Information and Software Technology 45 (2003) 23–34

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Efficient storage and querying of sequential patterns in database systems[☆]

Alexandros Nanopoulos[a], Maciej Zakrzewicz[b], Tadeusz Morzy[b], Yannis Manolopoulos[a,*]

[a]*Department of Informatics, Aristotle University, Thessaloniki, Greece*
[b]*Institute of Computing Science, Poznan University of Technology, Poznan, Poland*

Received 15 April 2002; revised 15 July 2002; accepted 19 August 2002

Abstract

The number of patterns discovered by data mining can become tremendous, in some cases exceeding the size of the original database. Therefore, there is a requirement for querying previously generated mining results or for querying the database against discovered patters. In this paper, we focus on developing methods for the storage and querying of large collections of sequential patterns. We describe a family of algorithms, which address the problem of considering the ordering among elements, that is crucial when dealing with sequential patterns. Moreover, we take into account the fact that the distribution of elements within sequential patterns is highly skewed, to propose a novel approach for the effective encoding of patterns. Experimental results, which examine a variety of factors, illustrate the efficiency of the proposed method.

Keywords: Pattern post-processing; Sequential patterns; Persistency signatures

1. Introduction

Mining from large databases (also referred to as *database mining*) sets new challenges and opportunities to database technology itself [15]. There is a need for new query languages and query processing methods that will address the requirements posed by database mining. Most of the existing data mining applications, however, assume a loose coupling between the data-mining environment and the database [1]. The narrowing of the 'gap' between data mining and databases refers to the problem of developing data mining algorithms that will present a tighter coupling with the DBMS [1]. This problem has started recently to be confronted by introducing new design specifications of the DBMS (not having to adhere to third NF, reduction of concurrency control and recovering overhead, synergy between OLTP and Data Mining [33]). Moreover, efficient algorithms that exploit the support and achieve a tighter coupling with existing DBMS have been proposed [1] (see [34] for a comparison of several implementations).

Nevertheless, it is important to observe that a major obstacle in the wide spread use of data mining technology is not only insufficient performance, but also the absence of a paradigm for the robust development of data-mining applications and their integration with the DBMS [15]. Along the lines of the latter observation, Imieliński and Mannila describe a long term paradigm, called KDDMS (Knowledge and Data Discovery Management System[1]), which is based on developing KDD query languages (see Section 1.1 for a more detailed description), building optimizing compilers for ad hoc mining queries and application programming interfaces (APIs). Although several KDD query languages have been proposed (e.g. Mine-Rule [20], MSQL [16], DMQL of DBMiner [11,23]), few methods have been proposed in the other directions; for instance, OLE DB for Data Mining [29], the Discovery Board system [37], or the system proposed in Ref. [21].

1.1. Motivation

A KDD query language allows for the expression of KDD queries (i.e. a query in a KDDMS), which are predicate resulting into a set of KDD objects (e.g.

☆ Work supported by a Bilateral Greek – Polish Program.

* Corresponding author.

E-mail addresses: alex@delab.csd.auth.gr (A. Nanopoulos), maciej.zakrzewicz@cs.put.poznan.pl (M. Zakrzewicz), tadeusz.morzy@cs.put.poznan.pl (T. Morzy), manolopo@delab.csd.auth.gr (Y. Manolopoulos).

[1] The term KDDMS is introduced in Ref. [15], and the KDD prefix in this term means Knowledge and Data Discovery, which is different from the broad use of the term KDD as Knowledge Discovery and Data mining.

classification/association rules, discovered clusters) and database objects (e.g. tuples) [16]. Existing KDD query languages are usually SQL-like that contains extensions to handle KDD objects. A KDD object may not exist or may be *pre-generated* and stored in a database, like a rule-base that stores discovered association rules [17]. Since the number of discovered patterns can be extremely large, KDD querying should facilitate both the above two cases: the selective generation of patterns (i.e. the discovery of patterns with user-defined constraints) and the management of previously generated results (i.e. the handling of already mined patterns). Regarding the former case, techniques for pushing constraints down to the mining process have been proposed, mainly for association rules [30].

However, the latter case also presents a main challenge, since there is an urge need to deal with the large numbers of discovered patterns. Pattern post-processing is about the selection of discovered patterns according to user-defined criteria, the further 'mine-around' them, or the querying of the database against patterns to identify transactions that satisfy certain criteria. These operations are included in existing KDD query languages; e.g. the SelectRules or the Typical/Atypical primitives in MSQL [16]. However, for the actual implementation, little support is provided by existing DBMS or mining tools for the persistency of the discovered patterns. There is a need for persistency of patterns because, without being able to efficiently manage the possible large volumes of generated patterns (that may be the accumulated outcome of several mining sessions), the user/analyst has to confront an overwhelming situation, which does not advocate the KDD process.[2] Therefore, what is required is the ability for the persistent storage, which will help towards a more systematic development of data mining applications over DBMSs.

1.2. Contribution and paper organization

In this paper, we are concerned with the development of methods for the storage and querying of large collections of discovered sequential patterns [2]. We focus on the *pattern query*, that is, the finding of all sequential patterns that contain an ordered set of user-defined elements. As described, pattern queries are essential primitives for selective processing of patterns, e.g. for further examination/mining, etc. (see Section 3 for the formal definition of pattern query).

The contributions of this paper are summarized as follows:

- We describe equivalent sets, a method which addresses the problem of taking into account the ordering of items within sequences. Although the consideration of ordering among elements is crucial for sequential patterns (which are ordered sequences of sets of items [2]), existing work has not addressed this problem.
- We recognize that signatures constitute a compact and effective representation of equivalent sets. Therefore, we develop a family of algorithms (called SEQ) within the framework of data structures for signatures, which address the problem of indexing sequential patterns and processing pattern queries.
- We make the observation that the distribution of elements within sequential patterns is highly skewed. Therefore, we propose a novel approach for the effective encoding of equivalent sets, which is incorporated to the one of the algorithms of the SEQ family (SEQ(A)).
- We provide extensive experimental results, examining a variety of factors, which illustrate the performance of the described methods.

[2] It has to be noticed that the management of data mining results is considered in Ref. [8] as a necessary step in the whole KDD process.

The rest of this paper is organized as follows. Section 2 gives an overview of the related work, whereas the scheme for the representation of sequential patterns is described in Section 3. The family of signature-based algorithms is presented in Section 4. Section 5 contains the experimental results, and finally Section 6 concludes the paper.

2. Background and related work

2.1. Problem description

We begin by illustrating the problem of indexing sequential patterns with a small example drawn from the web usage analysis [5,26,32,31]. Let a web access log depicted in Fig. 1. The web log represents the history of user's visits to a web server, done with a web client program. Each web log entry represents a single user's access to a web page and contains the client's *IP* address, the timestamp, the *URL* address of the requested object, and some additional information. Several requests from the same client may have identical timestamps since they may represent accesses to different objects of a single web page. Access requests issued by a client within a single session with a web server constitute a client's access sequence (or simply *sequence*).

