In order to be able to store sequences of sets, we propose a straightforward modification. On appearance lists associated with items, we store sequence identifiers, as well as elements' number in this sequence. We also require, that the entries on appearance lists were ordered first by the identifier of the sequence, and next by the elements number. Notice, that such modification allows us to store full information about sequences of sets. Exemplary database and index are shown on tables 1(a) and 1(b) respectively.

Basic idea for the set subsequence query algorithm is as follows. Let us consider the appearance list of any of the items in the query sequence. It is easy to notice, that the set of different sequences, which are referred to by the entries on this list, is the upper bound on the result set. The best (smallest) upper bound may be obtained from the appearance list of the item with the lowest support. Any further processing of the query should just narrow the first estimate of the result set. Therefore, the next step of the algorithm, should be to analyze the entries on the remaining appearance lists to verify, if the sequences from the previously obtained upper bound, are indeed results of the query. If we assume that there is no correlation between the items, then the best pruning may obtained if the next analysed appearance list is the list corresponding to the next item with the lowest support. Therefore, we process the items from the query sequence in the order of their support. Let us now consider the main loop of the algorithm. As stated before, we read the appearance list of the item from the query sequence, which has the lowest support. For each of the entries on this list, we check if the sequence, which is referred to by the entry is indeed a supersequence of the query sequence. The above discussion is summarized by the algorithm 1

In the previous discussion we omitted the problems associated with the order of the items in the query sequence. We shall address them now. Let us start with the main loop of the algorithm. Consider a situation in which the item x from the query

Algorithm 1 An algorithm for set subsequence queries

INPUT: Query sequence $\mathcal{Q}$.
OUTPUT: Result set $results$.

1. Allocate temporary table called map of size equal to $|\mathcal{Q}|$ and fill it with NULLs.
2. $lastId \leftarrow -1$
3. Convert the query sequence $\mathcal{Q}$ to the sequence of pairs $\langle x, s\rangle_i$, where x is an item, and s is the number of the element in the query sequence. These pairs should be ordered by the increasing support of the items. For the sake of simplicity, we shall denote this transformed query sequence as $\mathcal{Q}^T$.
4. For each of the entries (id, sn) on the appearance list of the item x from the pair $\langle x, s\rangle_1$ (corresponding to the item with the lowest support), such that $sn >= s$ and $id > lastId$, perform the following steps:
   (a) $map[s] \leftarrow sn$
   (b) Call function $checkSub(2, id)$ (algorithm 2).
   (c) If the result of the last call to the checkSub function is TRUE, then:
      i. $lastId \leftarrow id$
      ii. Store id in the result set $results$.
   (d) $map[s] \leftarrow null$

sequence, which has the lowest support, is in the element s_i of the query sequence. Let the $\mathcal{S}$ be any sequence such, that $\mathcal{Q} \sqsubseteq \mathcal{S}$. Because $\mathcal{Q} \sqsubseteq \mathcal{S}$, the sequence $\mathcal{S}$ must contain an element s'_j such that $s_i \subseteq s'_j$. It is easy to notice, that there must exist such s'_j that $j \geq i$. The above discussion is reflected by the condition $sn \geq s$ in the point 4 of the algorithm 1. The verification algorithm is based on similar observations. The verification algorithm, given the sequence number, reads the appearance list of the second item with the lowest support. Only entries referring to the analysed sequence need to be analysed (other entries are irrelevant, because we verify only a single sequence). Next, we further narrow the set of entries (from the appearance list) which need to be analysed, by using the information about the order of the items. We will describe this process in detail in the next paragraph. For now, lets just assume, that the verification algorithm determines which entries on the appearance list of the next item from the query sequence refer to the elements, which do not violate the order of elements in the query sequence. For each of the entries on the appearance list, which weren't eliminated, we recursively call the verification algorithm and analyze the appearance list of the third item from the query sequence. This process is repeated until we find an entry in some appearance list for each of the items from the query sequence (this is possible only if the database sequence is the supersequence of the query sequnce), or determine, that we can't find such entries. The above discussion is summarized by the algorithm 2

The performance of the algorithm depends heavily on the algorithms, which eliminate entries from the appearance lists. In order to perform such pruning we allocate an auxiliary table called *map* of the size equal to $|\mathcal{Q}|$. This table is used to "map" query sequence element number, to the database sequence element number. The verification algorithm can use the information stored in this table to narrow the relevant set of entries on the next appearance list. Let us first consider a situation, in which the next analysed item from the query sequence is in the same element, as one of the previously analysed items. It is easy to notice, that, by reading the appropriate entry in the table *map*, we know in which element of the

Algorithm 2 Function checkSub used by the algorithm 1, which verifies if the candidate sequence is indeed the supersequence of the query sequence.

