Chapter IX

Indexing Techniques for Web Access Logs

Yannis Manolopoulos, Aristotle University of Thessaloniki, Greece

Mikolaj Morzy, Poznan University of Technology, Poland

Tadeusz Morzy, Poznan University of Technology, Poland

Alexandros Nanopoulos, Aristotle University of Thessaloniki, Greece

Marek Wojciechowski, Poznan University of Technology, Poland

Maciej Zakrzewicz, Poznan University of Technology, Poland

ABSTRACT

Access histories of users visiting a web server are automatically recorded in web access logs. Conceptually, the web-log data can be regarded as a collection of clients' access-sequences, where each sequence is a list of pages accessed by a single user in a single session. This chapter presents novel indexing techniques that support efficient processing of so-called pattern queries, which consist of finding all access sequences that contain a given subsequence. Pattern queries are a key element of advanced analyses of web-log data, especially those concerning typical navigation schemes. In this chapter, we discuss the particularities of efficiently processing user access-sequences with pattern queries, compared to the *case of searching unordered sets. Extensive experimental results are given, which examine a variety of factors and illustrate the superiority of the proposed methods over indexing techniques for unordered data adapted to access sequences.*

INTRODUCTION

Web servers have recently become the main source of information on the Internet. Web access logs record the access history of users who visit a web server. Each web-log entry represents a single user's access to a web resource (HTML document, image, CGI program, etc.) and contains the client's IP address, the timestamp, the URL address of the requested resource, and some additional information. Conceptually, the web-log data can be regarded as a collection of clients' access-sequences, where each sequence is a list of pages accessed by a single user in a single session. Extraction of user access-sequences is a required pre-processing step in advanced analyses of web logs (called web-log mining), and it involves data cleaning and techniques of forming user sessions (see Cooley, Mobasher, & Srivastava, 1999; Lou, Liu, Lu, & Yang, 2002).

One of the most popular data mining problems in the context of web-log analysis is discovery of access patterns (Chen, Park, & Yu, 1998; Pei, Han, Mortazavi-Asl, & Zhu, 2000). Each access pattern is a sequence of web pages which occurs frequently in user access-sequences. Sequential access-patterns provide information about typical browsing strategies of users visiting a given website, e.g., "10% of users visited the page about a palmtop X, and later a page about a docking cradle for the palmtop X." After some frequently occurring sequences have been discovered, the analyst should be able to search for user access-sequences that support (i.e., contain) the patterns. The latter operation finds several applications, e.g., searching for typical/atypical user access-sequences.

Moreover, web-log mining algorithms, such as WUM (Spiliopoulou & Faulstich, 1998), use templates to constrain the search space and to perform more focused mining, according to the user's requirements. For instance, the user may specify the mining of sequences containing the subsequence <A * B * C>. Thus, a selection of the user access-sequences can be performed to collect those satisfying the given template. In the following, we refer to these types of queries over the database of user access-sequences as pattern queries.

Since web logs tend to be large, a natural solution to support efficient processing of pattern queries would be indexing web access-sequences.

Unfortunately, no indexes specific for this purpose have been proposed before, and existing indexing techniques for single-valued and set-valued attributes are not applicable or are inefficient, as they do not take ordering into consideration. These techniques can be applied merely to locate sequences built from the same set of elements as the query sequence, likely introducing many false drops if the actual task is a subsequence search.

In this chapter, we describe indexing methods for large collections of access sequences extracted from web access logs. The target of the chapter is twofold. First, it intends to clarify the particularities of efficiently processing user access-sequences with pattern queries, compared to the case of searching unordered sets. It will demonstrate how these particularities make the existing (traditional) solutions inadequate, and will show how they call for different index structures and searching algorithms. The second objective of this chapter is to organize recent research that has been conducted in the area of pattern queries (to a significant extent by authors of this chapter) and to present it in an integrated and comparative way. The key concept is the development of a family of methods, based on signatures capturing the presence of certain elements in a sequence as well as the ordering between the sequence elements (a factor that has not been examined by existing signature schemes). Emphasis is placed on scalability to web-logs' sizes. Extensive experimental results are given, which examine a variety of factors and illustrate the superiority of the proposed methods over indexing techniques for unordered data adapted to access sequences.

The rest of this chapter is organized as follows. We start with the introduction to web-log analysis, the reasons for indexes for web logs, and the critique of existing indexing techniques. This is followed by the description of the family of novel sequence-indexing methods for pattern queries. Next, the experimental results on the comparison of the described methods are presented. Finally, the discussion of possible future trends and some conclusions are given.

ANALYSIS OF WEB ACCESS LOGS

Each access to a web resource made by a browser is recorded in the web server's log. An example of the log's contents is depicted in Figure 1. Each entry represents a single request for a resource and contains the IP address of the requesting client, the timestamp of the request, the name of the method used and the URL of the resource, the return code issued by the server, and the size

Figure 1: An Example of a Web Log

```
150.254.31.173  -- [21/Jan/2003:15:48:52 +0100] "GET /mmorzy " 301 328
150.254.31.173  -- [21/Jan/2003:15:48:52 +0100] "GET /mmorzy/index.html " 200 9023
150.254.31.173  -- [21/Jan/2003:15:48:52 +0100] "GET /mmorzy/acrobat.gif " 304
144.122.228.120 -- [21/Jan/2003:15:48:56 +0100] "GET /imgs/pp1.gif " 200 2433
150.254.31.173  -- [21/Jan/2003:15:48:58 +0100] "GET /mmorzy/research.html " 200 8635
60.54.23.11     -- [21/Jan/2003:15:48:59 +0100] "GET /mmorzy/db/slide0003.htm " 200
24808
150.254.31.173  -- [21/Jan/2003:15:49:03 +0100] "GET /mmorzy/students.html " 200 7517
150.254.31.173  -- [21/Jan/2003:15:49:08 +0100] "GET /mmorzy/db_course.html " 200 10849
144.122.228.120 -- [21/Jan/2003:15:49:16 +0100] "GET /reports/repE.html " 200 76685
150.254.31.173  -- [21/Jan/2003:15:49:22 +0100] "GET /mmorzy/html.gif " 200 1038
150.254.31.173  -- [21/Jan/2003:15:49:22 +0100] "GET /mmorzy/zip.gif " 200 1031
144.122.228.120 -- [21/Jan/2003:15:50:03 +0100] "GET /imgs/polish.gif " 200 109
```

of the requested object (for brevity, we omit additional details recorded in the log, such as the protocol or the detailed identification of the browser).