Assume that the log of Fig. 1 is stored in the relation $R(IP, TS, URL)$, depicted in Fig. 2a (IP is the client's IP, TS the time-stamp, and URL address of the requested object). A pattern query (see Definition 1 for a formal description) identifies all access sequences, which contain a collection of given addresses with a specified order. For instance, assume that we want to find all access sequences stored in the relation R, which contain A, E, and F in the following order: $\{A\} \rightarrow \{E\} \rightarrow \{F\}$. The *SQL* query, which implements the above defined pattern query is depicted in Fig. 2b.

Pattern queries can be useful for web-log analysis, e.g. for dynamic advertising or web site linkage reorganization. Other applications include user transactions in e-commerce sites, in telecommunications or retail records, and in

```
154.11.231.17  – –  [13/Jul/2000 : 20 : 42  +0200]  “GET  /  HTTP/1.1”  200  1673
154.11.231.17  – –  [13/Jul/2000 : 20 : 42  +0200]  “GET  /apache_pb.gif  HTTP/1.1”  200  2326
154.11.231.17  – –  [13/Jul/2000 : 20 : 43  +0200]  “GET  /demo.html  HTTP/1.1”  200  520
192.168.1.25  – –  [13/Jul/2000 : 20 : 42  +0200]  “GET  /demo.html  HTTP/1.1”  200  520
192.168.1.25  – –  [13/Jul/2000 : 20 : 44  +0200]  “GET  /books.html  HTTP/1.1”  200  3402
160.81.77.20  – –  [13/Jul/2000 : 20 : 42  +0200]  “GET  /  HTTP/1.1”  200  1673
154.11.231.17  – –  [13/Jul/2000 : 20 : 4  +0200]  “GET  /cdisk.html  HTTP/1.1”  200  3856
192.168.1.25  – –  [13/Jul/2000 : 20 : 49  +0200]  “GET  /cdisk.html  HTTP/1.1”  200  3856
154.11.231.17  – –  [13/Jul/2000 : 20 : 51  +0200]  “GET  /books.html  HTTP/1.1”  200  3402
10.111.62.101  – –  [13/Jul/2000 : 20 : 42  +0200]  “GET  /new/demo.html  HTTP/1.1”  200  971
```

$$\Downarrow$$

```
192.168.1.25:  /demo.html → /books.html  → /cdisk.html
```

Fig. 1. An example of a web access log and a web access sequence.

production processes. Therefore, a pattern query can identify all the discovered sequential patterns among user transactions that contain a purchase of a Cassiopeia palmtop followed by purchase of a docking cradle for the palmtop. With the identified sequential patterns that satisfy the query, we can designate a sales campaign for the two items.

Since SQL language does not contain a sequence search statement, to specify this kind of query in SQL, multiple joins or multiple nested subqueries are required, as illustrated in Fig. 2b. This may presents large query response times, especially for large databases. The straightforward solution of using a sequential scan may require considerable I/O for large data collections. This is especially true for transaction data, web access log data, telecommunication data, which are usually large. In general, finding sequences containing a given subsequence in a relational or object-relational database is a complex and time-consuming task. Thus, there is a problem of appropriate optimizing the database access while performing pattern queries, which is the objective of this work.

IP	TS	URL
1	1	O
1	1	B
1	2	A
1	3	F
1	4	E
2	1	A
2	2	E
2	3	F
3	1	O
4	2	N

(a)

```
select IP
from R a, R b, R c
where a.IP = b.IP
   and b.IP = c.IP
   and a.TS < b.TS
   and B.TS < c.TS
   and a.URL = ’A’
   and b.URL = ’E’
   and c.URL = ’F’;
```

(b)

Fig. 2. (a) The relation R of web access sequences (b) example of pattern query.

2.2. Related work

Traditional database accessing methods (B^+-Tree, bitmap index, etc.) are record-oriented, i.e. they are used to optimize queries based on exact matches of single records. Therefore, they are inappropriate for optimizing pattern queries, which deal with partial matching of multi-record sequences. Another related area concerns the problems of indexing for exact and approximate string searching, which have received considerable attention [27]. Some examples of indexing methods developed for string matching are suffix trees [9], suffix array [19], Q-grams [10], and Q-samples [28]. Also related is the work on episode matching, with which only insertions in the text are permitted [6,36]. However, the unordered nature of sequence elements and the freedom to represent their order in the sequence makes the techniques developed for string and episode matching inappropriate for optimizing pattern queries.

There is one index structure, called set-based index, that is eligible to optimize pattern queries. Set-based index structures have been developed to support queries with set-valued predicates (the so-called *subset, superset and set equality, queries*) [13,24]. Given a finite set of items I (i.e. I is the domain of items), a transaction T is a set of items which is a subset of I. A database D is a set of transactions. Typical example of a subset query Q is to retrieve all transactions from D that contain a given query set Q. A superset query Q is to retrieve all transactions from D those items are contained in a given query set Q. A set equality query Q is to retrieve all those transactions from D that contain exactly a given query set. Finally, a similarity query Q is to retrieve all transactions from D which are most ‘similar’ to Q for suitably defined notion of similarity between sets. These kinds of queries appear in text retrieval systems [3], keyword-based search in databases [14] or in object-oriented database systems [18]. There are only few proposals for set-based indexing [7,13,18,24]. Helmer and Moerkotte [13] adopted traditional techniques, like sequential signature files, signature trees, extensible signature hashing and inverted files, for indexing set-valued attributes (i.e. attributes that are sets of items). It has been observed [13] that the inverted file index structure dominated other index structures for subset and superset queries in terms of query processing time. The problem of applying signature files to retrieving a given set in a large collections of sets was also analyzed by Kitagawa, Ishikawa, and Obho [18]. In Ref. [24] a set-based bitmap index is presented, which facilitates the fast subset searching in relational databases. The index is based on the creation of *group bitmap keys*, which are a special case of superimposed coding via hashing of transactions' contents. The superimposition of group bitmap keys results from the logical OR of all hashed transactions' contents. This method requires a post-processing step to remove false-drops. A false-drops is a set the group bitmap key of which satisfies the query but the set itself does not, and it incurs due to superimposition. Experimental results in Ref. [24] showed that the proposed hash group bitmap index significantly outperforms traditional index structures including B$^+$-trees and bitmap index for subset queries. Other solutions to similarity search problem for text retrieval was proposed in Ref. [7].

All the aforementioned set-based indexing approaches do not consider the ordering of items within the searched query set, which is crucial in storing and querying sequential data. To realize the shortcomings of the set-based indexes let us take the initial example. A set-based index may be applied to support the pattern query from Fig. 2b in the following way. The relation R is transformed into the relation R_{set}, depicted in Fig. 3, and a set-based index is created on the attribute *URL*. Then, using the index, all records from R_{set} containing the searched query item set are retrieved (the order of items in the item set is omitted). Additional post-processing verifying step is necessary to eliminate sequences having incorrect ordering of item sets. The verifying step may cause the significant overhead related to reading and verifying a large number of false-drops from the database (e.g. the sequence 1).

