Maintaining Knowledge-Bases of Navigational Patterns from Streams of Navigational Sequences

Ajumobi Udechukwu, Ken Barker, Reda Alhajj
ADSA Lab, Dept. of Computer Science, University of Calgary, Canada
{ajumobiu, barker, alhajj}@cpsc.ucalgary.ca

Abstract

In this paper we explore an alternative design goal for navigational pattern discovery in stream environments. Instead of mining based on thresholds and returning the patterns that satisfy the specified threshold(s), we propose to mine without thresholds and return all identified patterns along with their support counts in a single pass. We utilize a sliding window to capture recent navigational sequences and propose a batch-update strategy for maintaining the patterns within a sliding window. Our batch-update strategy depends on the ability to efficiently mine the navigational patterns without support thresholds. To achieve this, we have designed an efficient algorithm for mining contiguous navigational patterns without support thresholds. Our experiments show that our algorithm outperforms the existing techniques for mining contiguous navigational patterns. Our experiments also show that the proposed batch-update strategy achieves considerable speed-ups compared to the existing window update strategy, which requires total re-computation of patterns within each new window.

Keywords: *Data streams, navigational patterns, web-usage mining.*

1. Introduction

An interesting problem in web usage mining that has attracted the attention of several researchers is the discovery of traversal patterns (or link navigation patterns) of web users [1]. Tracking user-browsing habits provides useful information for service providers and businesses, and ultimately should help to improve the effectiveness of the service provided. In popular e-commerce sites, the web logs receive continuous streams of entries. For these web sites to improve their performance by utilizing discovered navigational patterns, the navigational sequences should be treated as data streams. In this work, we propose a framework for mining and updating contiguous navigational patterns from streams of navigational sequences. We utilize a sliding window to capture the most recent set of navigational patterns.

The class of patterns identified from streaming web log sequences in this work is contiguous navigational patterns. We assume pre-processing steps are applied to the web logs [2] before they are added to the stream of navigational sequences. Generally, there are two broad techniques for mining navigational patterns – level-wise, apriori-based techniques [1]; and tree-based techniques [5, 7]. The apriori-based algorithms are derived from early algorithms for mining sequential patterns and association rules. These algorithms are level-wise and utilize candidate generation and test techniques so it is possible to define various test conditions for candidate patterns before they are included in the result-set. These algorithms can be used to discover different types of navigational patterns by adjusting the test conditions, so they can be used for mining generalized navigational patterns and for constrained navigational patterns. For example, given the navigational sequence (A, C, K, M, O, R) representing objects (or web pages) visited by a user in order. A generalized navigational pattern would include “A, M, R” as a valid pattern because the objects are visited in order. The test condition may also be set to constrain gaps between successive objects. If the maximum allowable gap between two consecutive objects in a pattern is 1, then pattern “A, M, R” becomes invalid because the distance between ‘A’ and ‘M’ is 2. When such constraints are incorporated in the mining process the result is known as a constrained navigational pattern. A special case of constrained navigational patterns is contiguous navigational patterns, which results when the allowable gap between successive objects is set to 0 whereby no gap is allowed. The apriori-based algorithms can discover every type of navigational pattern discussed above.

1097-8585/05 $20.00 © 2005 **IEEE**

However, the candidate generation and test technique utilized in apriori-based algorithms is computationally expensive.

The development of tree-based techniques for association rule mining motivates the use of tree-based techniques for navigational pattern discovery [7]. Unfortunately, the flexibility enjoyed in apriori-based techniques for navigational pattern mining is not transferable to their tree-based counterparts. The tree-based techniques found in the literature identify all generalized paths but cannot differentiate constrained navigational patterns (including contiguous patterns) [5, 7]. Note that the set of constrained navigational patterns are included in the set of generalized navigational patterns. However, some application scenarios are better served by constrained or contiguous patterns. In such cases, the existing tree-based algorithms cannot be used because they cannot differentiate between the different types of navigational patterns. For example, Nakagawa and Mobasher [6] show that recommendation systems based on contiguous sequential patterns perform better than those based on association or generalized sequential patterns in websites with low link connectivity, deep traversal depths, or dynamically generated content. These results support earlier results that advocate the use of constrained patterns for web page prediction [9]. Our work contributes an efficient tree-based algorithm for mining contiguous navigational patterns that outperforms existing algorithms for discovering contiguous navigational patterns. We also contribute a novel *batch-update* strategy for maintaining current patterns from a stream of navigational sequences.