ASSUMPTIONS: We assume, that the transformed query sequence $\mathcal{Q}^T$, result set $results$ and temporary table map are globally accessible.
INPUT: Recursion level $level$, candidate sequence identifier id.
OUTPUT: TRUE, if the sequence id is the supersequence of the query sequence, FALSE if not.

1. If $level > \|\mathcal{Q}\|$ then return TRUE. If the condition is not satisfied, then perform the following steps:
2. Retrieve the pair $\langle x, s \rangle_{level}$ from $\mathcal{Q}^T$.
3. If $map[s] \neq NULL$ then perform the following steps:
   (a) Check on the appearance list of the item x if it contains the entry $\{id, map[s]\}$.
   (b) If it doesn't, return false.
   (c) If it does, return the value returned by the function call: $checkSub(id, level + 1)$.
4. If $map[s] = NULL$ then perform the following steps:
   (a) $l \leftarrow lowerBound(s)$ (algorithm 3)
   (b) $u \leftarrow upperBound(s)$ (algorithm 4)
   (c) For each of the entries (i, sn) on the appearance list of the item x, such that $sn \geq l \wedge sn \leq u$ perform the following steps:
      i. $map[s] \leftarrow sn$
      ii. If the value returned by the call to the function $checkSub(id, level + 1)$ is TRUE then return TRUE.
      iii. $map[s] \leftarrow NULL$
   (d) Return FALSE.

verified sequence should the next query sequence item be stored, thereby narrowing the relevant entries on the appearance list to just one. Now, let us consider the situation, in which the analysed item from is not from one of the mapped elements. Because the element is not mapped we may not narrow relevant element numbers to just one. However, we may calculate the upper and lower bounds of the element number. Steps for calculating these bounds are shown on algorithms 3 and 4. Due to lack of space, we are not able to explain these algorithms, however, they are very simple and straightforward.

Algorithm 3 Function *lowerBound* calculating the smallest possible set mapping for the given set number in the query sequence.

ASSUMPTIONS: We assume, that the temporary table map is globally accessible.
INPUT: Query sequence element number s.
OUTPUT: The least element number in the database sequence, which may be analysed as a potential superset of the element s.

1. Find the largest index in the table map, which is smaller then s, and the value stored in the table under this index is not NULL.
2. If such index does not exist, return s.
3. If such index exists, store it in the variable i.
4. Return $map[i] + s - i$.

The algorithm for incremental updates of the index is straightforward. To reflect changes in database, just remove entries on appearance lists which correspond to the removed items, and add new entries, which correspond to the added items. Detailed steps for updating the sequences are presented by the algorithm 5

Algorithm 4 Function *upperBound* calculating the largest possible set mapping for the given set number in the query sequence.

ASSUMPTIONS: We assume, that the temporary table *map* is globally accessible.
INPUT: Query sequence element number s.
OUTPUT: The largest possible element number in the database sequence, which may be analysed as a potential superset of the element s.

1. Find the smallest index in the table *map*, which is larger then s, and the value stored in the table under this index is not NULL.
2. If such index does not exist, return ∞.
3. If such index exists, store it in the variable i.
4. Return $map[i] + s - i$.

Algorithm 5 An algorithm for incremental updates of the index

INPUT: Old version of the sequence $\mathcal{S}^{old}$ (if inserting $\|\mathcal{S}^{old}\| = 0$), new version of the sequence $\mathcal{S}^{new}$ (if deleting $\|\mathcal{S}^{new}\| = 0$), sequence identifier id.
OUTPUT: Modified index.