R_{set}

IP	URL
1	O, B, A, F, E
2	A, E, F
3	O
4	N

Fig. 3. The relation R_{set} of sets of web accesses.

Summarizing, the problem of evaluating queries with 'sequence-valued' predicates has been neglected by database community. To the best of our knowledge there is only one approach dealing with indexing of sequential patterns [22,38]. More details are given in Section 4.2.

3. Representation of sequential patterns

Let I be a set of items. An *itemset* is an unordered set of items from I (we follow the notation of Ref. [2] and we call a set of items as itemset). A sequential pattern P is defined as an ordered list of *itemsets* [2]. Thus, $P = \langle X_1, \ldots, X_n \rangle$, where each X_i itemset is called element of P. A sequential pattern $Q = \langle Y_1, \ldots, Y_m \rangle$ $(1 \leq m \leq n)$ is contained by P (we note $Q \preceq P$), if there exist a sequence of m integers $j_1 < \cdots < j_m$ $(1 \leq j_i \leq n)$ for which $Y_1 \subseteq X_{j_1}, \ldots, Y_m \subseteq X_{j_m}$. Therefore, ordering is considered between items of different itemsets, but not between items of the same itemset.

Definition 1. (Pattern Query) Given a database D of sequential patterns and a query sequential pattern Q, a pattern query finds all members P of D for which it holds that $Q \preceq P$ (Q is contained by P).

3.1. Equivalent sets

We assume the existence of an *item-mapping* function $f(i)$ that maps each $i \in I$ to an integer value (since I may contain any type of literals). Henceforth we assume that literals are mapped to consecutive positive integers starting from 1, although any other mapping can be followed. For instance, for $I = \{A, B, C, D, E\}$ we have $f(A) = 1, f(B) = 2, f(C) = 3, f(D) = 4$ and $f(E) = 5$.

We also consider an *order mapping* function $f_0(x, y)$ that transforms a sequential pattern of the form $\langle \{x\}, \{y\} \rangle$ $(x, y \in I)$ to an integer value. For instance, for $f_0(x, y) = 6 \cdot f(x) + f(y)$, we have $f_0(A, B) = 8$. It has to be noticed that the intuition for the use of $f_0(x, y)$ is that it takes into account the ordering, i.e. $f_0(x, y) \neq f_0(x, y)$. This is the reason why we use the weighing with the integer (6 in the previous example) which is by one larger than the largest $f(x)$ value (i.e. this way $f_0(x, y) \neq f_0(y, x)$ and $f_0(x, y)$ values are always larger than $f(x)$ values).

Finally, we denote that in a sequential pattern $P = \langle X_1, \ldots, X_n \rangle$, $x < y$ $(x, y \in I)$, if $x \in X_i$, $y \in X_j$ and $i < j$. Based on the above, we give the definition for the *equivalent set* of a sequential pattern [22].

Definition 2. (Equivalent Set) Given a sequential pattern $P = \langle X_1, \ldots, X_n \rangle$, the equivalent set E of P is defined as:

$$E = \left(\bigcup_{x \in X_1, \ldots, X_n} \{f(x)\} \right) \cup \left(\bigcup_{x, y \in X_1, \ldots, X_n, x < y} \{f_0(x, y)\} \right)$$

For instance, let $P = \langle\{A,B\},\{C\},\{D\}\rangle$ be a sequential pattern. Using the mapping functions that were described above, we get:

$$\begin{aligned} E &= \{f(A), f(B), f(C), f(D)\} \\ &\quad \cup \{f_0(A,C), f_0(B,C), f_0(A,D), f_0(B,D), f_0(C,D)\} \\ &= \{1,2,3,4\} \cup \{9,15,10,16,22\} \\ &= \{1,2,3,4,9,15,10,16,22\} \end{aligned}$$

According to Definition 2, an equivalent set is the union of two sets: the one resulting by considering each element separately and the one from considering pairs of elements. $S(E)$ denotes the former set, consisting of $f(x)$ (i.e. single) elements, and $P(E)$ the latter set, consisting of $f_0(x,y)$ (i.e. pairwise) elements. Based on Definition 2, it is easy to show the following.

Corollary 3. *Let two sequential patterns Q,P and the corresponding equivalent sets E_Q and E_P. If Q is contained by P, then $E_Q \subseteq E_P$.*

Therefore, equivalent sets allow us to express a pattern query problem as the problem of finding all sets of items that contain a given subset (note that Corollary 3 is not reversible in general). Also, it is easy to see that if $E_Q \subseteq E_P$, then $S(E_Q) \subseteq E_P)$ and $P(E_Q) \subseteq (E_P)$.

Evidently, the length $|E|$ of an equivalent set E of a pattern P depends on the lengths of P's elements. Due to the consideration of ordering (the examination of f_0 function for every pair of items between different itemsets) the length of E grows rapidly.

Lemma 4. *Let n be a given number of items and $P = \langle X_1,\dots,X_m\rangle$ a sequential pattern, where $|X_1 \cup \cdots \cup X_m| = n$. The equivalent set $E_{\max}$ with the maximum length is produced in the case that each X_i is a singleton, i.e. $|X_i| = 1$. In this case it holds that $m = n$ and $|E_{\max}| = n + (n/2)$.*

Proof. We will use induction on the number of items n. For $n = 2$ the truth trivially holds. We assume that the truth holds for n. We will prove for $n+1$.

Let $i_1,\dots,i_n$ be the n items for which (due to induction) we assume that the maximum equivalent set is produced when $P = \langle\{i_1\},\{i_2\},\dots,\{i_n\}\rangle$. If i_{n+1} is the new item, it can be added in P either by being inserted to one of the n singleton elements ($\{i_k\}, 1 \le k \le n$) of P, or by becoming a new singleton element[3]. In the former case (without harm of the generality) assume that i_{n+1} is inserted in. The length of the corresponding equivalent set will be equal to $(n+1) + (n/2) + (n-1) = 2\cdot n + (n/2)$. In the latter case, we will have $P = \langle\{i_1\},\{i_2\},\dots,\{i_n\},\{i_{n+1}\}\rangle$, hence the length of the equivalent set will be $n + 1 + ((n+1)/2)$. It is easy to show for $n > 2$ that $n + 1 + ((n+1)/2) > 2\cdot n + (n/2)$.

Conclusively, the maximum length of the equivalent set is produced when each element of P is a singleton. The value of the length in this case is equal to $n + (n/2)$. □

[3] Recall that the elements of P are unordered sets.

3.2. Signatures

Since the size of equivalent sets increases rapidly with increasing pattern length (Lemma 4, they can be represented more efficiently by using *superimposed signatures*. A signature is a bitstring of F bits (denoted as signature *length*) and is used to indicate the presence of elements in a set. Each element of a set can be encoded, by using a hash function, into a signature that has exactly m out F bits equal to '1' and all other bits equal to '0'. The value of m is called the *weight* of the signature. The signature of the whole set is defined as the result of the superimposition of all element signatures (i.e. each bit in the signature of the set is the logical OR operation of the corresponding bits of all its elements). Given two equivalent sets E_1, E_2 and their signatures $S(E_1)$, $S(E_2)$, it holds that $E_1 \subseteq E_2 \Rightarrow S(E_1)$ AND $S(E_2) = S(E_1)$.

