Control and Cybernetics

vol. **38** (2009) No. 1

How to improve efficiency of analysis of sequential data?*†

by

Witold Andrzejewski, Zbyszko Królikowski and Tadeusz Morzy

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

Abstract: Many of todays database applications, including market basket analysis, web log analysis, DNA and protein sequence analysis utilize databases to store and retrieve sequential data. Commercial database management systems allow to store sequential data, but they do not support efficient querying of such data. To increase the efficiency of analysis of sequential data new index structures need to be developed. In this paper we propose an indexing scheme for non-timestamped sequences of sets, which supports set subsequence queries. Our contribution is threefold. First, we describe the index logical and physical structure, second, we provide algorithms for set subsequence queries utilizing this structure, and finally we perform experimental evaluation of the index, which proves its feasibility and advantages in set subsequence query processing.

Keywords: sequential data, indexing, market basket analysis.

1. Introduction

Many of current database applications process complex data types such as: sets, sequences, time series, objects, semistructured data and graphs. Such application domains include, but are not limited to: bioinformatics, market basket analysis, web server event logging or stock price analysis. In bioinformatics strings of symbols representing either DNA or protein sequences are processed. The analysis is based on finding sequences or subsequences similar to the query sequences. Market basket analysis is based on analysis of either sets of bought items or sequences of sets of items bought by a single customer in some period of time. Queries issued in market basket analysis are in most cases subset queries or set subsequence queries. Web server logs are sequences of timestamped events

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

†Submitted: June 2008; Accepted: October 2008.

which may be analysed in order to find frequent user behaviour habits or suspicious user activities. Logically web logs are sequences of items, where each item represents some event. Histories of stock prices are represented as time series, sequences of timestamped real values. Analysis of such histories is based on retrieval of subsequences of stored time series whose shape is similar to the user specified shape. Notice that most of the aforementioned data types are sequential and their processing is not trivial. Commercially available object-relational database management systems allow to store such sequences (strings, sequences of items, sequences of sets of items and time series), but they do not support efficient querying of such data. Thus, the processing of queries for databases storing sequences is very costly.

Several indexing schemes for sequences have been developed. Most of indexes for sequences were designed either for time series (see Agrawal, Faloutsos and Swami 1993, and Faloutsos, Ranganathan, and Manolopoulos, 1994), DNA and protein sequences (strings) (see Ukkonen, 1992, 1995, Weiner, 1973, McCreight, 1976, and many more) or sequences of items (see Wang et al., 2003, and Mamoulis and Yiu, 2004). However, the aforementioned solutions can only be used to index sequences of atomic values (items, real values or symbols of either amino acid or nucleotides). Almost nothing has been done with regard to more general indexing of sequences of sets. According to our knowledge, the only solutions for sequences of sets developed so far, were proposed by us in Andrzejewski, Morzy and Morzy (2005), and Andrzejewski and Morzy (2006). However, the Generalized ISO-Depth Index (Andrzejewski et al., 2005) was designed to support sequences of sets with timestamps, which is different type of sequences from the one considered in this paper, the SEQ-Trie index (Andrzejewski and Morzy, 2006) is not incrementally updatable, whereas the AISS index (Andrzejewski and Morzy, 2006) is an approximate index and, depending on the physical database structure, may require a costly verification phase.

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. Our contribution is threefold. First, we propose the logical and physical structure of the index which may be easily implemented over existing database management systems. Second, we provide algorithms for set subsequence queries, utilizing this structure, as well as algorithms for incremental updates of the index, and finally we perform experimental evaluation of the index, which proves its feasibility and advantage in set subsequence query processing.

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 the search 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, and Yi and Faloutsos, 2000), compression-based metrics (see Keogh, Lonardi and Ratanamahatana, 2004) and dynamic time warping metrics (see Vlachos et al., 2003). Often, a technique for reduction of dimensionality of the problem is employed, such as discrete Fourier transform (see Agrawal, 1993, and Faloutsos and Swami, 1994). String indexes usually support search 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, 1992, 1995, Weiner, 1973, and McCreight, 1976) or suffix array (see Manber and Myers, 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 and Yiu, 2004), which uses a set of relational tables and a set of B^+-tree indexes.

Indexing of sets is a well researched subject. Many indexes were developed including, but not limited to: S-Tree (see Deppisch, 1986), Signature File (see Faloutsos and Christodoulakis, 1984), Partial Order Tree (see Goczyła, 1997), RD-Tree (see Hellerstein and Pfeffer, 1994), Hierarchical Bitmap Index (see Andrzejewski et al., 2003) and Inverted File (see Helmer and Moerkotte, 1999).