1. Let $O = \left\{(x, s) : x \in \mathcal{A}^{old} \wedge \text{s is the set number of an item x}\right\}$
2. Let $N = \{(x, s) : x \in \mathcal{A}^{new} \wedge \text{s is the set number of an item x}\}$.
3. For each $(x, s) \in O \setminus N$ delete from the appearance list of the item x entry (id, s).
4. For each $(x, s) \in N \setminus O$ insert into the appearance list of the item x entry (id, s).

We shall now discuss the physical structure of an index. It is easy to notice, that the algorithm for query execution reads the index in three different ways: scan the whole appearance list, scan appearance list entries refering to a single sequence and only a given interval of elements and read a single entry on the appearance list (check if a given entry is on the list, or not). Our physical structure of the index should support such access methods and furthermore, it should allow us to easily insert, delete and sort entries on the appearance lists. Let us consider the slightly modified B^+ tree which stores only keys (no data is associated with them). Let keys be the triples $\langle x, id, s\rangle$, where x is the items identifier, id is the identifier of a sequence and s is the number of the element in the sequence id in which the item x is contained. Let the order imposed on those triples be the lexicographic one, first by items, then by sequence identifier and finally by the set number. Such B^+ tree has all of the required properties. All of the aforementioned index access types can be represented as either range or point queries to the B^+ tree index. Notice, that such implementation has other advantages: very simple 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).

5. Performance tests

We have performed three different experiments testing impact of the: number of sequences, average sequence size and average element size on the index performance. For each of the experiments we built 20 databases, 10 of which were built using the uniform distribution and the other 10 were built using the zipfian distribution

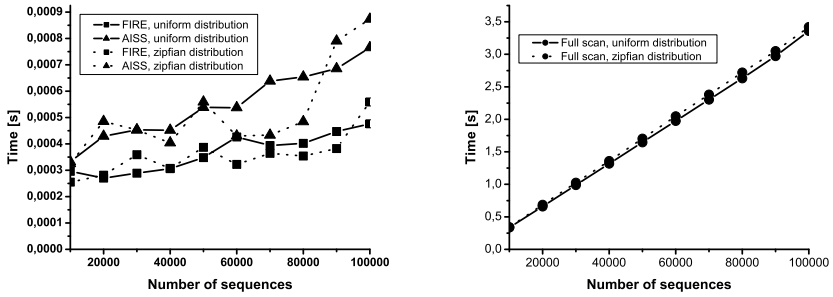
Table 2. Experiment parameters

Parameter	Exp.1	Exp.2	Exp.3
number of different 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 sequence size [items]	1	5-95	5
maximal sequence size [items]	10	15-105	15
number of sequences	10000-100000	10000	10000
page/node size [bytes]	4096B	4096B	4096B

for itemset generation. For each of the databases we randomly built 40 queries. During experiments these sets of queries were executed 10 times. Obtained query processing times were averaged. We compare the performance of the FIRE index to the performance of the only other incrementally updatable index for sequences of sets, the AISS index. Table 2 summarizes the experiment parameters.

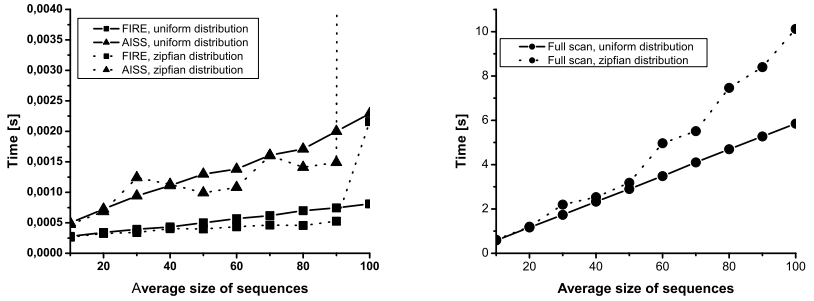
The first experiment tested the impact of the number of sequences stored in database on the index performance. Figure 1(a) presents the performance of the FIRE index for zipfian and uniform distributions in comparison to the performance of the AISS index. Figure 1(b) presents the same experiments without index. Analysing the figure 1(a) one may notice a few things. First, the query processing times of both, the AISS index and the FIRE index depend linearly on the number of sequences stored in the database. Second, the query processing times of the FIRE index are smaller than those of the AISS index. Third, query processing times do not depend significantly on the distribution of the items. Fourth, when we compare query processing times to those presented on Figure 1(b) we may notice, that they are three orders of magnitude smaller.