Signatures provide a quick filter for testing the subset relationship between sets. Therefore, if there exist any bits of $S(E_1)$ that are equal to '1' and the corresponding bits of $S(E_2)$ are not also equal to '1', then E_1 is not a subset of E_2. The inverse of the latter statement, however, does not hold in general and, evidently, *false-drops* may result from collisions due to the superimposition. To verify a drop (i.e. to determine if it is a true- or a false-drop), we have to examine the corresponding sequences with the containment criterion. In order to minimize the false drops, it has been proved [4] that, for sets of length T, the length of the signatures has to be equal to:

$$F = m\cdot T/\ln 2 \tag{1}$$

Henceforth, based on the approach of Ref. [12] for the case of set-valued object databases, we assume that m is equal to one. Given a collection of sequential patterns, in Section 4 we examine effective methods for organizing the representations of the patterns, which consist of signatures of equivalent sets.

4. Family of SEQ algorithms

4.1. A Simple sequential algorithm

Let D be the database of sequential patterns to be indexed. A simple data structure for indexing elements of D is based on the paradigm of signature-file [4], and is called *SEQ(C)* ("SEQ" denotes that the structure is sequential, and "C" that it uses a complete signature representation for the equivalent set). It corresponds to the direct (i.e. naive) use of signatures of equivalent sets, and is given for comparison purposes. The construction algorithm for SEQ(C) is given in Fig. 4a ($S(E)$ is the signature of equivalent set E).

The algorithm for querying the structure for a given sequential pattern q, is given in Fig. 4b. Initially (step 1), the equivalent set, E_q, of q is calculated. Then, each signature in the structure is examined against the signature $S(E_q)$ (step 4, where *and* denotes the bitwise and of the signatures). The verification of each drop is applied at steps 5–7. The result, consisting of the sequential patterns from D that satisfy query q, is returned in set R.

The cost of the searching algorithm can be decomposed as follows:

(1) *Index scan cost (I/O)*: to read the signatures from the sequential structure.
(2) *Signature test cost (CPU)*: to perform the signature filter test.
(3) *Data scan cost (I/O)*: to read patterns in case of drops.
(4) *Verification cost (CPU)*: to perform the verification of drops.

The signature test is performed very fast, thus the corresponding cost can be neglected. Since the drop verification involves a main memory operation, it is much smaller compared to the Index and Data Scan costs that involve I/O. Therefore the latter two costs determine the cost of the searching algorithm. Moreover, it is a common method to evaluate indexing algorithms by comparing the number of disk accesses, e.g. [4,12,35].

For SEQ(C), the calculation of F (signature length) with Eq. (1) is done using the expected length, $|\bar{E}|$, of equivalent sets (in place of T). Since $|\bar{E}|$ grows rapidly, F can take large values, which increase the possibility of collisions during the generation of signatures (i.e. elements that are hashed in the same position within signatures). Collisions result to false-drops, due to the ambiguity that is introduced (i.e. we cannot determine which of the elements, that may collide in the same positions of the signature, are actually present). Thus, a large Data Scan cost for the verification step incurs. Moreover, large sizes of equivalent sets increase the Index Scan cost (because they result into larger F values, thus to an increase in the size of the index).

Due to the drawbacks described above, in the following we consider the SEQ(C) method as a base to develop more effective methods. Their main characteristic is that they do not handle at the same time the complete equivalent set (i.e. all its elements) for the generation of signatures, so as to avoid the described deficiencies of SEQ(C).

4.2. Partitioning of equivalent sets

In Refs. [38,22] a partitioning technique is proposed that divides equivalent sets into a collection of smaller subsets. With this method, large equivalent sets are partitioned into smaller ones. Thus, in the resulting signatures we will have reduced collision probability, fewer false-drops and reduced Data Scan cost.

Definition 5. (Partitioning of equivalent sets) Given a user-defined value β, the equivalent set E of a sequential pattern P is partitioned into a collection of $E_1,\ldots, E_k$ subsets by:

- dividing P into $P_1,\ldots, P_k$ subsequences, such that $\bigcup_{i=1}^{k} P_i = P$, $P_i \cap P_j = \emptyset$ for $i \neq j$, and
- having E_i be the equivalent set of P_i, where $|E_i| < \beta, 1 \leq i \leq k$.

According to Definition 5, we start from the first element of P being the first element of P_1. Then, we include the following elements from P in P_1, while the equivalent set of P_1 has length smaller than β. When this condition does not hold, we start a new subsequence, P_2. We continue the same process, until all the elements of P have been examined. For instance, let $P = \langle\{A,B\},\{C\},\{D\},\{A,F\},\{B\},\{E\}\rangle$. For β equal to 10, we have: $P_1 = \langle\{A,B\},\{C\},\{D\}\rangle$ and $P_2 = \langle\{A,F\},\{B\},\{E\}\rangle$, because in this case $|E(P_1)| = 9$ and $|E(P_2)| = 9$. Notice that the equivalent set of E has length

```
1.  $F = \emptyset$
2.  forall $p \in D$
3.      $E$ = Equivalent_Set($p$)
4.      $F += \langle S(E), \mathrm{pointer}(p)\rangle$
5.  endfor
```

(a)

```
1.  $E_q$ = Equivalent_Set($q$)
2.  $R = \emptyset$
3.  forall $\langle s, \mathrm{pointer}(p)\rangle \in F$
4.      if $s$ and $S(E_q) = S(E_q)$
5.          Retrieve $p$ from $D$
6.          if $q \preceq p$
7.              $R$ += $p$
8.          endif
9.      endif
10. endfor
```

(b)

Fig. 4. SEQ(C) method: (a) construction algorithm. (b) search algorithm.

```
1.  $R = \emptyset$
2.  forall Equivalent Sets $E = \{E_1 \ldots E_k\}$ stored as $\langle S(E_1), \ldots, S(E_k),\ \text{pointer}(p)\rangle$
3.      $startPos = 0$
4.      for $(i = 1;\ i \le k$ and $startPos \le |q|;\ i{+}{+})$
5.          $endPos = startPos$
6.          $contained$ = true
7.          while ($contained ==$ true and $endPos \le |q|$)
8.              $E_q = $ Equivalent Set$(q[startPos,\ endPos])$
9.              if $S(E_q)$ and $S(E_i) == S(E_q)$
10.                 $endPos{+}{+}$
11.             else
12.                 $contained$ = false
13.             endif
14.         endwhile
15.         $startPos = endPos$
16.     endfor
17.     if $startPos > |q|$
18.         Retrieve $p$ from $D$
19.         if $q \preceq p$
20.             $R \mathrel{+}= p$
21.         endif
22.     endiif
23. endfor
```

Fig. 5. SEQ(P) method: search algorithm.

equal to 32, which is much larger than the length of the partitions.