Recently, works on sequences of categorical data were extended to sequences of sets. The Generalized ISO-Depth Index proposed in Andrzejewski, Morzy and Morzy (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 and Morzy (2006b), 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 and Morzy (2006a) 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 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_p, $p = 1, 2, \ldots, n$ are itemsets. Each itemset in the

sequence is called an *element* of a sequence. Given an element s_p we say that this element has a *position* p. Given two elements s_p and s_r where $p < r$ we say that element s_p has a lower position than element s_r. Conversely, we say that element s_r has a greater position than s_p. Each element s_p of a sequence $\mathcal{S}$ is denoted $\{x_1, x_2, \ldots, x_n\}$, where x_i, $i \in \langle 1, n\rangle$ are items. We define the *length* of a sequence as a number of elements 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 element 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 some of items from the elements, and removing 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

4.1. Logical index structure

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.

The basic inverted file structure, which may be used for indexing databases of 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 x, the appearance list associated with item x lists identifiers of all the sets from database, that contain that item. Inverted file index is particularly 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 id, as well as element position r in this sequence. We also require that the entries on appearance lists be ordered first by the identifier of the sequence, and next by the element position. Notice that such modification allows us to store full information about sequences of sets. We denote the appearance list of the item x as $\mathcal{L}^x$. By $(id, r) \in \mathcal{L}^x$ we denote the fact that the entry (id, r) is stored on the appearance list of the item x. Exemplary database and index are shown in Table 1(a) and 1(b), respectively.

Table 1. Examples

(a) Exemplary database

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

(b) FIRE index for an exemplary database

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

4.2. Algorithms

We present two algorithms, utilizing the FIRE index structure to process set subsequence queries (the so-called recursive and non-recursive algorithms), as well as an algorithm for incremental updates of the index.

The main idea for the recursive set subsequence query algorithm is based on the following observations. Let us consider the appearance list $\mathcal{L}^x$, where x is 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, i.e. the set $\{id : (id, r) \in \mathcal{L}^x\}$, is the upper bound on the result set (because supersequences must contain all of the items from the query sequence). Such upper bound may be obtained for any item such that $x \in \mathcal{Q}$, however, the best (smallest) upper bound may be obtained from the appearance list $\mathcal{L}^x$ 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 check if the sequences from the previously obtained upper bound fulfill the query conditions. Notice that though the upper bounds for each appearance list $\mathcal{L}^x$, $x \in \mathcal{Q}$, may be different, identifiers of supersequences of the query sequence must be on all of them. Consequently, processing of the consecutive appearance lists should involve checking if they contain entries corresponding to the currently verified sequence from the first upper bound. If we assume that there is no correlation between the items, then the best pruning may be obtained if the

next analysed appearance list is the list corresponding to the next item with the lowest support, as the number of common sequence identifiers would be the smallest. Therefore, we should process the items from the query sequence in the order of their support. To obtain the proper order of processing of the items, we should transform the query sequence $\mathcal{Q}$ to the sequence of pairs $\langle x, p \rangle_i$, where x is an item, and p is the position of the element in the query sequence. These pairs should be ordered by the increasing support of the items. If an item appears more then once in the query sequence, we should use the position of the element for disambiguation. We shall denote this transformed query sequence as $\mathcal{Q}^T$.

Let us now consider the main loop of the algorithm. The main loop should iterate through all of the entries on the appearance list corresponding to the item with the lowest support. For each of the entries (id, r) on this list, the algorithm should verify, if the sequence identified by id is indeed a supersequence of the query sequence. Let us consider a situation, in which the item with the lowest support appears a number times in a single sequence. In such a case, the main loop would have to verify several times the same sequence, whether it is a supersequence of the query sequence. Such behaviour is necessary, because the algorithm which verifies each sequence (described in the next paragraph) will detect the supersequence of the query sequence only when the position r from the analysed appearance list entry refers to the element from the database sequence, which is a superset of the query sequence element at position p. However, if the verification algorithm detects that the analysed database sequence is already in the result set of the query, then the consecutive entries referring to the same sequence may be omitted.

In the previous discussion we omitted the problems associated with the order of the items in the query sequence. We shall address them now. Consider a situation in which the item x from the query sequence, which has the lowest support, is in the element s_p 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'_r such that $s_p \subseteq s'_r$. It is easy to notice that there must exist such s'_r that $r \geq p$. Therefore, during processing of the appearance list of the item with the lowest support we should only verify entries such that $r \geq p$.