Information stored in the web log can be directly used for simple statistical analyses in order to derive such measures as frequency of accesses to particular resources, numbers of accesses from individual network addresses, number of requests processed in a time unit, etc. However, for advanced and reliable analyses of the way users navigate through the website, information from the log needs processing and cleansing before it can be used. Several requests can have identical timestamps because they represent accesses to different elements of a single web page (in our example, the second entry represents an access to a web page `index.html`, whereas the third entry represents a retrieval of the image `acrobat.gif`, which is probably displayed on that page). Secondly, different records in a web log can have identical IP address and still refer to different users. For example, some users may access the web page from behind a proxy server. Some access paths might not be recorded in the log because browsers cache recently visited pages locally. In order to use the web log for advanced analysis, it must be transformed into a set of clients' access-sequences, where each sequence describes a navigation path of a single user during a single session. Interesting descriptions of web-log transformation and cleansing techniques can be found in Cooley et al. (1999).

Figure 2 presents an example client sequence derived from the web log from Figure 1. This sequence represents the session of a student who started from the teacher's main page (`index.html`) and navigated to a research page (`research.html`). The next request was made for the page containing information for students (`students.html`), and from there, the student went to the database course page (`db_course.html`). The analysis of the website's structure revealed that there was no link between `research.html` and `students.html`, so probably the student used the browser's back button.

Figure 2: An Example of a Client's Access Sequence

```
/mmorzy/index.html → /mmorzy/research.html → /mmorzy/students.html →
/mmorzy/db_course.html
```

A client's sequences extracted from the web log can be stored in a database and further analyzed to discover common access patterns. Such frequent patterns, which are subsequences occurring in a large fraction of the client's sequences, are called sequential patterns (Agrawal & Srikant, 1995). Sequential patterns can be used in several ways, e.g., to improve website navigation, to personalize advertisements, to dynamically reorganize link structure and adapt website contents to individual client requirements, or to provide clients with automatic recommendations that best suit customer profiles. Interpretation of discovered access patterns involves extracting access sequences that contain a given pattern (subsequence). If no indexes are available for the web log, such an operation requires a costly sequential scan of the whole web-log data. Another example of an operation that would benefit from web-log indexes is focused pattern mining that is to be confined to access sequences which contain a given subsequence (Spiliopoulou & Faulstich, 1998).

REVIEW OF EXISTING INDEXING TECHNIQUES

Traditional database systems provide several indexing techniques that support single tuple access based on single attribute value. The most popular indexes include B-trees (Comer, 1979), bitmap indexes (Chan & Ioannidis, 1998), and R-trees (Guttman, 1984). Contemporary database systems allow for storage of set-valued attributes, either in the form of abstract data types (ADTs) or nested tables. However, traditional indexes do not provide mechanisms to efficiently query such attributes, despite of the fact that the need for subset search operators has been recognized (Graefe & Cole, 1995). Indexing of set-valued attributes was seldom researched and resulted in few proposals.

The first access methods for set-valued attributes were developed in the area of text retrieval systems. Signature files (Faloutsos & Christodoulakis, 1984) and signature trees (S-trees) (Deppisch, 1986) utilize the idea of superimposed coding of bit vectors. Each element is represented by a fixed-width signature, with *m* bits set to '1.' Signatures are superimposed by a bitwise

OR operation to create a set representation. Signatures can be stored in a sequential signature file or a signature tree, or they can be stored using extendible signature hashing. Implementation details and performance evaluation of different signature indexes can be found in Helmer (1997) and Helmer and Moerkotte (1999). Improved methods for signature tree construction were proposed in Tousidou, Nanopoulos and Manolopoulos (2000).

Another set-indexing technique, proposed initially for text collection indexing, is inverted file (Araujo, Navarro, & Ziviani, 1997). Inverted file consists of two parts: the vocabulary and the occurrences. The vocabulary contains all elements that appear in indexed sets. A list of all sets containing a given element is maintained along with each element. All lists combined together form the occurrences. Inverted files are very efficient for indexing small and medium-sized collections of elements. An exhaustive overview of text indexing methods and pattern matching algorithms can be found in Baeza-Yates and Ribeiro-Neto (1999).

Indexing of set-valued attributes attracted the attention of researchers from the object-oriented database systems domain. These studies resulted in the evaluation of signature files in an object-oriented database environment (see Ishikawa, Kitagawa, & Ohbo, 1993; Nørvåg, 1999) and in the construction of the nested index (Bertino & Kim, 1989).

An interesting proposal stemmed from a modification of a well-known R-tree index, namely a Russian Doll tree (RD-tree) (Hellerstein & Pfeffer, 1994). The structure of the tree reflects a transitive containment relation. All indexed sets are stored in tree leafs, while inner nodes hold descriptions of sets contained in the child nodes. Descriptions can be complete representations of bounded sets, signatures or Bloom filter (Bloom, 1970) representations, range set representations, or hybrid solutions.

The first proposals of specialized indexes for set-valued attributes in the domain of data mining were formulated by Morzy and Zakrzewicz (1998). Two indexes were presented: a group bitmap index and a hash group bitmap index. The first index uses a complete and exact representation of indexed sets, but results in very long index keys. Every set is encoded as a bitmap of length n, where n denotes the number of all possible items appearing in the indexed sets. For each set, the i-th bit is set to '1' if this set contains item i. A subset search using a group bitmap index consists in bitwise comparison of index bitmaps, with the bitmap representing the searched subset. A hash group bitmap index uses a technique similar to Bloom filter. It represents indexed sets approximately, by hashing set elements to a bitmap of fixed length. The length of this bitmap is usually much smaller than the number of all possible elements. This technique allows some degree of ambiguity, which results in false drops and implies the verification of answers obtained from the index.

All indexing techniques for set-valued attributes do not consider the ordering of elements within the set. Therefore, those indexes are not suitable for sequence queries. However, set-indexing techniques can be applied to locate sequences built from the same elements as a given query sequence. Using those indexes for a subsequence search requires an additional post-processing step, in which all answers returned from the index are checked for a real containment of the searched sequence, and all sequences containing the searched elements in a wrong sequence, so-called false drops, are pruned. The number of false drops can be huge when compared to the number of correct answers, so this verification step adds significant overhead to query processing and will likely lead to unacceptable response times for sequential queries (for sequential patterns, many sequences contain searched elements but not necessarily in the correct order). Nevertheless, such an adaptation of set-oriented indexes can be considered a natural reference point for evaluating the performance of novel sequence-indexing techniques.

SEQUENTIAL DATA INDEXING METHODS FOR PATTERN QUERIES

In this section, we provide a formal definition of a pattern query over a database of web-log sequences, and describe sequential data indexing methods to optimize pattern queries. The indexing methods are built upon the concepts of equivalent sets, their partitioning, and their approximations.

Basic Definitions

Definition 1 (Pattern query). Let I be a set of *items*. A *data sequence* X is defined as an ordered list of items. Thus, $X = <x_1 x_2 ... x_n>$, where each $x_i \in I$ (x_i is called an *element* of X). We say that a data sequence $X = <x_1 x_2 ... x_n>$ *is contained* in another data sequence $Y = <y_1 y_2 ... y_n>$ if there exist integers $i_1 < i_2 < ... < i_n$ such that $x_1 = y_{i1}$, $x_2 = y_{i2}$, ..., $x_n = y_{in}$.