We denote the above method as *SEQ(P)* (*P* stands for partitioning). The construction algorithm for SEQ(P) is analogous to the one of SEQ(C), depicted in Fig. 4a. After step 3, we have to insert:

3a. Partition E into $E_1, \ldots E_k$

and step 4 is modified accordingly:

```
4.  forall $E_i$
4a.     $F + = \langle S(E_i),\ \text{pointer}(p_i)\rangle$
4b. endfor
```

The search algorithm for SEQ(P) is based on the following observation [22]. For each partition of an equivalent set E, a query pattern q can be decomposed in a number of subsequences. Each subsequence is separately examined against the partitions of E. The algorithm is depicted in Fig. 5.

We assume that an equivalent set is stored as a list that contains the signatures of its partitions, along with a pointer to the actual pattern (step 1). At steps 4–16, the query pattern is examined against each partition and the maximum query part than can be matched by the current partition is identified. The part of query q from *startPos* to *endPos* is denoted as $q[startPos, endPos]$. At the end of this loop (step 17), if all query parts have been matched against the partitions of the current equivalent set (this is examined at step 17, by testing the value of *startPos* variable), then the verification step is performed at steps 18–20.

SEQ(P) partitions large equivalent sets in order to reduce their sizes and, consequently, the Data Scan cost (because it reduces the possibility of collisions within the signatures, thus it results into fewer false-drops). However, since a separate signature is required for each partition of an equivalent set, the total size of the stored signatures increases (the length of each signature in this case is determined by Eq. (1), having in mind that the size of each partition of the equivalent set is equal to β (Definition 5)). Thus, the Index Scan cost may be increased[4].

4.3. Using approximations of equivalent sets

In this section we propose a different method for organizing equivalent sets. It is based on the observation that the distribution of elements within sequential patterns is skewed, since the items that correspond to frequent subsequences (called *large* according to the terminology of Ref. [2]) have larger appearance frequency. Therefore, the pairs of elements that are

[4] Using very small values of β and thus very small signature lengths for each partition, so as not to increase Index Scan cost, has the drawback of significantly increasing the false-drops and the Data Scan cost.

```
1.  forall $i \in I$
2.    find $NN(i) = \{i_j \mid i_j \in I, 1 \le j \le k, i_j \ne i, \forall\, l \notin NN(i)\ \ supp_D(i, i_j) \ge supp_D(i, l)\}$
3.  endfor
4.  $F = \emptyset$
5.  forall $p \in D$
6.      $E$ = Equivalent_Set$(p)$
7.      forall $(x, y) \in P(E)$
8.          if $y \notin NN(x)$
9.              remove pair $(x, y)$ from $E$
10.         endif
11.     endfor
12.     $F+ = \langle S(E), \text{pointer}(p)\rangle$
13. endfor
```

Fig. 6. SEQ(A) method: construction algorithm.

considered during the determination of an equivalent set are not equiprobable.

Due to the above, some pairs have much higher co-occurrence probability than others. The length of equivalent sets can be reduced by taking into account only the pairs with high co-occurrence probability. This represents an approximation of equivalent sets, and the resulting method is denoted as SEQ(A) ("A" stands for approximation). The objective of SEQ(A) is the reduction in the lengths of equivalent sets (so as to reduce Data Scan costs) with a reduction in the sizes of the corresponding signatures (so as to reduce the Index Scan costs).

Recall that $P(E)$ denotes the part of the equivalent set E, which consists of the pairwise elements. Also, $\text{supp}_D(x, y)$ denotes the support of pair (x, y) in D (i.e. the normalized frequency of sequence (x, y) [2]), where $x, y \in I$ and the pair $(x, y) \in P(E)$. The construction algorithm for SEQ(A) is depicted in Fig. 6.

The search algorithm of SEQ(A) is analogous to that of SEQ(C). However, step 1 of the algorithm depicted in Fig. 4b has to be modified accordingly (identical approximation has to be followed for the equivalent set of a query pattern):

```
1.   $E_q$ = Equivalent_Set$(q)$
1a.  forall $(x, y) \in P(E_q)$
1b.      if $y \notin NN(x)$
1c.          remove pair $(x, y)$ from $E_q$
1d.      endif
1e.  endfor
```

Lemma 6. *The SEQ(A) algorithm correctly finds all sequences that satisfy a given pattern query.*

Proof. Let a pattern query Q and its equivalent set E_Q. Also let a sequence S for which $Q \preceq S$ (i.e. Q is contained by S) and E_S its equivalent set. As described (see Corollary 3, it holds that $E_Q \subseteq E_S$, $S(E_Q) \subseteq S(E_S)$ and $P(E_Q) \subseteq P(E_S)$. □

In SEQ(A), let us denote E'_Q and E'_S the equivalent sets of Q and S, respectively, under the approximation imposed by this algorithm. From the construction method of SEQ(A) we have that $S(E'_Q) = S(E_Q)$ and $S(E'_S) = S(E_S)$. Therefore, $S(E'_Q) \subseteq S(E'_S)$.

Focusing on the pairwise elements, let an element $\xi \in P(E_S) - P(E'_S)$ (i.e. ξ is excluded from $P(E'_S)$ due to step 9 of SEQ(A). We can have two cases:

(1) If $\xi \in P(E_Q)$, then $\xi \in P(E_Q) - P(E'_Q)$ (i.e. ξ is also excluded from $P(E'_Q)$, due to the construction algorithm of SEQ(A)—see Fig. 6). Therefore, SEQ(A) removes the same elements from $P(E'_Q)$ and $P(E'_S)$. Since $P(E_Q) \subseteq P(E_S)$, by the removal of such ξ elements, we will have $P(E'_Q) \subseteq P(E'_S)$.
(2) If $\xi \notin P(E_Q)$, then the condition $P(E'_Q) \subseteq P(E'_S)$ is not affected, since such elements excluded from $P(E'_S)$ are not present in $P(E_Q)$, and thus in $P(E'_Q)$.

From both the above cases we have $P(E'_Q) \subseteq P(E'_S)$.

Conclusively, $S(E'_Q) \cup P(E'_Q) \subseteq S(E'_S) \cup P(E'_S)$, which gives $E'_Q \subseteq E'_S$. Hence, we have proved that $Q \preceq S \Rightarrow E'_Q \subseteq E'_S$, which guarantees that SEQ(A) will not miss any sequence S that satisfies the given pattern query (this can be easily seen in a way analogous to Corollary 3.

From the above it follows that with the approximation method of SEQ(A) no loss in precision is triggered (evidently there is no reason to measure the precision/recall, since the method is always accurate).[5] On the other hand, SEQ(A) and all other SEQ algorithms are based on signatures. Therefore, they may incur false-drops, i.e. the fetching of sequences for which their signatures satisfy

[5] Evidently, it easy to see that SEQ(C) and SEQ(P) also do not trigger any loss in precision.

the query condition but the actual sequences do not. The number of false-drops directly affects the Data Scan cost, since the fetching of a large number of sequences requires more I/O operations.