Let us now consider the sequence verification algorithm. As was stated briefly before, this algorithm checks if there are entries on all of the consecutive appearance lists, which refer to the verified sequence. To work correctly, this algorithm must take into account the information about order of the elements, as well as the fact that the item may appear in the sequence more than once. Therefore, this algorithm should try to assign all of the items from the query sequence to some entries on the appearance lists, referring to the verified sequence in such a way that the order of the items in the query sequence and the items corresponding to the appearance list entries in the database sequence are the same. To achieve this we suggest the following, recursive, appearance list processing schema.

Let id' be the identifier of the sequence which is to be verified. For each entry $(id', r2)$ on the appearance list $\mathcal{L}^{x2}$ such that $(x2, p2)_2 \in \mathcal{Q}^T$ and $r2$ does not disturb the order of the items, find entries $(id', r3)$ on the appearance list $\mathcal{L}^{x3}$ such that $(x3, p3)_3 \in \mathcal{Q}^T$ and $r3$ does not disturb the order of the items. For each of the entries found on the appearance list $\mathcal{L}^{x3}$ find the entries $(id', r4)$ on the appearance list $\mathcal{L}^{x4}$ such that $(x4, p4)_4 \in \mathcal{Q}^T$ and $r4$ does not disturb the order of the items. Perform such search until you find an entry for all of the pairs in $\mathcal{Q}^T$. If the entry for the last pair of the sequence $\mathcal{Q}^T$ is found, then the sequence id' should be included in the result set and further verification of this sequence may be aborted. In order to be able to find such r values in the appearance list entries, which respect the ordering of the items, we propose the following solution. We allocate an auxiliary table called MAP of the size equal to $|\mathcal{Q}|$. We use this series to "map" the query sequence elements to database sequence elements. The value r stored at the index number p in the table MAP means that we have mapped the pth query sequence element to the rth element of the database sequence. If $MAP[p] = null$ then the query sequence element at the position p is not mapped. Initially, this series is filled with $null$s. Given a pair (x, p) of the converted query sequence $\mathcal{Q}^T$ and an identifier id of the verified sequence, if we find an entry (id, r) on the appearance list $\mathcal{L}^x$, we map the query sequence element at position p to the rth database sequence element, by storing the value r in $MAP[p]$. The auxiliary table MAP may be utilized in two ways. First, if during recursive scanning of the appearance list we start processing of the pair (x, p) from the sequence $\mathcal{Q}^T$ such that $MAP[p] \neq null$ we know that the only entry on the appearance list $\mathcal{L}^x$, which does not disturb the order of the items, is the entry $(id', MAP[p])$. Second, if during recursive processing of the appearance list we start processing of the pair (x, p) from the sequence $\mathcal{Q}^T$, such that $MAP[p] = null$, we may find the lower and upper bound on the position r, which does not disturb the order of the items. Let us start with the calculation of the lower bound. Two cases, which must be addressed during the calculation of this bound are presented by Figs. 1 and 2. The first case (Fig. 1) represents a situation, in which there are no mappings of the query sequence elements to the database sequence elements for query elements, which have a lower position than the currently analyzed element. In such a case the database sequence element should have the position at least equal to the position of the query element (this is the same situation as in the main loop of the set subsequence query algorithm). The second case (Fig. 2) represents a situation, in which there are mappings of the query sequence elements to the database sequence elements for query elements which have a lower position then the currently analyzed element. In such a case, we must find among these mappings, the one that corresponds to the query element that is the nearest to the analysed query element. The lower bound is equal to the value stored in the found mapping plus the number of elements located between the two analysed query sequence elements. Steps for calculating the lower bound are given by the algorithm 3. The algorithm for calculating the upper bound is very similar. As before, there are two cases,

which are illustrated by Figs. 3 and 4. In the first case (Fig. 3), there are no mappings for any of the query sequence elements, which have position greater than the analysed element. In such a case, there is no upper bound, so the algorithm should return ∞. In the second case (Fig. 4) there are mappings for some of query sequence elements, which have a position greater than the analysed query sequence element. In such a case we must find among these mappings, the one that corresponds to the query element that is the nearest to the analysed query element. The upper bound is equal to the value stored in the found mapping minus the number of elements located between the two analysed query sequence elements. Steps for calculating the upper bound are given by the algorithm 4.

Concluding, the data stored in the table MAP enables the sequence verifying algorithm to respect the order of the items in the sequence as it allows for determining which elements of the database sequence may be mapped to the given query sequence element. The above discussion is summarized by the algorithms 1 and 2.