The second experiment tested impact of the average size of sequences stored in the database on the index performance. Figure 2(a) presents the performance of the FIRE index for zipfian and uniform distributions in comparison to the performance of the AISS index. Figure 2(b) presents query processing times for the same experiments when full scan of database is used. Let us consider the results presented on the Figure 2(a). Once again we may observe linear dependency of query processing times on the average size of sequences stored in the database. The FIRE index, as before, processes queries faster then the AISS index. However, we may notice, that query processing times for databases with the zipfian distribution are a bit smaller then the query processing times for the uniform distribution. This may be explained by the following observations. When the zipfian distribution is used, some appearance lists are very long, but there are also multiple very short appearance lists. Because we start query processing with the items with the lowest support (and probably with shortest appearance lists), we obtain smaller sets of sequences to verify in the main loop of the query processing algorithm, which



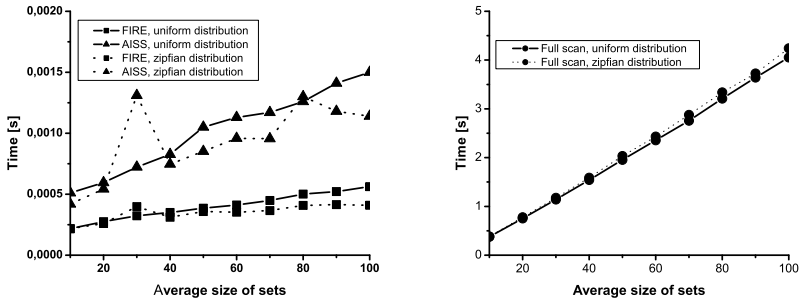
(a) Using index (b) Not using index

Figure 1. Number of sequences



(a) Using index (b) Not using index

Figure 2. Average size of sequences



(a) Using index (b) Not using index

Figure 3. Average size of sets

improves performance of the index. One may also make another interesting observation: the trend of growth of query execution times, when the zipfian distribution is used, is not as stable as in experiments with uniform distribution. This is particurarly apparent for the average query execution times on databases with sequences of average size equal to 100. In this case, one of the randomly generated queries was very short, and composed of only frequent items. As was pointed out in Andrzejewski, et al. (2006), this is a very bad case for the AISS index, as it requires to retrieve and analyze a large part of the database. As we can clearly see, the FIRE index behaves much better in this case. When we compare query processing times to those presented on Figure 2(b) we may notice, that they are three orders of magnitude smaller.

The third experiment tested impact of the average size of sets in the database on the index performance. Figure 3(a) presents the performance of the FIRE index for zipfian and uniform distributions in comparison to the performance of the AISS index. Figure 3(b) presents the same experiments without index. Let us analyse the Figure 3(a). The dependency of the query execution times on the average size of sets is also linear. As in previous experiments FIRE index is faster then the AISS index, query processing times are a little bit smaller for the databases with the zipfian distribution of items, and finally, when compared to the query processing times observed on the Figure 3(b), FIRE index is three orders of magnitude faster then the full scan of database.

6. Conclusions & Future work

We have proposed a new indexing scheme capable of retrieving of sequences of sets based on sequence containment. We have proposed the logical and physical structure, and we have developed the algorithms for index construction, set subsequence query processing and incremental updates of the index. Our index is capable of storing full information about indexed sequences and therefore it doesn't need to make any assumptions as to the database physical and logical structure. As we have experimentally shown, the FIRE index is faster then the AISS index and processes set subsequence queries three orders of magnitude faster then the full scan of database. Query processing times are also almost independent on the distribution of the items (they may be even shorter when the items have skewed distribution).

In future we plan on designing algorithms for other classes of queries for sequences of sets as well as performing extensive performace tests on real world data to determine more of the possible application domains of our index. We also plan on designing compression schemes of our index, to lessen the number of disk accesses required to process the queries.

References

Aggarwal, C. C., Wolf, J. L. and Yu, P. S. (1999) A new method for similarity indexing of market basket data. *Proceedings of the ACM SIGMOD Conference on Management of Data*, 407–418, New York, ACM Press.

Agrawal, R., Faloutsos, C. and Swami, A. N. (1993) Efficient similarity search in sequence databases. *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, 69–84, Chicago, Springer Verlag.