The Data Scan cost of SEQ(A) is reduced, compared to SEC(C), due to the fewer false-drops introduced by the drastic reduction in the sizes of equivalent sets. This is examined experimentally in Section 5. It has to be noticed that the selection of the user-defined parameter k, for the calculation of the NN set in algorithm of Fig. 6, has to be done carefully. A small k value will remove almost all pairs from an equivalent set and in this case the Data Scan cost increases (intuitively, if the equivalent set has very few elements, then the corresponding signature will be full of '0', thus the filtering test becomes less effective). In contrast, a large k value will present a similar behavior as the SEQ(C) algorithm, since almost all pairs are considered. The tuning of the k value is examined in Section 5.

Moreover, differently from SEQ(P), the Index Scan cost for SEQ(A) is reduced, because smaller signatures can be used for the equivalent sets (due to their reduced sizes) and no partitioning is required. Therefore, SEQ(A) combines the advantages of both SEQ(P) and SEQ(C).

5. Performance results

This section contains the experimental results on the performance of all methods. For purposes of comparison, we also consider the approach of generating signatures directly from sequential patterns, i.e. not using equivalent sets and thus, ignoring the ordering of elements (this method is denoted as SEQ(U), where U stands for unordered). First, we describe the data generator that was used for the experiments, and then we present the comparison among all methods.

5.1. Data generation

In order to evaluate the performance of the algorithms over a large range of data characteristics, we generated synthetic sets of sequential patterns. Our data generator considers a model analogous to the one described in Ref. [2]. Following the approach of Refs. [38,22] so as to examine the worst case for equivalent sets, according to Lemma 4 we consider sequential patterns with elements being single items (singletons). Our implementation is based on a modified version of the generator developed in Ref. [25], which was used to produce sequential patterns for the case of web-user traversals (see also Ref. [22]). The reason is because they actually consist of sequences of single items.

The generator builds a pool of sequences, each of them being a sequence of pairwise distinct items from a domain I. The length of each such sequence is a random variable that follows Poisson distribution with a given mean value. A new pool sequence keeps a number of items from the previous one, determined by the correlation factor. Since we are interested in the effect of item ordering within sequences, we modified the generator of Ref. [25] so as to perform a random permutation of the common items before inserting them in the new pool sequence. This results into sequences that contain items with different ordering, thus examines the impact of this factor. The rest of each sequence is formed by selecting items from I with uniform distribution. Each sequence in the pool is associated with a weight. This weight corresponds to its selection probability and is a random variable that follows exponential distribution with unit mean (weights are normalized in the sequel, so that the sum of the weights for all paths equals 1). A sequential pattern is created by picking a sequence from the pool, tossing a L-sided weighted coin (L is the pool size), where the weight for a side is the probability of picking the corresponding pool sequence. In each sequential pattern, a number of random items from I (i.e. following uniform distribution) are inserted to simulate the fact that pool sequences are used as seeds and should not be identical to the actual sequential patterns. This number is determined by the length of the sequential pattern, which is a random variable following Poisson distribution with a given mean value denoted as S. The total number of generated sequential patterns is denoted as N. Each result presented in the following is the average from 5 generated data sets, and for each data set we used 100 queries for each case (e.g. query size, number of sequences, etc.).

5.2. Results

First we focus on the tuning of k for SEQ(A). We used data sets with S set to 10, $|I|$ was set to 1000 and N was equal to 50,000. We measured the total number of disk accesses (Index- and Data-Scan cost) with respect to the length of the query patterns. The results for SEQ(A) with respect to k are depicted in Fig. 7, where k is given as a percentage of $|I|$. As shown, for small values of k (less than 5%), SEQ(A) requires a large number of accesses, because very small

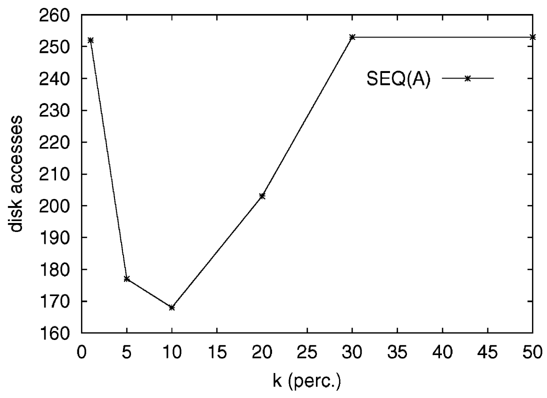

Fig. 7. Tuning of k.

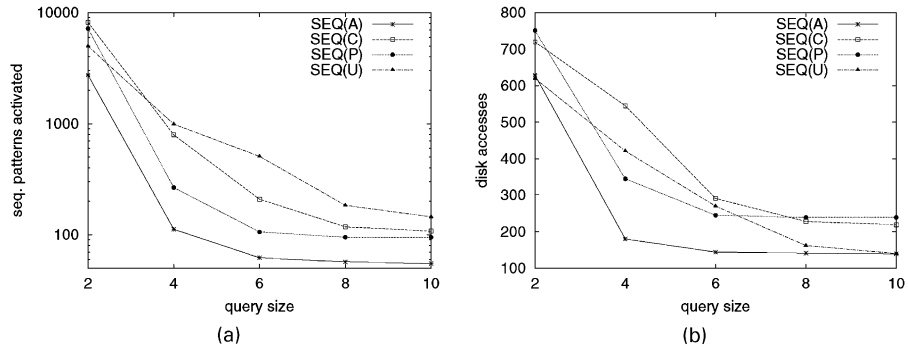

Fig. 8. Comparison of methods: (a) number of activated sequential patterns in the database w.r.t. query size. (b) Disk accesses w.r.t. query size.

equivalent sets are produced that give signatures with almost all bits equal to ‘0’. Thus, as explained in Section 4.3, the filtering of SEQ(A) becomes low and the Data Scan cost increases. On the other hand, for large k values (larger than 20%) very large equivalent sets are produced and SEQ(A) presents the drawbacks of SEQ(C). The best performance results when setting k to 10% of $|I|$, which is the value used henceforth.

Our next experiments considers the comparison of all SEQ methods. We used data sets that were similar to the ones used in the previous experiment. Based on Eq. (1) we used the following signature sizes: for SEQ(C) equal to 96 bits, for SEQ(P) and SEQ(A) equal to 64 and for SEQ(U) equal to 32. For SEQ(A), k was set to 10% of $|I|$ and for SEQ(P), β was set to 44 (among several examined values, the selected one presented the best performance). We measured the number of activated sequential patterns in the database. This number is equal to the total number of drops, i.e. the sum of actual- and false-drops. Evidently, for the same query, the number of actual-drops (i.e. the sequential patterns that actually satisfy the query) is the same for all methods. Therefore, the difference in the number of activated sequential patterns directly results from the difference in the number of false-drops. The results are illustrated in Fig. 8a (the vertical axis is in logarithmic scale).[6] In all cases, SEQ(A) outperforms all other methods, indicating that its approximation technique is effective in reducing the Data Scan cost through the reduction of false-drops.