Given a database D of data sequences and a data sequence Q, a *pattern query* consists of finding in D all data sequences that contain Q. In other words, a pattern query formulates a problem of finding all data sequences containing a set of user-defined elements in a specified order.

Definition 2 (Equivalent set). In order to represent data sequences in a compact way, we introduce the concept of an *equivalent set*.

An *item mapping function $f_i(x)$*, where $x \in I$, is a function which transforms a literal into an integer value (we assume that 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.

An *order mapping function $f_o(x,y)$*, where $x, y \in I$ and $f_o(x,y) \neq f_o(y,x)$, is a function which transforms an item sequence $<x\,y>$ into an integer value. It has to be noticed that the intuition for the use of $f_o(x, y)$ is that it takes into account the ordering, i.e., $f_o(x, y) \neq f_o(y, x)$. Henceforth, we assume order mapping functions of the form $f_o(x,y) = a * f_i(x) + f_i(y)$, where a is greater than the largest $f_i(x)$ value (i.e. $f_o(x,y) \neq f_o(y,x)$ and f_o values are always larger than f_i values).

Given a sequence $X = <x_1 x_2 \ldots x_n>$, the *equivalent set E* of X is defined as:

$$E = \left(\bigcup_{x_i \in X} \{f_i(x_i)\} \right) \cup \left(\bigcup_{x_i, x_j \in X,\, i<j} \{f_o(x_i, x_j)\} \right)$$

where: f_i is an item mapping function and f_o is an order mapping function.

Example 1: For instance, for $I = \{A, B, C, D, E\}$, we have $f_i(A) = 1$, $f_i(B) = 2$, $f_i(C) = 3$, $f_i(D) = 4$, $f_i(E) = 5$, and $f_o(x, y) = 6 * f_i(x) + f_i(y)$ (e.g., $f_o(A,B) = 8$). Given a sequence $X = <A, C, D>$, using the mapping functions that were described above, we get: $E = (\{f_i(A)\} \cup \{f_i(C)\} \cup \{f_i(D)\}) \cup (\{f_o(A, C)\} \cup \{f_o(A, D)\} \cup \{f_o(C, D)\}) = \{1, 3, 4, 9, 10, 22\}$.

According to Definition 2, an equivalent set is the union of two sets: the one resulting by considering each element separately, and the other from considering pairs of elements. $S(E)$ denotes the former set, consisting of values of f_i, and $P(E)$ the latter set, consisting of values of f_o. Based on Definition 2, it is easy to show the following.

Corollary 1. Let two sequences 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 1 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 $P(E_Q) \subseteq E_P$.

Equivalent Set Signatures

Equivalent sets can be efficiently represented with *superimposed signatures*.

A signature is a bitstring of L bits (L is called signature *length*) and is used to indicate the presence of elements in a set. Using a hash function, each element of a set can be encoded into a signature that has exactly m out of L bits equal to '1' and all other bits equal to '0'. The value of m is called the *weight* of the element. 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 $sig(E_1)$, $sig(E_2)$, it holds that $E_1 \subseteq E_2 \Rightarrow sig(E_1)$ AND $sig(E_2) = sig(E_1)$.

Signatures provide a quick filter for testing the subset relationship between sets. Therefore, if there exist any bits of $sig(E_1)$ that are equal to '1,' and the corresponding bits of $sig(E_2)$ 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 an actual drop or a false drop), we have to examine the corresponding sequences with the containment criterion. In order to minimize the number of false drops, it has been proved (Faloutsos & Christodoulakis, 1984) that, for sets of size T, the length of the signatures has to be equal to: $L = m * T / \ln 2$.

Henceforth, we assume that m is equal to 1 [based on the approach from Helmer and Moerkotte (1997)], and the signature of the element x is the binary representation of the number $2^{x \bmod L}$ (with the least significant bit on the left). Given a collection of sequences, in the following section we examine effective methods for organizing the representations of the patterns, which consist of signatures of equivalent sets.

Family of Sequence Indexing Methods

We describe three methods of sequential data indexing to optimize pattern queries: SEQ(C)—which uses complete signatures of equivalent sets; SEQ(P)—which uses signatures of partitioned equivalent sets; and SEQ(A)—which uses signatures of approximated equivalent sets. Next, we discuss the possibility of extending the methods with advanced tree structures to store the signatures.

A Simple Sequential Index (SEQ(C) – "Complete")

Let *D* be the database of sequences to be indexed. A simple data structure for indexing elements of *D* is based on the paradigm of signature file (Faloutsos & Christodoulakis, 1984), 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. The construction algorithm for SEQ(C) is given in Figure 3a (*sig*(*E*) is the signature of equivalent set *E*).

Example 2: Let us consider the following example of SEQ(C) index entry construction. Assume the data sequence to be indexed is $X = <A, C, D, E>$. Assume the set *I* of items and item mapping functions and order mapping functions from Example 1, and L=10.
The equivalent set for the data sequence *X* is the following: $E = \{f_i(A), f_i(C), f_i(D), f_i(E), f_o(A,C), f_o(A,D), f_o(A,E), f_o(C,D), f_o(C,E), f_o(D,E)\} = \{1, 3, 4, 5, 9, 10, 11, 22, 23, 29\}$
Therefore, the SEQ(C) index entry will consist of the following signature (starting with the least significant bit): $sig(E) = 1111110001$.

The algorithm for querying the structure for a given sequence *Q* is given in Figure 3b. Initially (step 1), the equivalent set, E_Q, of *Q* is calculated. Then, each signature in the structure is examined against the signature $sig(E_Q)$ (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 sequences 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 (see, e.g., Faloutsos & Christodoulakis, 1984; Helmer & Moerkotte, 1997; Tousidou, Nanopoulos, & Manolopoulos, 2000).

Figure 3: SEQ(C) Method: (a) Construction Algorithm (b) Search Algorithm

(a)

```
1. $F = \emptyset$
2. **forall** $P \in D$
3.   $E$ = Equivalent_Set($P$)
4.   $F = F \cup \{<sig(E), \text{pointer}(P)>\}$
5. **endfor**
```

(b)

```
1. $E_Q$ = Equivalent_Set($Q$)
2. $R = \emptyset$
3. **forall** $<s, \text{pointer}(P)> \in F$
4.   **if** ($s$ AND $sig(E_Q)$) = $sig(E_Q)$
5.       Retrieve $P$ from $D$
6.       **if** $Q$ is contained in $P$
7.           $R = R \cup \{P\}$
8.       **endif**
9.   **endif**
10. **endfor**
```

For SEQ(C), the calculation of L (signature length) is done using the expected size of equivalent sets $|\overline{E}|$ (in place of T). Since $|\overline{E}|$ grows rapidly, L can take large values, which increases the possibility of collisions during the generation of signatures (i.e., elements that are hashed into the same position within signatures). Collisions result in 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 in larger L values and, consequently, in 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).