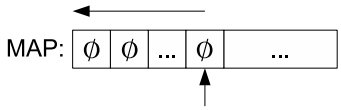

Figure 1. Lower bound, case 1

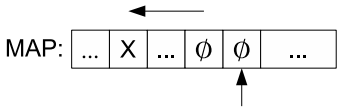

Figure 2. Lower bound, case 2

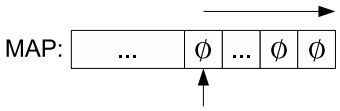

Figure 3. Upper bound, case 1

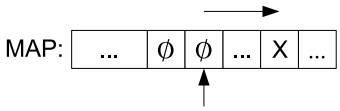

Figure 4. Upper bound, case 2

This concludes the description of the recursive algorithm for set subsequence queries. We shall now demonstrate an exemplary execution of the presented algorithm using an exemplary database and index (see Table 1).

EXAMPLE 1 *Let us consider the following query sequence:*

$$\mathcal{Q} = \langle \{1,3\}, \{4\} \rangle .$$

The first step is to convert this query sequence to a sequence of pairs of item identifier and its position in the query sequence, ordered by the support of the items. After conversion, we obtain the following sequence:

$$\mathcal{Q}^T = \langle (4,2)(1,1)(3,1) \rangle .$$

Next, we retrieve the first (and, in this case, only) entry from the appearance list of the item 4*. This entry refers to the fourth element of the second database sequence. Because item* 4 *comes from the second element from the query sequence,*

Algorithm 1 An algorithm for set subsequence queries

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

1. Allocate auxiliary table called MAP of size equal to $|\mathcal{Q}|$ and fill it with NULLs.
2. $lastId \leftarrow -1$
3. Transform the query sequence $\mathcal{Q}$ into the sequence of pairs $\langle x, p\rangle_i$, denoted $\mathcal{Q}^T$, 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 and element positions.
4. For each of the entries (id, r) on the appearance list of the item x from the pair $\langle x, p\rangle_1$ (corresponding to the item with the lowest support), such that $r >= p$ and $id > lastId$, perform the following steps:
   (a) MAP[p] $\leftarrow r$
   (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[r] $\leftarrow null$

Algorithm 2 Function checkSub used by the algorithm 1, which verifies whether 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 auxiliary 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 $\langle x, p\rangle_{level}$ from $\mathcal{Q}^T$.
3. If MAP[p] $\neq NULL$ then perform the following steps:
   (a) Check on the appearance list of the item x if it contains the entry $\{id,$ MAP[p]$\}$.
   (b) If it does not, return false.
   (c) If it does, return the value returned by the function call: $checkSub(id, level + 1)$.
4. If MAP[p] $= NULL$ then perform the following steps:
   (a) $l \leftarrow lowerBound(p)$ (algorithm 3)
   (b) $u \leftarrow upperBound(p)$ (algorithm 4)
   (c) For each of the entries (id, r) on the appearance list of the item x, such that $r \geq l \wedge r \leq u$ perform the following steps:
      i. MAP[p] $\leftarrow r$
      ii. If the value returned by the call to the function $checkSub(id, level + 1)$ is TRUE then return TRUE.
      iii. MAP[p] $\leftarrow NULL$
   (d) Return FALSE.

we map the second element from the query sequence to the fourth element of the database sequence by storing 4 under the second entry of the MAP table. Now, we need to analyze item 1 from the query sequence, which comes from the first element of the query sequence. By analysing the MAP table (algorithms 3 and 4) we obtain lower and upper bound on the position of the database element corresponding to the first element of the query sequence. The obtained lower bound

Algorithm 3 Function *lowerBound*, calculating the smallest possible element mapping for the given element position in the query sequence.

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

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

Algorithm 4 Function *upperBound*, calculating the largest possible element mapping for the given element position in the query sequence.

ASSUMPTIONS: We assume that the auxiliary table MAP is globally accessible.
INPUT: Query sequence element number p.
OUTPUT: The greatest possible element position in the database sequence, which may be analysed as a potential superset of the element p.

1. Find the smallest index in the table MAP, which is larger than p, 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 $MAP[i] + p - i$.