The paper is organized as follows. Section 2 presents related work. Section 3 presents our methodology for discovering and maintaining current navigational patterns within a sliding window. We present and discuss our experimental results in Section 4. Conclusions and areas of future work are discussed in Section 5.

2. Related work

Chen *et al.* [1] propose algorithms for mining maximal forward references from navigational patterns. The authors also propose apriori-based algorithms for mining frequent, contiguous navigational patterns from the set of maximal forward references. Our work also involves mining contiguous navigational patterns but we utilize a more efficient tree-based algorithm for this purpose. We are interested in streams of navigational sequences but not static collections of sequences. Our approach also does not require pre-defined support thresholds.

Pei *et al.* [7] propose a tree-based technique for mining web navigational patterns termed the WAP-tree. The authors show that the WAP-tree outperforms the earlier apriori-based techniques. However, the WAP-tree and its variants (e.g., the PLWAP-tree [5]) cannot differentiate all the different types of navigational patterns that can be mined with the apriori-based algorithms. The WAP-tree and its variants are also driven by support thresholds. Our work differs significantly from the WAP-tree and its variants as we do not address the same problem. Our approach is directed at mining contiguous navigational patterns and is not driven by support thresholds. Our work is also tailored for environments with streaming navigational sequences.

Giannella *et al.* [3] propose a framework for mining frequent association patterns from data streams. Their work is similar in spirit to the work in this paper. However, we are interested in mining constrained navigational patterns rather than association patterns. We also propose an efficient incremental pattern-update strategy within sliding windows, which differs from Giannella *et al.*'s total re-computation strategy [3].

3. Methodology

Our aim is to mine and maintain contiguous navigational patterns from input streams of navigational sequences. We use a windowing technique to capture the current navigational sequences. The input stream of navigational sequences is continuous but the navigational sequences are treated in batches. For example, if the base time unit utilized is one minute, then all the navigational sequences obtained from the web log for each minute are processed together as a batch. The window is used to determine which batches participate in maintaining the set of current contiguous navigational patterns. Several windowing techniques exist. For example, given that the window size is four batches and the following batches of navigational sequences arrive in this order: B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, & B12. We can have a windowing strategy that processes batches B1 to B4 in one window and then batches B5 to B8 next. This would mean that after processing batch B4 the system waits to accumulate batches B5 to B8 before continuing. This approach does not support the aim of real time processing in data stream environments. An alternative windowing approach would be to have a sliding window that gets

1097-8585/05 $20.00 © 2005 IEEE

updated with the arrival of each batch. Using our example, that means that batches B1 to B4 are processed in the first instance. When batch B5 arrives, the window becomes batches B2 to B5. This presents different challenges because we must remove pattern counts arising from batch B1 and add those from batch B5. This can be done in two ways:

- Start a new mining process for each new window and accumulate the patterns for the navigational sequences present in that window.
- Design efficient techniques for removing the impact of the batch leaving the window while adding the new batch without starting afresh.

Intuitively, the second approach is preferable if the window contains several batches so it would be cheaper to remove the impact of one batch on the discovered patterns than starting the entire process from scratch. Before we discuss our strategies for maintaining patterns within a window, we first discuss our algorithm for mining contiguous navigational patterns from a batch of navigational sequences.

3.1. Mining contiguous navigational patterns

The aim here is to find objects (or web pages) that are accessed together and the navigational patterns frequently exhibited by system users. However, unlike most algorithms, our technique is not based on predefined support (or another threshold). We reported an algorithm for mining contiguous navigational patterns based on an *adapted generalized suffix tree (GST)* that does not require support thresholds earlier [8]. However, the *adapted GST* algorithm cannot be used efficiently with the window maintenance scheme discussed in Section 3.2. We propose a less compact data structure termed the AC-NAP tree (**A**ll **C**ontiguous – **N**avigational **A**ccess **P**attern tree) for mining the contiguous patterns without pre-defined support thresholds.

The AC-NAP tree is a general (not necessarily balanced) tree with the following characteristics:

1. Each node except the root node has a symbol, a label, a count, and a last-id property. The symbol indicates the object/webpage represented by the node. The count accumulates the number of unique navigational sequences that have reached or traversed the node. The last-id holds the identifier of the last sequence to reach or traverse the node.
2. The concatenation of node symbols from the root node to any internal or leaf node constitutes the node's label. Each node label represents a navigational path with a support count equal to the node's count property.
3. Each node can have an arbitrary number of children, but no two paths from any node are identical so no two child-nodes of any node can have the same symbol.