Partitioning of Equivalent Sets (SEQ(P) – "Partitioned")

In Zakrzewicz (2001) and Morzy, Wojciechowski and Zakrzewicz (2001), 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 have a reduced collision probability, fewer false drops, and a reduced Data Scan cost.

Definition 3 (Partitioning of equivalent sets). Given a user-defined value β, the equivalent set E of a data sequence P is partitioned into a collection of $E_1, \ldots, E_p$ subsets: (1) by dividing P into $P_1, \ldots, P_p$ subsequences such that $\bigcup_{i=1}^{p} P_i = P$, $P_i \cap P_j = \varnothing$ for $i \neq j$; and (2) by having E_i be the equivalent set of P_i, where $|E_i| < \beta$, $1 \leq i \leq p$.

According to Definition 3, 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 size 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.

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 Figure 3a. After step 3, we have to insert:

3a. Patrition E into $E_1, \ldots, E_p$

and step 4 is modified accordingly:

4. $F = F \cup \{<sig(E_1), \ldots, sig(E_p), \text{pointer}(P)>\}$

Example 3: Let us consider the following example of SEQ(P) index entry construction. Assume the data sequence to be indexed is $X = <A, C, D, E>$. Assume the set I of items and item mapping functions and order mapping functions from Example 1, β=4 and L=4.

The data sequence X will be split into the following two data sequences: $X_1 = <A, C>$ and $X_2 = <D, E>$.

Next, we give the equivalent sets for the two data sequences. Notice the sizes of both sequences do not exceed β.

$E_1 = \{f_i(A), f_i(C), f_o(A,C)\} = \{1, 3, 9\}$
$E_2 = \{f_i(D), f_i(E), f_o(D,E)\} = \{4, 5, 29\}$

Therefore, the SEQ(P) index entry will consist of the following signatures:

$sig(E_1) = 0101$
$sig(E_2) = 1100$

The search algorithm for SEQ(P) is based on the following observation (Morzy et al., 2001): For each partition of an equivalent set E, a query pattern

Figure 4: SEQ(P) Method: Search Algorithm

```
1. $R = \varnothing$
2. forall Equivalent Sets $E = E_1 \ldots E_p$ stored as <$sig(E_1)$, …, $sig(E_p)$, pointer($P$)>
3.     $startPos = 0$
4.     for ($i$=1; $i \le p$ 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 $sig(E_Q)$ AND $sig(E_i) = sig(E_Q)$
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 $Q$ is contained in $P$
20.            $R = R \cup \{P\}$
21.        endif
22.    endif
23. endfor
```

Q can be decomposed into a number of subsequences. Each subsequence is separately examined against the partitions of E. The algorithm is depicted in Figure 4.

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 sequence (step 1). At steps 4-16, the query pattern is examined against each partition, and the maximum query part that can be matched by the current partition is identified. The part of query sequence 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 in 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 keeping in mind that the size of each partition of the equivalent set is equal to β (Definition 3)]. Thus, the Index Scan cost may be increased (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).

Using Approximations of Equivalent Sets (SEQ(A) – "Approximate")

In Nanopoulos, Zakrzewicz, Morzy and Manolopoulos (2003), a different method for organizing equivalent sets is proposed. 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 Agrawal and Srikant (1995)] have larger appearance frequency. Therefore, the pairs of elements that are 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 sizes of equivalent sets can be reduced by taking into account only the pairs with high co-occurrence probability. This leads to approximations 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 sizes of equivalent sets (so as to reduce Data Scan costs), with a reduction in the lengths 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 an ordered pair (x, y) in D (i.e., the normalized frequency of sequence $<x\ y>$ (Agrawal & Srikant, 1995)), where x, y Î I and the pair $(x, y) \in P(E)$. The construction algorithm for SEQ(A) is depicted in Figure 5.

SEQ(A) reduces the sizes of equivalent sets by considering for each element $i \in I$, only k most frequent ordered pairs, with i as the first element. In step 2 of the algorithm, those frequent pairs are discovered and represented in the form of *NN* sets containing k most frequent successors for each element. In steps 8-10, the *NN* sets are used to filter out infrequent pairs.

Example 4: Let us consider the following example of SEQ(A) index entry construction. Assume the data sequence to be indexed is $X = <A, C, D,$

Figure 5: SEQ(A) Method: Construction Algorithm

```
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.    $F = F \cup \{<sig(E), \text{pointer}(P)>\}$
13. endfor
```

E>. Assume the set *I* of items and item mapping functions and order mapping functions from Example 1, *k*=1 and *L*=10. Given are the following support values for the element pairs: $supp_D(D,E) = 0.280$, $supp_D(E,B) = 0.220$, $supp_D(A,C) = 0.120$, $supp_D(C,E) = 0.101$, $supp_D(A,D) = 0.075$, $supp_D(C,D) = 0.053$, $supp_D(A,E) = 0.040$, , $supp_D(B,A) = 0.037$, and, for the other pairs, i.e, *(A,B), (B,C), (B,D), (B,E), (C,A), (C,B), (D,A), (D,B), (D,C), (E,A), (E,C), (E,D)* the support values less than 0.037.

Based on the support values of the element pairs, we construct the *NN* sets for each item. Each *NN* set contains only one item (*k*=1) which forms the strongest pair:

$$NN(A) = \{C\},\ NN(B) = \{A\},\ NN(C) = \{E\},\ NN(D) = \{E\},\ NN(E) = \{B\}$$

The equivalent set will not represent the pairs which are not represented in the *NN* sets:

$$E = \{f_i(A), f_i(C), f_i(D), f_i(E), f_o(A,C), f_o(C,E), f_o(D,E)\} = \{1, 3, 4, 5, 9, 23, 29\}$$

Therefore, the SEQ(A) index entry will consist of the following signature:

$$sig(E) = 0101110001.$$

The search algorithm of SEQ(A) is analogous to that of SEQ(C). However, step 1 of the algorithm depicted in Figure 3b 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**
```

During the generation of the approximation of the query's equivalent set, the *NN* sets are used, which implies that they have to be kept in memory. However, this presents a negligible space overhead.

Lemma 1. The SEQ(A) algorithm correctly finds all sequences that satisfy a given pattern query.

Proof. Let Q denote a pattern query and E_Q denote its equivalent set. Also, let S denote a sequence for which Q is contained in S, and let E_S denote its equivalent set. As described (see Corollary 1), 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 Figure 5). 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 x 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 is contained in 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 1).

From the above it follows that, although the SEQ(A) method is based on the concept of approximation, no loss in precision is triggered (evidently, there is no reason to measure the precision/recall, since the method is always accurate). 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 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 SEQ(C), due to the fewer false drops introduced by the drastic reduction in the sizes of equivalent sets. This is examined, experimentally, in the next section. It should be noted that the selection of the user-defined parameter *k* for the calculation of the *NN* set in algorithm of Figure 5 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 0s, 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 the next section.

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. Thus, SEQ(A) combines the advantages of both SEQ(P) and SEQ(C).

Using Tree Structures to Store Signatures

SEQ(C), SEQ(P), and SEQ(A) assume that the elements of *F* (computed signatures together with pointers to data sequences) are to be stored in a sequential signature file. Nevertheless, SEQ(A) (and SEQ(C)) lead to one signature for each equivalent set. As a consequence, improved signature indexing methods are applicable in these cases, for instance, the S-tree (Deppisch, 1986).[1] By using a tree structure, we can avoid checking each signature during the search of those that satisfy the subset criterion (so as to answer the corresponding pattern query).

The S-tree is a height-balanced tree, having all leaves at the same level. Each node contains a number of pairs, where each pair consists of a signature and a pointer to the child node. In an S-tree, the root can accommodate at least two and at most M pairs, whereas all other nodes can accommodate at least m and at most M pairs, where $1 \le m \le M/2$.

Example 5: Let us consider the following example of S-tree construction. Assume that the index type is SEQ(C) and the data sequences to be indexed are the following:

X_1=<A,B,D> X_2=<C,D> X_3=<A,E> X_4=<A,C,D>
X_5=<A,D> X_6=<B,D> X_7=<B,C,E> X_8=<A,D,E>.

Assume also that the item mapping functions and order mapping functions are the same as in Example 1, and that L=10. The equivalent sets for data sequences are the following:

$E_1=\{1,2,4,8,10,16\}$ $E_2=\{3,4,22\}$ $E_3=\{1,5,11\}$ $E_4=\{1,3,4,9,10,22\}$
$E_5=\{1,4,10\}$ $E_6=\{2,4,16\}$ $E_7=\{2,3,5,15,17,23\}$ $E_8=\{1,4,5,10,11,29\}$

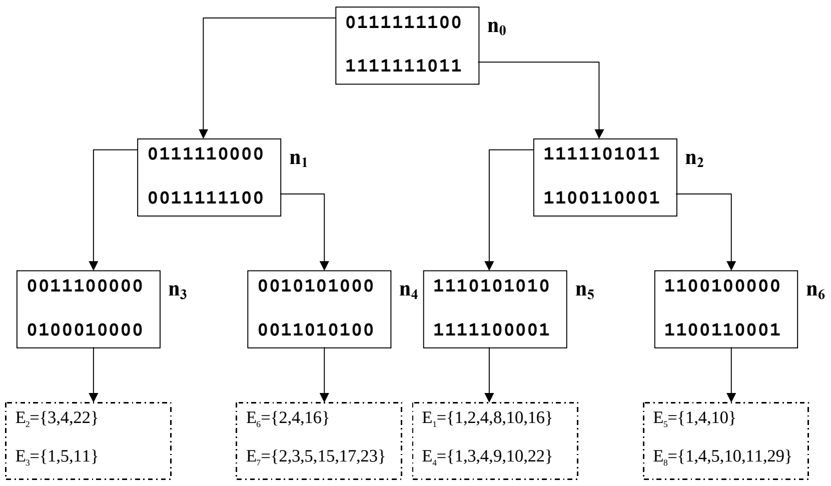
The signatures corresponding to the equivalent sets are presented below:

$sig(E_1)=1110101010$ $sig(E_2)=0011100000$ $sig(E_3)=0100010000$ $sig(E_4)=1111100001$
$sig(E_5)=1100100000$ $sig(E_6)=0010101000$ $sig(E_7)=0011010100$ $sig(E_8)=1100110001$

In Figure 6, an example of an S-tree with three levels is depicted. The nodes at the data level, which are depicted with a dashed line, represent the indexed equivalent sets. The leaves contain the signatures of equivalent sets. The signatures are assigned to the S-tree leaves in a random order. The signatures in internal nodes are formed by superimposing the signatures of their children nodes. For instance, the two signatures of node n_3 are superimposed to form the signature 0111110000 that is stored in the parent node n_1. It has to be noticed that, due to the hashing technique used to extract the object signatures, and due to the superimposition of child signatures, the S-tree may contain duplicate signatures corresponding to different objects.

Successful searches in an S-tree (for equivalent sets that are supersets of the query equivalent set) proceed as follows. First, we compute the signature of the query equivalent set. Next, we start from the root of the S-tree, comparing the query signature with the signatures in the root. For all signatures

Figure 6: Example of a Signature Tree



of the root that contain 1s, at least at the same positions as the query signature, we follow the pointers to the corresponding children nodes. Evidently, more than one signature may satisfy this requirement. The process is repeated recursively for all these children, down to the leaf level, following multiple paths. Thus, at the leaf level, all signatures satisfying the subset query are found, leading to the data nodes, whose equivalent sets are checked so as to discard the false drops.[2] In case of an unsuccessful search, searching in the S-tree may stop early at some level above the leaf level (this happens when the query signature has 1s at positions, where the stored signatures have 0s).

The S-tree is a dynamic structure that allows for insertions/deletions of signatures. In Nanopoulos, Zakrzewicz, Morzy and Manolopoulos (2002), the organization of signatures is done with the use of enhanced signature-tree indexing methods, based on Tousidou et al. (2000) and Nanopoulos and Manolopoulos (2002). The advantages of the latter approaches over the original S-tree (Deppisch, 1986) are with respect to the split policy that is used. Due to space restrictions, more details can be found in Tousidou et al. (2000) and Nanopoulos and Manolopoulos (2002).

Finally, we have to notice that both the approximation method and the method that uses the complete representation of equivalent sets can use an S-tree structure. Nevertheless, in the following, we mainly focus on the use of S-trees for the approximation method because the method that uses the complete representation tends to produce saturated signatures (full of 1s), which does not yield S-trees with good selectivity.

PERFORMANCE RESULTS