is equal to 1*, and the upper bound is equal to* 3 *(see Fig. 5(a)). Because we analyse the second database sequence, we now search the appearance list of item* 1 *for entries referring to first, second or third element of the database sequence. There are two such entries:* $(2, 1)$ *and* $(2, 2)$ *(see Fig. 5(b)). We start processing these entries with the entry* $(2, 1)$*. This entry refers to the first element of the database sequence, which means that we need to map the first element of the query sequence to the first element of the database sequence. We assign* 1 *to the first entry of the* MAP *table. We now process the third item from the query sequence. This item comes from the first element of the query sequence. Because we analyse the second database sequence, and the first element of the query sequence is mapped to the first element of the database sequence, we search for the entry* $(2, 1)$ *on the appearance list of the item* 3*. There is no such entry, therefore, we now need to retract to analysis of the second item of the query sequence (see Fig. 5(c)). We now choose the second entry retrieved from the appearance list of the item* 1 *(entry* $(2, 2)$*). We map the first element of the query sequence to the second element of the database sequence by assigning value* 2 *to the first entry of the* MAP *table and analyse the appearance list of item* 3*. Because we analyse the second database sequence and the first element of the query sequence is mapped to the second element of the database sequence, we search for the entry* $(2, 2)$ *on the appearance list of the item* 3*. There is such an entry (see Fig. 5(d)). Because entries on appearance lists have been found for all of the*

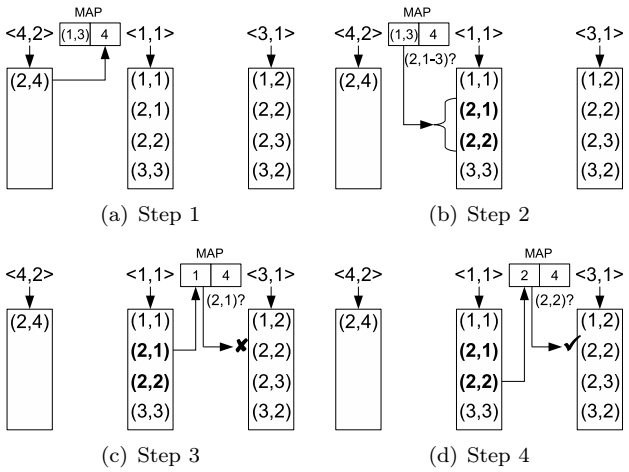

Figure 5. Illustration for Example 1

items from the query sequence, the second database sequence identifier should be stored in the result set. Because there are no more entries on the appearance list of the item 4*, we terminate the algorithm. The result set stores only identifier of the second database sequence.*

Though algorithm 1 offers very good performance (as it will be shown in Section 5) there are some special cases, in which it could perform poorly. Let us consider a database composed only of the following database sequence: $\mathcal{S}^1 = \langle\{1,2,3\},\{1,2,3\},\{1,2,3\},\{1,2,4\}\rangle$ and a query sequence $\mathcal{Q} = \langle\{1,2,4\}\rangle$. The index for such a database is shown in Table 2.

Table 2. FIRE index for a database composed only of the sequence $\mathcal{S}^1 = \langle\{1,2,3\},\{1,2,3\},\{1,2,3\},\{1,2,4\}\rangle$

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

Assume that after the transformation of the query sequence, we obtain the following sequence of pairs: $\langle(1,1)(2,1)(4,1)\rangle$. Processing of this query will start with the analysis of the appearance list of item 1. First, the only element of the query sequence will be mapped to the first element of the database sequence.

Next, we process item 2. Because the element of the query sequence has been mapped we only need to check if the entry $(1, 1)$ exists on the appearance list of item 2. Because such entry exists we now need to check if the same entry exists on the appearance list of item 4. Unfortunately, there is no such entry. Because of this, we will need to retract to the first step of the algorithm, and choose the second entry from the appearance list of item 1. Obviously, once again we will need to retract for this and even the third entry on the appearance list of item 1, each time performing almost complete verification of the database sequence. The database sequence will only be verified for the last entry on this list. To remedy this problem, we propose another, non-recursive, algorithm for set subsequence queries, which is based on the algorithm for solving subset queries using simple inverted file.

Consider an itemset Q. If we read from the index appearance lists of all of the items from the set and find their intersection, we obtain all the supersets of the set Q. We extend this algorithm to include information about the order of the elements in the query sequence. Given the query sequence $\mathcal{Q}$, we find all of the supersets of the element q_1. For the sake of simplicity, let us assume that each sequence in the database contains at most one such superset. Database sequences storing the supersets found in this step form the first approximation of the result set, because such sequences are supersequences of a prefix of the query sequence composed of a single element. Next, we search for all of supersets of the element q_2, but discard all of the elements that do not belong to the sequences which were found in the previous step or their position in the database sequence is smaller than the position of the previously found superset from the same sequence. Once again, assume that each sequence in the database stores at most one superset of the element q_2. Next approximation of the result set is a set of sequences storing supersets found in this step (it is a set of sequences which are supersequences of the prefix of the query sequence composed of two first elements of the query sequence). We repeat the last step for the rest of the elements from the query sequence, obtaining more accurate approximations of the result set. The last approximation is equal to the result set. While this procedure allows us to find all of the supersequences of the query sequence, it will only work properly under the assumption that each sequence in the database stores at most one superset of each of the query sequence elements. If at one of the steps more than one superset in a single database sequence is found, there is a problem of choosing which supersets position should be used as input for the next step of the algorithm. For the algorithm to work properly, we need to chose the superset with the lowest position. Clearly, if any other superset had been chosen, a superset of one of the next elements in the query sequence could be pruned, resulting in a false dismissal of the database sequence which could possibly be a result of the query. The above discussion is summarized by the algorithm 5. This basic algorithm may be optimized thanks to the following observation. Notice that in the loop iterating through the query sequence elements, in each iteration whole appearance lists are read and their intersections found. As can be clearly