Since the query performance depends both on the Data- and the Index-Scan cost, for the case of the previous experiment we measured the total number of disk accesses. The results are depicted in Fig. 8b, with respect to the query size (i.e. the number of elements in the query sequential patterns).

Focusing on SEQ(P), we see that for all query sizes it performs better than or almost the same as SEQ(C). Especially for medium size queries, the performance difference between the two methods is larger. This is due to the reduced Data Scan cost for these cases (fewer false-drops, as also given in Fig. 8a), resulted from the partitioning technique. Moving on to SEQ(U), we observe that for medium query sizes it is outperformed by SEQ(P), but for very small and large ones it performs better. These two cases present two different situations (see also the following experiment):(i) For very small queries (e.g. of size two) many signatures are activated and a large part of the database is scanned during verification. Hence a large Data Scan cost is introduced for all methods. This can be called as ‘pattern explosion’ problem. (ii) For large queries (with size comparable to S) there are not many different sequential patterns in the database with the same items but with different ordering. Therefore, ignoring the ordering does not produce many false-drops. In this case a small number of signatures is activated (see also as also given in Fig. 8a) and all methods have a very small and comparable Data Scan cost. Since at these two extreme cases both SEQ(P) and SEQ(U) have comparable Data Scan cost, SEQ(P) looses out due to the increased Index Scan cost, incurred from the use of larger signatures (SEQ(U) does not use equivalent sets, thus it uses 32-bit signatures; in contrast SEQ(P) uses for a 64-bit signature for each partition, thus the total size is a multiple of 64-bits).

Turning our attention to SEQ(A), we observe that it significantly outperforms SEQ(C) and SEQ(P) for all query sizes. Regarding the two extreme cases, for the pattern explosion problem SEQ(A) does not present the drawback of SEQ(C) and SEQ(P). In this case it performs similar to SEQ(U), which uses much smaller signatures. The same applies for the very large queries. For all the other cases, SEQ(A) clearly requires much smaller accesses than SEQ(U).

Our next series of experiments examine the sensitivity of the methods. We first focus on the effect of S. We generated data sets with the other parameters being the same with the previous experiments, and we varied the length S of sequential patterns (the signature lengths were tuned against S). The resulted numbers of disk accesses are depicted in Fig. 9a, for query size equal to $S/2$ in each case. Clearly, the disk accesses for all

[6] We choose to show the differences in the false-drops through the total number of drops, since we are interested in the relative overall performance of the methods and the direct impact on the Data Scan cost.

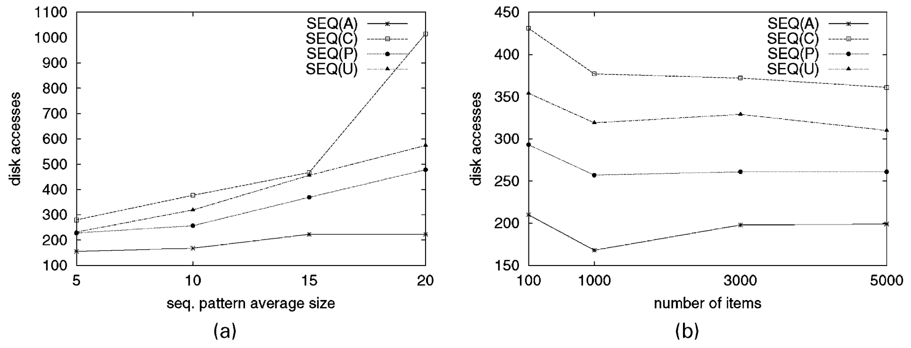

Fig. 9. Effect of: (a) S. (b) $|I|$.

methods increase with increasing S. SEQ(A) perform better than all other methods and is not affected by increasing S as much as the other methods. SEQ(P) presents the second best performance. It has to be noticed that for large values of S, the performance of SEQ(C) degenerates rapidly.

We also examined the effect of the cardinality of I (domain of items). For this experiment S was set to 10 and the average query size was equal to 5. The other parameters in the generated data sets were the same as those in previous experiments, and we varied I. The results are illustrated in Fig. 9b. As shown, for all methods, very small values of $|I|$ (e.g. 100) require a much larger number of disk accesses. This is due to the larger impact of ordering, since more permutations of the same sets of items appear within sequential patterns. SEQ(A) presents a significantly improved performance, compared to all other methods, whereas SEQ(P) comes second best.

Finally, we examined the scalability of the algorithms with respect to the number N of sequential patterns (denoted sequences for simplicity). The rest parameters for the generated data sets were the same with the ones in the experiments depicted in Fig. 8. In the generated data sets we varied N. The results depicted in Fig. 10.

As shown, for all methods, the disk accesses increase in terms of increasing N. SEQ(A) compares favorably with the remaining algorithms, whereas SEQ(P) comes second best. As in all previous results, SEC(C) presents the worst performance.

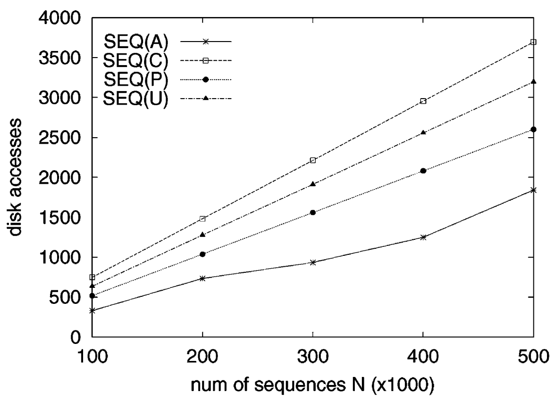

Fig. 10. Scalability w.r.t. number of sequences N.

6. Conclusions

We considered the problem of the efficient storage and querying of sequential patterns in database systems. Sequential patterns and pattern queries can constitute essential primitives for *pattern post-processing*, which involves the persistency of discovered patterns, their selection according to user-defined criteria, or the querying to identify transactions that satisfy certain criteria. Although these operations are included in existing KDD query languages, little support is provided in this direction by existing DBMS or mining tools. This impacts the tighter coupling of data mining with DBMSs and the more systematic development of data mining applications over database systems.

We described a family of algorithms, which are based on the notion of *equivalent set* to address the drawbacks of existing methods that do not consider the ordering among the elements of sequential patterns. Moreover, we considered the fact that the distribution of elements in sequential patterns is highly skewed and we proposed a novel encoding scheme, which is able to combine the advantages of the other described algorithms.

The comparison of all methods is given through experimental results, which examine a variety of factors. We tested the impact of query size, the tuning of a parameter for the method that capitalizes on the skew of element distribution, the effect of sequential-pattern lengths and the domain size, and finally the scalability. These results clearly illustrate the superiority of the proposed method against others that either do not consider the ordering of elements or are based on direct (straightforward) representation schemes. In future work, we will examine the integration of the developed scheme with tree indexes for signature data [35].