In this section, we present the experimental results concerning the performance of the examined sequence indexing methods, namely SEQ(C), SEQ(P), and SEQ(A). As we mentioned earlier, to the best of our knowledge, no other sequence-indexing methods applicable to web-log data have been proposed so far. Therefore, as a reference point, from existing indexing methods we choose one of the set-indexing methods — hash group bitmap index (Morzy & Zakrzewicz, 1998). This method can be applied to subsequence searches in the following manner. The index is built on sets of elements forming web-log sequences. Then, it can be used to locate sequences built from the same elements as the query sequence. The hash group bitmap index, similarly to our sequence indexing methods, can return false drops, so the results obtained by using it have to be verified (in the original proposal by using the subset test). Since we apply this index for subsequence search, in the verification step, the subsequence test with the query sequence is performed. We can consider this approach as a sequence indexing method based on generating signatures directly from sets of elements forming access sequences, ignoring the ordering of elements. In our performance study, the latter method is denoted as SEQ(U) ("U" stands for unordered). It should be noted that by comparing to an adaptation of a set-indexing method that is based on the same idea as the three proposed sequence-indexing methods, we can evaluate the advantages of taking element ordering into account. Finally, we examine the method that combines the approximation technique and the indexing with signature trees (this method is denoted as TREE(A)).

All methods were implemented in Visual C++, and the experiments were run on a PC with 933 MHz Intel Pentium III Processor, 128 MB RAM, under the MS Windows 2000 operating system. For the experiments, we used both synthetic and real data sets. The former are detailed in the sequel (i.e., we describe the synthetic data generator). Regarding the latter, we have tested several real web access logs. For brevity, we present results on the *ClarkNet* web log[3], which, after cleansing, contained 7200 distinct URLs organized into 75,000 sequences.

Table 1 summarizes the parameters that are used henceforth.

Synthetic Data Generation

In order to evaluate the performance of the algorithms over a large range of data characteristics, we generated synthetic sets of user sequences. Our data generator considers a model analogous to the one described in Agrawal and

Table 1: Summary of the Parameters Used

Symbol	Definition
I	Domain of items (distinct pages).
M	Number of possible frequent sequences.
N	Number of sequences in the data set.
S	Average length of sequences.
k	Number of most frequent successors considered for each item (for the approximation method).

Srikant (1995). Following the approach of Zakrzewicz (2001) and Morzy et al. (2001) (so as to examine the worst case for equivalent sets), we consider sequences with elements being single items (singletons). Our implementation is based on a modified version of the generator developed in Nanopoulos, Katsaros and Manolopoulos (2003), which was used to produce synthetic web-user traversals that consist of single items (see also Morzy et al., 2001).

The generator builds a pool of sequences, each being a sequence of pair-wise 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 effects of item ordering within sequences, we modified the generator of Nanopoulos et al. (2002) so as to perform a random permutation of the common items before inserting them in the new pool sequence. This results in sequences that contain items with different ordering, thus, examining 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 user sequence is created by picking a sequence from the pool and tossing an *M*-sided weighted coin (*M* is the pool size), where the weight for a side is the probability of picking the corresponding pool sequence. In each user sequence, 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 resulting user sequences. The length of the sequence determines this number, which is a random variable following Poisson distribution with a given mean value denoted as *S*. The total number of generated sequences is denoted as *N*. Each result presented in the following is the average of five generated data sets, and, for each data set, we used 100 queries for each case (e.g., query size, number of sequences, etc.).

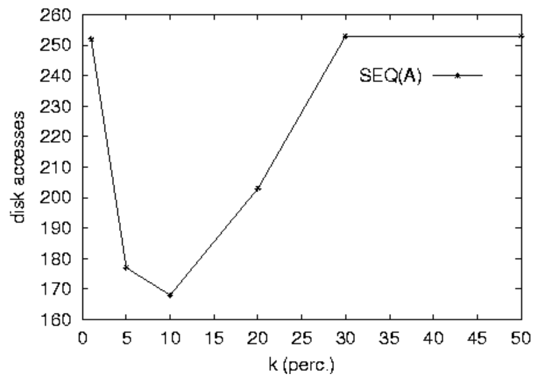
Results with Synthetic Data

In this section, we are interested in comparing all methods against several data characteristics. For this reason, we used synthetic data sets. In order to present the results more clearly, we focus on the SEQ algorithms, whereas results on the combination of the approximation technique with tree structures (i.e., TREE(A)) are given separately in the following section. This is because our objective is to examine the effectiveness of the approximation technique and, since SEQ(A) and TREE(A) use the same approximation technique, they lead to the same gains with respect to the data-scan cost.[4]

First, we focus on the tuning of k for SEQ(A). We used data sets with S set to 10, $|I|$ set to 1,000 and N equal to 50,000. We measured the total number of disk accesses (cost to scan both the index and the data) with respect to the length of the query sequences. The results for SEQ(A) with respect to k are depicted in Figure 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 equivalent sets are produced that give signatures with almost all bits equal to '0.' Thus, as has been explained, the filtering of SEQ(A) becomes low and the cost to scan the data increases. On the other hand, for large k values (more than 20%), very large equivalent sets are produced, and SEQ(A) presents the drawbacks of SEQ(C). The best performance results occurred when setting k to 10% of $|I|$, which is the value used henceforth.

Our next experiments consider the comparison of SEQ methods. We used data sets that were similar to the ones used in the previous experiment. 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 percent 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 user sequences 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., user sequences that actually satisfy the query) is the same for all methods. Therefore, the difference in the number of activated user sequences directly results from the difference in the number of false drops. The results are illustrated in Figure 8a (the vertical axis

Figure 7: Tuning of k

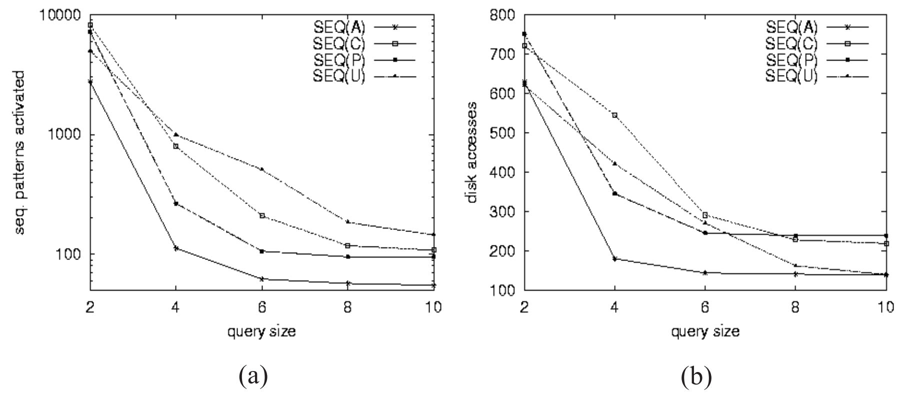


is in logarithmic scale). In all cases, SEQ(A) outperforms all other methods, indicating that its approximation technique is effective in reducing the cost to scan the data through the reduction of false drops.

Since the query performance depends both on the cost to scan the data and the index contents, for the case of the previous experiment, we measured the total number of disk accesses. The results are depicted in Figure 8b, with respect to the query size (i.e., the number of elements in the query sequence).