seen, we need to read only parts of these lists, as we are only interested in entries referring to the elements in sequences, which were found in the previous step. Furthermore, the interesting entries should have the position greater than the position of the previously found element from the same sequence. Retrieval of such entries requires a more complicated access to the physical index structure which, for big candidate sets, could result in poor performance. However, after several iterations, when the candidate sets are smaller, it may be beneficial for the algorithm to use this optimization. The optimizations discussed above are included in the algorithm 6.

Algorithm 5 Basic non-recursive algorithm for set subsequence queries

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

1. Read appearance lists of all of the items from the element q_1 and find their intersection. The obtained set of entries will be denoted E.
2. $E \leftarrow \{(id, r) : (id, r) \in E \wedge \nexists (id, r') \in E : r' < r\}$
3. For each of the elements q_i, where $i = 2, \ldots, |\mathcal{Q}|$ perform the following steps:
   (a) Read appearance lists of all of the items from the element q_i and find their intersection. The obtained set of entries will be denoted E'.
   (b) $T \leftarrow \{(id, r) : (id, r) \in E' \wedge \exists (id, r') \in E : r' < r\}$
   (c) $E \leftarrow \{(id, r) : (id, r) \in T \wedge \nexists (id, r') \in T : r' < r\}$
4. $results \leftarrow \{id : (id, r) \in E\}$

Algorithm 6 Improved non-recursive algorithm for set subsequence queries

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

1. Read appearance lists of all of the items from the element q_1 and find their intersection. The obtained set of entries will be denoted E.
2. $E \leftarrow \{(id, r) : (id, r) \in E \wedge \nexists (id, r') \in E : r' < r\}$
3. For each of the elements q_i, where $i = 2, \ldots, |\mathcal{Q}|$ perform the following steps:
   (a) If the set E is small enough then
      i. Read appearance lists of all of the items from the element q_i and find their intersection. Retrieve only entries (id, r) such that $\exists (id, r') \in E : r > r'$ The obtained set of entries will be denoted E'.
      ii. $E \leftarrow \{(id, r) : (id, r) \in E' \wedge \nexists (id, r') \in E' : r' < r\}$
   (b) Else
      i. Read appearance lists of all of the items from the element q_i and find their intersection. The obtained set of entries will be denoted E'.
      ii. $T \leftarrow \{(id, r) : (id, r) \in E' \wedge \exists (id, r') \in E : r' < r\}$
      iii. $E \leftarrow \{(id, r) : (id, r) \in T \wedge \nexists (id, r') \in T : r' < r\}$
4. $results \leftarrow \{id : (id, r) \in E\}$

EXAMPLE 2 *Let us now demonstrate an exemplary execution of the non-recursive set subsequence query processing algorithm. Our example will use the algorithm 6. We assume that the first approximation of the result set is "small*

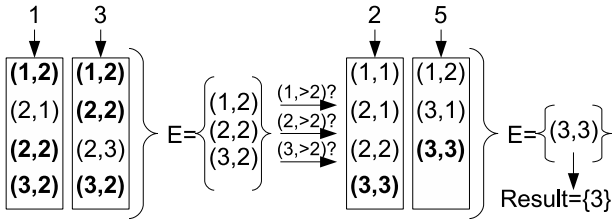

Figure 6. Illustration for Example 2

enough" to use this optimization. Consider the following query sequence:

$$\mathcal{Q} = \langle \{1, 3\}, \{2, 5\} \rangle .$$

As a first step, we retrieve appearance lists of items 1 *and* 3*, and calculate their intersection. The computed intersection consists of entries:* $(1, 2)$*,* $(2, 2)$ *and* $(3, 2)$*. Normally, as a next step, we would remove all entries such that there exists another entry referring to the element of the same sequence, but with lower position, however, in this intersection there are no such entries. Therefore, the set* E *stores all of the obtained entries. Next, we start analysing the second element from the query sequence. Because we use optimization, we iterate through all of the entries stored in set* E *and for each such entry we retrieve from the appearance lists of items* 2 *and* 5 *entries referring to the same sequence, but with higher position. We calculate intersections of the obtained sets of entries. For entries* $(1, 2)$ *and* $(2, 2)$ *from the set* E *the obtained intersections are empty. For entry* $(3, 2)$*, the obtained intersection is composed of a single entry* $(3, 3)$*. Therefore, the new set* E *stores only one entry* $(3, 3)$*. Because we finished analysing all of the elements from the query sequence, we obtain the result set from the last set* E*. The result set stores the identifier of the third sequence in the database. This example is illustrated by Fig. 6.*