References

[1] R. Agrawal, K. Shim, Developing tightly-coupled data mining applications on relational database systems, Proceedings of International Conference on Knowledge Discovery in Databases and Data Mining (KDD'96), 1996, pp. 287–290.

[2] R. Agrawal, R. Srikant, Mining Sequential Patterns, Proceedings of IEEE International Conference on Data Engineering (ICDE'2001), 1995, pp. 3–14.

[3] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, Reading, MA, 1999.

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

[5] R. Cooley, B. Mobasher, J. Srivastava, Data preparation for mining world wide web browsing patterns, Knowledge and Information Systems 1 (1) (1999) 5–32.

[6] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, J. Karkkainen, Episode matching, Proceedings of Symposium Combinatorial Pattern Matching (CPM'97), 1997, pp. 12–27.

[7] C. Diamantini, M. Panti, A. Conceptual, indexing method for content-based retrieval, Proceedings of International Workshop on Database and Expert Systems Applications (DEXA'99), 1999, pp. 193–197.

[8] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, The KDD process for extracting useful knowledge from volumes of data, Communications of the ACM 39 (11) (1996) 27–34.

[9] R. Giegerich, S. Kurtz, J. Stoye, Efficient implementation of lazy suffix trees, Proceedings of the Third Workshop on Algorithm Engineering (WAE'99), 1999, pp. 30–42.

[10] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, D. Srivastava, Using q-grams in a DBMS for approximate string processing, Bulletin of the IEEE Technical Committee on Data Engineering 24 (4) (2001) 28–34.

[11] J. Han, Y. Fu, W. Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, O. Zaïane, DBMiner: a system for Mining Knowledge in large relational Databases, Proceedings of International Conference on Knowledge Discovery in Databases and Data Mining (KDD'96), 1996, pp. 250–255.

[12] S. Helmer, G. Moerkotte, Evaluation of main memory join algorithms for joins with set comparison join predicates, Proceedings of International Conference on Very Large Databases (VLDB'97), 1997, pp. 386–395.

[13] Helmer, S., Moerkotte, G., 1999. A study of four index structures for set-valued attributes of low cardinality. Reihe Informatik 2, University of Mannheim. pp. 20.

[14] A. Hulgeri, G. Bhalotia, C. Nakhe, S. Chakrabarti, S. Sudarshan, Keyword search in databases, IEEE Data Engineering Bulletin 24 (3) (2001) 22–32.

[15] T. Imielinski, H. Mannila, A database perspective on knowledge discovery, Communications of the ACM 39 (11) (1996) 58–64.

[16] T. Imielinski, A. Virmani, MSQL: a query language for database mining, Data Mining and Knowledge Discovery 3 (4) (1999) 373–408.

[17] T. Imielinski, A. Virmani, A. Abdulghani, DataMine: Application programming interface and query language for database mining, Proceedings of International Conference on Knowledge Discovery in Databases and Data Mining (KDD'96), 1996, pp. 256–262.

[18] M. Kitagawa, Y. Ishikawa, N. Obho, Evaluation of signature files as set access facility in OODBs, Proceedings of the ACM SIGMOD Conference on Management of Data, Santa Barbara, CA, 1993, pp. 247–256.

[19] U. Manber, E. Myers, Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.

[20] R. Meo, G. Psaila, S. Ceri, A. New, SQL-like Operator for mining association rules, Proceedings of International Conference on Very Large Databases (VLDB'96), 1996, pp. 122–133.

[21] T. Morzy, M. Wojciechowski, M. Zakrzewicz, Data mining support in database management systems, Proceedings of International Conference on Data Warehousing and Knowledge Discovery (DaWaK'2000), 2000, pp. 382–392.

[22] T. Morzy, M. Wojciechowski, M. Zakrzewicz, Optimizing pattern queries for web access logs, Proceedings of East European Conference on Advances in Databases and Information Systems (ADBIS'2001), 2001, pp. 141–154.

[23] T. Morzy, M. Zakrzewicz, SQL-Like Language for database mining, Proceedings of East European Conference on Advances in Databases and Information Systems, 1997, pp. 311–317.

[24] T. Morzy, M. Zakrzewicz, Group bitmap index: a structure for association rules retrieval, Proceedings of International Conference on Knowledge Discovery in Databases and Data Mining (KDD'98), 1998, pp. 284–288.

[25] A. Nanopoulos, D. Katsaros, Y. Manolopoulos, A data mining algorithm for generalized web prefetching, IEEE Transactions on Knowledge and Data Engineering, 2002, in press.

[26] A. Nanopoulos, Y. Manolopoulos, Finding generalized path Patterns for web log data mining, Proceedings of East-European Conference on Advances in Databases and Information Systems (ADBIS-DASFAA'2000), 2000, pp. 215–228.

[27] G. Navarro, A. Guided, Tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.

[28] G. Navarro, E. Sutinen, J. Tanninen, J. Tarhio, Indexing text with approximate q-grams, Proceedings of the 11th Annual Symposium On Combinatorial Pattern Matching (CPM'2000), 2000, pp. 350–363.

[29] A. Netz, S. Chaudhuri, U. Fayyad, J. Bernhardt, Integrating data mining with SQL databases: OLE DB for Data Mining, Proceedings of IEEE International Conference on Data Engineering (ICDE'2001), 2001, pp. 379–387.

[30] R. Ng, L. Lakshmanan, J. Han, A. Pang, Exploratory mining and pruning optimizations of constrained association rules, Proceedings of ACM International Conference on Management of Data (SIGMOD'98), 1998, pp. 13–24.

[31] J. Pei, J. Han, B. Mortazavi-Asl, H. Zhu, Mining, Access patterns efficiently from web Logs, Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'2000), 2000.

[32] M. Perkowitz, O. Etzioni, Adaptive web sites; an AI challenge, Proceedings of the 15th International Joint Conference of AI, 1997.

[33] E. Riedel, C. Faloutsos, G. Ganger, D. Nagle, Data mining on an OLTP system (Nearly) for free, Proceedings of ACM International Conference on Management of Data (SIGMOD'2000), 2000, pp. 13–21.

[34] S. Sarawagi, S. Thomas, R. Agrawal, Integrating mining with relational database systems: alternatives and Implications, Proceedings of ACM International Conference on Management of Data (SIGMOD'98), 1998, pp. 343–354.

[35] E. Tousidou, A. Nanopoulos, Y. Manolopoulos, Improved methods for signature-tree Construction, The Computer Journal 43 (4) (2000) 301–314.

[36] Z. Tronicek, Episode matching, Proceedings of Symposium Combinatorial Pattern Matching (CPM'2001), 2001, pp. 143–146.

[37] Virmani, A., 1998. Second generation data mining: concepts and implementation. PhD Thesis, Rutgers University.

[38] M. Zakrzewicz, Sequential index structure for content-based retrieval, Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'2001), 2001, pp. 306–311.