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, for these cases, cost to scan the data (fewer false drops, as also given in Figure 8a), and 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): (1) 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 cost to scan the data is introduced for all methods. This can be called a "pattern explosion" problem. (2) For large queries (with size comparable to *S*), there are not many different user sequences 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 are activated, and all methods have a very small and comparable cost to scan the data. Since, at these two extreme cases, both SEQ(P) and SEQ(U) have comparable Data Scan cost, SEQ(P) loses out due to the increased Index Scan cost incurred through the use of larger signatures (SEQ(U) does not use equivalent sets, thus, it uses 32-bit signatures;

Figure 8: Comparison of Methods: (a) Number of Activated User Sequences in the Database w.r.t. Query Size (b) Disk Accesses w.r.t. Query Size



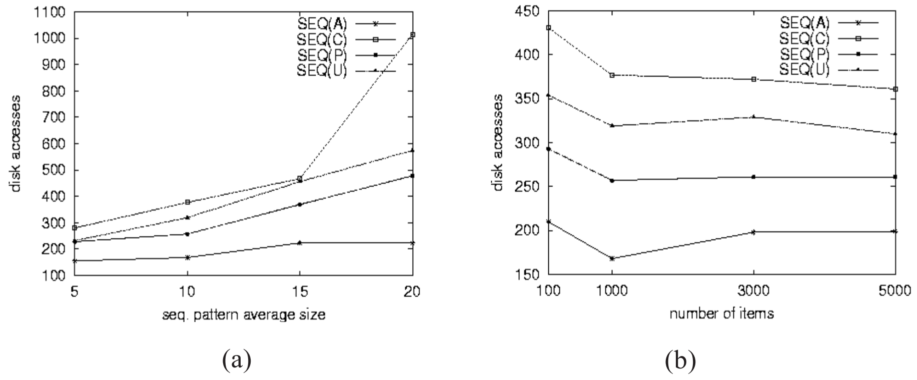
in contrast SEQ(P) uses for a 64-bit signature for each partition, thus, the total size is a multiple of 64-bits).

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 similarly 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 a much smaller number of disk accesses than SEQ(U).

Our next series of experiments examines the sensitivity of the methods. We first focus on the effect of *S*. We generated data sets, with the other parameters being the same as the previous experiments, and we varied the length *S* of sequences (the signature lengths were tuned against *S*). The resulting numbers of disk accesses are depicted in Figure 9a, for query size equal to *S*/2 in each case. Clearly, the disk accesses for all methods increase with increasing *S*. SEQ(A) performs better than all other methods, and it 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 five. The other parameters in the generated data sets were the same as those in previous experiments, and we varied *I*. The results are shown in Figure 9b. As shown, for all methods, very small values of |*I*| (e.g., 100) require a much larger

Figure 9: Effect of: (a) S (b) |I|



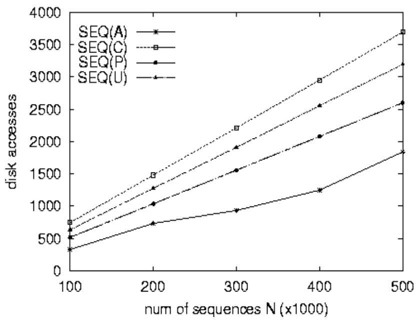
number of disk accesses. This is due to the larger impact of ordering, since more permutations of the same sets of items appear within sequences. SEQ(A) presents significantly improved performance compared to all other methods, whereas SEQ(P) comes second best.

Finally, we examined the scalability of the algorithms with respect to the number *N* of sequences. The rest parameters for the generated data sets were analogous to the ones in the experiments depicted in Figure 8, with varying *N*. The results are depicted in Figure 10a. 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, SEQ(C) presents the worst performance.

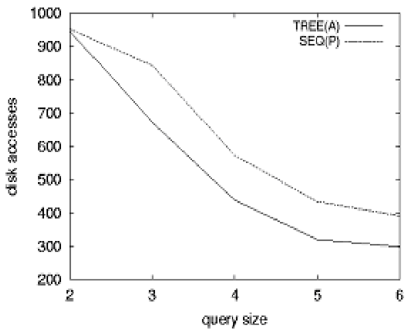
Results with Real Data

We now move on to examine the *ClarkNet* real web-log. Based on the previous results on synthetic data, we focus on the two most promising methods, namely (1) the partitioning, and (2) the approximation of equivalent sets. Moreover, we also considered the use of tree indexes for the signatures. Nevertheless, as described, the partitioning method (1) results in multiple signatures for each equivalent set, which can only be stored one after the other in a sequential file (i.e., SEQ(P) algorithm). In contrast, the approximation method (2) leads to only one signature for each equivalent set. Therefore, the latter signatures can be easily indexed with a tree structure for signatures, e.g., the S-tree (Deppisch, 1986). We used the approximation method and the improved S-tree variation proposed in Tousidou et al. (2000), and the resulting method is denoted as TREE(A) (i.e., approximation + a tree structure).

Figure 10: (a) Scalability w.r.t. Number of Sequences N. (b) I/O vs. Query Size for ClarkNet Web Log



(a)



(b)

The results for the comparison between SEQ(P) and TREE(A), with respect to query size (i.e., number of items in the query sequence), are given in Figure 10b. Evidently, TREE(A) significantly outperforms SEQ(P) in all cases. Only for very small queries (i.e., with two elements), do the methods present comparable performance, since a large number of the stored signatures and sequences are invoked by such queries (i.e., they have very low selectivity).

In summary, the approximation method has the advantage of allowing for a tree structure to index the signatures, which further reduces the cost to read the signatures. In combination with the reduced cost of scanning the data sequences due to lower number of false drops, the approximation method offers the best performance for large, real web logs.

FUTURE TRENDS

Analysis of the behavior of clients visiting a particular website is crucial for any companies or organizations providing services over the Internet. Understanding of clients' behavior is a key step in the process of improving the website. Nowadays, the information on how users navigate through a given web service is typically available in the form of web access logs. Knowing the limitations of web server logs, in the future, we may observe a tendency to log more accurate and complete information at the application server level. Nevertheless, after some preprocessing, the data to be analyzed by advanced tools (e.g., data mining tools) will have the form of a large collection of sequences stored in a company's database or data warehouse. A typical operation in the context of such data sets is searching for sequences containing a given subsequence. We believe that exploiting advanced indexing schemes, like those presented in this chapter, will be necessary to guarantee acceptable processing times.

Having this in mind, in the future, we plan to continue our research on sequence indexing, extending the most promising technique proposed so far, i.e., the method based on approximations of equivalent sets. We plan to examine alternative approximation schemes, such as: (1) global frequency threshold for ordered pairs of elements within a sequence (using most frequent pairs instead of most frequent successors for each item); and (2) information-content measures (considering only pairs that carry more information than others).

CONCLUSION

We considered the problem of efficient indexing of large web access logs for pattern queries. We discussed novel signature-encoding schemes, which are based on equivalent sets, to consider the ordering among the elements of access sequences, which is very important in the case of access logs. We have presented a family of indexing methods built upon the concept of equivalent sets. The performance of the proposed methods has been examined and compared, experimentally, with real and synthetic data. We tested the impact of query size, the tuning of the encoding schemes, and the scalability. These results illustrate the superiority of the proposed methods over existing indexing schemes for unordered data adapted to access sequences.