The index FIRE is easily updatable. The algorithm for incremental updates of the index is straightforward. To reflect changes in database, just remove entries on the 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 7.

Algorithm 7 An algorithm for incremental updates of the index

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

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

4.3. Physical index structure

We shall now discuss the physical structure of an index. It is easy to notice that the recursive algorithm for query execution reads the index in three different ways: scans the whole appearance list, scans the appearance list entries referring to a single sequence and only a given interval of elements, and reads a single entry on the appearance list (check whether a given entry is on the list, or not). The non-recursive algorithm either scans the whole appearance list or finds (on the appearance list) entries referring to elements from a single sequence, which have a position greater than the given value.

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 are associated with them). Let keys be the triples $\langle x, id, r\rangle$, where x is the item identifier, id is the identifier of a sequence, and r is the position 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 item, then by sequence identifier and finally by the element position. 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 exists only, if there is at least one entry from it stored in the tree).

5. Performance tests

We have performed three different experiments, testing the impact of: the number of sequences, the average sequence length and the average element size on the index performance. For each of the experiments we built 20 databases, 10 of which were built using uniform distribution and the other 10 were built using the zipfian distribution for element generation. For each of the databases we randomly built 40 queries. During experiments these sets of queries were executed 10 times. The 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 3 summarizes the experiment parameters.

The first experiment tested the impact of the number of sequences stored in database on the index performance. Fig. 7 presents the performance of the FIRE index for zipfian and uniform distributions in comparison to the performance of the AISS index and full scan of database. Analysing the Fig. 7 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, for the uniform distribution of the items the query processing times of

Table 3. Experiment parameters

Parameter	Experiment number:		
	1	2	3
number of different items	150000	150000	150000
item distribution	zipfian and uniform		
minimal element size [items]	1	1	5-95
maximal element size [items]	30	30	15-105
minimal sequence size [elements]	1	5-95	5
maximal sequence size [elements]	10	15-105	15
number of sequences	10000-100000	10000	10000
page/node size [bytes]	4096B	4096B	4096B

the recursive algorithm for the FIRE index are smaller than those of the AISS index, while the non-recursive algorithm is comparable with the AISS index algorithm. Third, for the index AISS and the recursive algorithm of the index FIRE, the query processing times do not depend significantly on the distribution of items. However, for the non-recursive algorithm, distribution of items has a great impact on the index performance. For the uniform distribution, the non-recursive algorithm is comparable with AISS index, whereas for skewed distribution, performance of this algorithm is worse by more than two orders of magnitude when compared to other algorithms. Fourth, when we compare query processing times when using index, to times obtained during a full scan of database, we may notice that they are three orders of magnitude smaller.

The second experiment tested the impact of the average length of sequences stored in the database on the index performance. Fig. 8 presents the performance of the FIRE index for zipfian and uniform distributions in comparison to the performance of the AISS index and the full scan of the database. Let us consider the results presented in Fig. 8. Once again we may observe linear dependency of query processing times on the average length of sequences stored in the database. The recursive algorithm of the FIRE index processes queries faster then the AISS index. We may also notice that query processing times (using the recursive algorithm) for databases with the zipfian distribution are a bit smaller than 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 the shortest appearance lists), we obtain smaller sets of sequences to verify in the main loop of the query processing algorithm, which improves the 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 particularly apparent for the average query execu-