Andrzejewski, W., Gaertig, P., Radom, M. and Antoniewicz, M. (2003) Opracowanie i analiza wydajnościowa indeksu dla przybliżonego wyszukiwania podzbiorów danych (polish).

Andrzejewski, W. and Morzy, T. (2006) AISS: An index for non timestamped set subsequence queries. *Proceedings of the 8th International Conference on Data Warehousing and Knowledge Discovery*, 503–512, Cracow, Springer.

Andrzejewski, W. and Morzy, T. (2006) SeqTrie: An index for data mining applications. *Proceedings of the 2nd ADBIS Workshop on Data Mining and Knowledge Discovery*, 13–25.

Andrzejewski, W., Morzy, T. and Morzy, M. (2005) Indexing of sequences of sets for efficient exact and similar subsequence matching. *Proceedings of the 20th International Symposium on Computer and Information Sciences*, 864–873, Istanbul, Springer-Verlag.

Deppisch. U. (1986) S-Tree: a dynamic balanced signature index for office retrieval. *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, 77–87, Pisa, ACM Press.

Faloutsos, C., Christodoulakis, S. (1984) Signature files: an access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems (TOIS)* **2**, 4, 267–288.

Faloutsos, C., Ranganathan, M. and Manolopoulos, Y. (1994) Fast subsequence matching in time-series databases. *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 419–429, Mineapolis, ACM Press.

Goczyła, K. (1997) The Partial-Order Tree: A New Structure for Indexing on Complex Attributes in Object-Oriented Databases. *Proceedings of the 23rd Euromicro Conference*, 47–54, Budapest, IEEE.

Hellerstein, J. M. and Pfeffer, A. (1994) The RD-Tree: an index structure for sets. *Technical Report 1252*, University of Wisconsin at Madison.

Helmer, S. and Moerkotte, G. (1999) A study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal — The International Journal on Very Large Data Bases*, **12**, 3, 244–261.

Ishikawa, Y., Kitagawa, H. and Ohbo, N. (1993) Evaluation of signature files as set access facilities in oodbs. *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Washington DC, 247–256, ACM Press.

Keogh, E., Chakrabarti, K., Pazzani, M. and Mehrotra, S. (2001) Locally adaptive dimensionality reduction for indexing large time series databases. *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, 151–162, Santa Barbara, ACM Press.

Keogh, E., Lonardi, S. and Ratanamahatana, C. A. (2004) Towards parameter free data mining. *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, 206–215, Seattle, ACM Press.

Levenshtein, V. I. (1965) Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademia Nauk SSSR*, **163**, 1, 845–848.

Mamoulis, N. and Yiu, M. L. (2004) Non-contiguous sequence pattern queries. *Proceedings of the 9th International Conference on Extending Database Technology*, LNCS **2992**, 783–800.

Manber, U. and Myers, G. (1990) Suffix arrays: a new method for on-line string searches. *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, 319–327, Philadelphia, Society for Industrial and Applied Mathematics.

Morzy, M. and Królikowski, Z. (2003) Approximate queries on set-valued attributes. *T. Morzy and B. D. Czejdo, editors, The 1st Symposium on Databases, Data Warehouses and Knowledge Discovery*, 87–96, Baden-Baden, Germany, Scientific Publishers OWN.

Nanopoulos, A., Manolopoulos, Y., Zakrzewicz, M. and Morzy, T. (2002) Indexing web access-logs for pattern queries. *Proceedings of the 4th international workshop on Web information and data management*, 63–68, Virginia, ACM Press.

Ukkonen, E. (1995) On-line construction of suffix trees. *Algorithmica*, **14**, 3, 249–260.

Vlachos, M., Hadjieleftheriou, M., Gunopulos, D. and Keogh, E. (2003) Indexing multidimensional time-series with support for multiple distance measures. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 216–225, Washington, ACM Press

Wang, H., Perng, C.-S., Fan, W., Park, S., and Yu, P. S. (2003) Indexing weighted-sequences in large databases. *Proceedings of International Conference on Data Engineering*, 63–74, Bangalore, IEEE Computer Society.

Weiner, P. (1973) Linear pattern matching algorithms. *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, 1–11, Iowa, IEEE