REFERENCES

Agrawal, R. & Srikant, R. (1995). Mining sequential patterns. In P. S. Yu & A. L. P. Chen (Eds.), *Proceedings of the 11th International Conference on Data Engineering* (pp. 3-14). Taipei, Taiwan: IEEE Computer Society.

Araujo, M. D., Navarro, G., & Ziviani, N. (1997). Large text searching allowing errors. In R. Baeza-Yates (Ed.), *Proceedings of the 4th South American Workshop on String Processing,* Valparaiso, Chile (pp. 2-20). Quebec, Canada: Carleton University Press.

Baeza-Yates, R. & Ribeiro-Neto, B. (1999). *Modern Information Retrieval*. New York: Addison-Wesley.

Bertino, E. & Kim, W. (1989). Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 196-214.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 422-426.

Chan, C. Y. & Ioannidis, Y. E. (1998). Bitmap index design and evaluation. In L. M. Haas & A. Tiwary (Eds.), *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data,* Seattle, Washington (pp. 355-366). New York: ACM Press.

Chen, M. S., Park, J. S., & Yu, P. S. (1998). Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2), 209-221.

Comer, D. (1979). The ubiquitous B-tree. *ACM Computing Surveys*, 11(2), 121-137.

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

Deppisch, U. (1986). S-tree: A dynamic balanced signature index for office retrieval. In *Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* Pisa, Italy (pp.77-87). New York: ACM Press.

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

Graefe, G. & Cole, R. L. (1995). Fast algorithms for universal quantification in large databases. *ACM Transactions on Database Systems*, 20(2), 187-236.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In B. Yormark (Ed.), *SIGMOD'84, Proceedings of Annual Meeting,* Boston, Massachusetts (pp. 47-57). New York: ACM Press.

Hellerstein, J. M. & Pfeffer, A. (1994). *The RD-tree: An index structure for sets.* Madison, WI: University of Wisconsin at Madison. (Technical Report 1252)

Helmer, S. (1997). *Index structures for databases containing data items with set-valued attributes*. Mannheim, Germany: Universität Mannheim. (Technical Report 2/97)

Helmer, S. & Moerkotte, G. (1997). Evaluation of main memory join algorithms for joins with set comparison join predicates. In M. Jarke, M. J. Carey, K. R. Dittrich, Fr. H. Lochovsky, P. Loucopoulos, & M. A.

Jeusfeld (Eds.), *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97),* Athens, Greece (pp. 386-395). San Francisco, CA: Morgan Kaufmann.

Helmer, S. & Moerkotte, G. (1999). *A study of four index structures for set-valued attributes of low cardinality*. Mannheim, Germany: Universität Mannheim. (Technical Report 2/99)

Ishikawa, Y., Kitagawa, H., & Ohbo, N. (1993). Evaluation of signature files as set access facilities in OOdbs. In P. Buneman & S. Jajodia (Eds.), *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington,DC* (pp. 247-256). New York: ACM Press.

Lou, W., Liu, G., Lu, H., & Yang, Q. (2002). Cut-and-pick transactions for proxy log mining. In C. S. Jensen et al. (Eds.), *Advances in Database Technology (EDBT 2002), 8th International Conference on Extending Database Technology,* Prague, Czech Republic, March 25-27, 2002(pp. 88-105). Berlin: Springer-Verlag.

Morzy, T. & Zakrzewicz, M. (1998). Group bitmap index: A structure for association rules retrieval. In R. Agrawal, P. E. Stolorz, & G. Piatetsky-Shapiro (Eds.), *Proceedings of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 284-288). New York: ACM Press.

Morzy, T., Wojciechowski, M., & Zakrzewicz, M. (2001). Optimizing pattern queries for web access logs. In A. Caplinskas & J. Eder (Eds.), *Advances in Databases and Information Systems,5th East European Conference* (ADBIS 2001), Vilnius, Lithuania, September 25-28,2001 (pp. 141-154). Berlin: Springer-Verlag.

Nanopoulos, A. & Manolopoulos, Y. (2002). Efficient similarity search for market basket data. *The VLDB Journal*, 11(2), 138-152.

Nanopoulos, A., Katsaros, D., & Manolopoulos, Y. (2003). A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, (forthcoming).

Nanopoulos, A., Zakrzewicz, M., Morzy, T., & Manolopoulos, Y. (2002). Indexing web access-logs for pattern queries. In *4th ACM CIKM International Workshop on Web Information and Data Management,* McLean, Virginia (pp. 63-68). New York: ACM Press.

Nanopoulos, A., Zakrzewicz, M., Morzy, T., & Manolopoulos, Y. (2003). Efficient storage and querying of sequential patterns in database systems. *Information and Software Technology*, 45(1), 23-34.

Nørvåg, K. (1999). Efficient use of signatures in object-oriented database systems. In J. Eder, I. Rozman, & T. Welzer (Eds.), *Advances in Databases and Information Systems, Proceedings of the 3rd East European Conference (ADBIS'99),* Maribor, Slovenia, September 13-16, 1999 (pp. 367-381). Berlin: Springer-Verlag.

Pei, J., Han, J., Mortazavi-Asl, B., & Zhu, H. (2000). Mining access patterns efficiently from web logs. In T. Terano, H. Liu, & A. L. P. Chen (Eds.), *Knowledge Discovery and Data Mining, Current Issues and New Applications, Proceedings of the 4th Pacific-Asia Conference (PAKDD 2000),* Kyoto, Japan, April 18-20, 2000 (pp. 396-407). Berlin: Springer-Verlag.

Spiliopoulou, M., & Faulstich, L. (1998). WUM - A tool for WWW ulitization analysis. In P. Atzeni, A. O. Mendelzon, & G. Mecca (Eds.), *The World Wide Web and Databases, International Workshop (WebDB'98),* Valencia, Spain, March 27-28, 1998, selected papers (pp. 184-203). Berlin: Springer-Verlag.

Tousidou, E., Nanopoulos, A., & Manolopoulos, Y. (2000). Improved methods for signature tree construction. *The Computer Journal*, 43(4), 301-314.

Zakrzewicz, M. (2001). Sequential index structure for content-based retrieval. In D. W.-L. Cheung, G. J. Williams, & Q. Li (Eds.), *Knowledge Discovery and Data Mining - PAKDD 2001, Proceedings of the 5th Pacific-Asia Conference,* Hong Kong, China, April 16-18, 2001 (pp. 306-311). Berlin: Springer-Verlag.

ENDNOTES

[1] This does not apply to SEQ(P) because it represents each equivalent set with several signatures.

[2] Clearly, after checking the equivalent sets, we have to examine the original sequences, so as to discard the false drops that are due to the use of equivalent sets. However, this step does not relate to the use of the S-tree, and is similar to the corresponding step in the SEQ methods.

[3] Available at the Internet Traffic Archive: http://ita.ee.lbl.gov/html/contrib/../traces.html.

[4] We have measured the performance of TREE(A) for these synthetic data and, as expected, we found that it outperforms SEQ(A) because of its reduced index-scan cost, which, however, is due to the tree index and independent from the approximation technique.