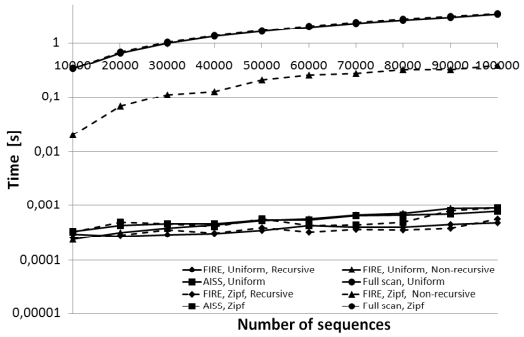

Figure 7. Number of sequences

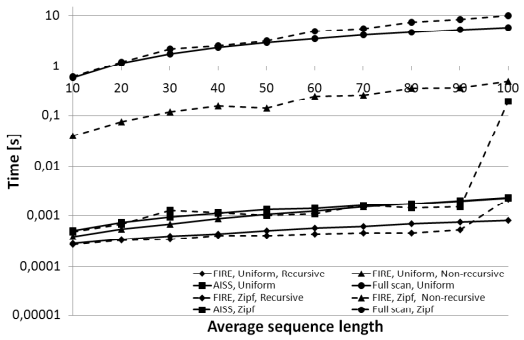

Figure 8. Average sequence length

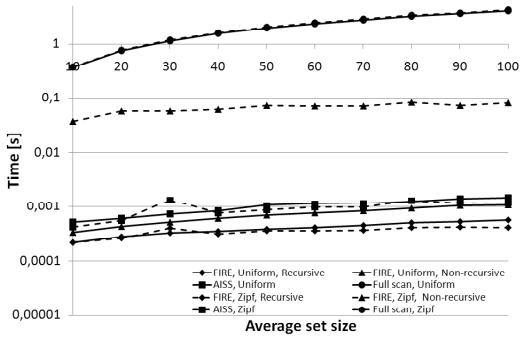

Figure 9. Average element size

tion times on databases with sequences of average length 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 and Morzy (2006a), 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 recursive algorithm of the FIRE index behaves much better in this case. The non-recursive algorithm, as in previous experiment, behaves differently depending on the distribution of the items. For uniform distribution, its performance is comparable with that of the AISS index, whereas for zipfian distribution its performance is by more than two orders of magnitude worse. When we compare query processing times using index to those obtained using full scan of the database, we may notice that they are by more than three orders of magnitude smaller.

The third experiment tested the impact of the average size of elements in the database on the index performance. Fig. 9 presents the performance of the FIRE index for zipfian and uniform distributions in comparison to the performance of the AISS index and full scan of the database. Let us analyse Fig. 9. The dependency of the query execution times on the average size of elements is also linear. As in previous experiments, FIRE index is faster than the AISS index, for the recursive algorithm query processing times are a little bit shorter for the databases with the zipfian distribution of items, whereas for the non-recursive algorithm query processing times on databases with the zipfian distribution are by more than an order of magnitude worse than for the rest of the algorithms. Finally, when we compare query processing times using index to those observed during a full scan of the database, FIRE index is by three orders of magnitude faster than the full scan of database.

6. Conclusions and future work

We have proposed a new indexing scheme, capable of retrieving 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 does not need any assumptions as to the physical and logical structure of the database. As we have experimentally shown, the FIRE index is faster than the AISS index and processes set subsequence queries by three orders of magnitude faster than the full scan of database. Query processing times are also almost independent of the distribution of items (they may be even shorter when the items have skewed distribution).

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

References

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*, Chicago. Springer Verlag, 69–84.

Andrzejewski, W., Gaertig, P., Radom, M. and Antoniewicz, M. (2003) Opracowanie i analiza wydajnościowa indeksu dla przybliżonego wyszukiwania podzbiorów danych (Development and efficiency analysis of an index for the approximate retrieval of data subsets; in Polish). Diploma Thesis. Poznan University of Technology.

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

Andrzejewski, W. and Morzy, T. (2006b) 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*, Istanbul. Springer Verlag, 864–873.

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*, Pisa. ACM Press, 77–87.

Faloutsos, C. and 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*, Minneapolis. ACM Press, 419–429.

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*, Budapest. IEEE, 47–54.

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.

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, Santa Barbara. ACM Press, 151–162.

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*, Seattle. ACM Press, 206–215.

LEVENSHTEIN, V.I. (1965) Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii 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**, Springer Verlag, 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*, Philadelphia. Society for Industrial and Applied Mathematics, 319–327.

MCCREIGHT, E.M. (1976) A space-economical suffix tree construction algorithm. *J. ACM* **23** (2), 262–272.

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*, Virginia. ACM Press, 63–68.

UKKONEN, E. (1992) Constructing suffix trees on-line in linear time. *Information Processing 92, Proceedings of IFIP 12th World Computer Congress, volume 1*. Elsevier Sci. Publ., 484–492.

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*, Washington. ACM Press, 216–225.

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*, Bangalore. IEEE Computer Society, 63–74.

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

YI, B.-K. and FALOUTSOS, C. (2000) Fast time sequence indexing for arbitrary L_p norms. *Proceedings of the 26th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 385–394.