The AC-NAP tree construction algorithm takes the set of navigational sequences with unique identifiers defined for each sequence as input. The objects (or web pages) in the system are allocated unique symbols and these are used in generating the navigational sequences. The steps involved in constructing the AC-NAP tree are:

1. Starting with the first navigational sequence, extract the suffixes of the sequence from position 1 to position n, where n is the length of the sequence. For example, given that 'L', 'Q', & 'R' are symbols used to represent objects (or web pages) in the system, and that "LQR" is the first navigational sequence to be added to the AC-NAP tree. The suffixes of the navigational sequence are "LQR" for position 1; "QR" for suffix starting at position 2; and "R" for suffix starting at position 3. When the AC-NAP tree is created, it is given a root node. When the first suffix is added to the tree, a child node is created for each item in the suffix. Each node in the tree has a node symbol – which is the object (or web page) represented at that node; a node label – which is the path from the root to that node (i.e., the concatenation of all the node symbols from the root to that node, inclusive of the node's symbol); a node count – which accumulates the number of unique navigational sequences that have reached the node; and a last-id property – which holds the identifier of the last suffix that reached that node. Note that suffixes inherit the unique identifiers of their parent navigational sequences. Figure 1 shows the steps in creating the base AC-NAP tree with the suffixes of the navigational sequence "LQR". We have numbered the nodes for ease of discussion. The shaded oval is the root node while the unshaded ovals are the other nodes of the tree. Each suffix of a sequence is added to the tree starting at the root node. The suffix being added follows the path of the child node whose symbol is the same as the next item in the suffix to be added. If there is no child node from the current node that has the next item on the suffix as a node symbol, a new child node is created from the current node and the symbol is assigned as the next item on the suffix being added. Thus, no two child-nodes from the same parent node may have identical symbols. A node label is assigned to a node at creation, for example in Figure 1, the node label

1097-8585/05 $20.00 © 2005 **IEEE**

for node 1 is "L"; the label for node 3 is "LQR"; the label for node 5 is "QR", *etc*. At creation, the last-id property of a new node is the same as the identifier of the navigational sequence from which the suffix is derived. The node-count property of a new node in the AC-NAP tree is set to 1.

2. The remaining sequences from the set of navigational sequences are added one at a time to the AC-NAP tree. For each sequence, all its suffixes are extracted and added to the tree in order. Each suffix is added to the tree by traversing the nodes starting at the root node (similar to the process used in constructing the base tree). However, in this case, when a suffix reaches a node, if the identifier of the suffix is different from the last-id property of the node, the node count property of the node is incremented by 1 and the last-id property is updated to the identifier of the suffix that just traversed (or reached) by the node.
3. Finally, output all node labels and counts to a database, adding a unique identifier for each pattern. The node labels are the patterns and the node counts are the support counts for the respective patterns.

Figure 2 shows an AC-NAP tree for the navigational sequences "LQR", "LQR", "LQ", and "QR". The representation of these navigational sequences using the adapted GST algorithm is shown in Figure 3.

The *adapted GST* accumulates patterns along edges so a stop signal ($) is included to ensure that all suffixes end at leaf nodes. The root node in Figure 3 is the shaded oval, the internal nodes are the unshaded ovals, and the rectangles represent the leaf nodes. Leaf nodes are handled differently than internal nodes in the *adapted GST* technique. (Udechukwu *et al.* [8] provide a full discussion on the *adapted GST* algorithm.) There are two internal nodes in Figure 3, with labels "LQ" and "Q", respectively. The node with label "LQ" is traversed by suffixes from three sequences so pattern "LQ" has a count of 3. Similarly, pattern "Q" has a count of 4 because its node is traversed by four sequences. Pattern "Q" from Figure 3 is equivalent to node 4 in the AC-NAP tree of Figure 2. Similarly, pattern "LQ" of Figure 3 is the equivalent of node 2 in Figure 2. The difference between these two representations is the AC-NAP tree reports all patterns while the *adapted GST* algorithm collapses some patterns that can be inferred from other patterns. For example, both patterns "L" with support count of 3 and pattern "LQ" with support count of 3 are reported in the AC-NAP tree but in the *adapted GST* representation only pattern "LQ" with support count of 3 is reported because pattern "L" does not occur independently and can be inferred. Collapsing some inferable patterns (as is the case with the *adapted GST* algorithm) reduces the output space, which is a laudable attribute in some data mining scenarios. However, when mining patterns independently across different windows, as is the case in this paper, it is challenging to identify and update subparts of the collapsed patterns. For example, pattern "LQ" may be found in batch B1 with support count 4 using the *adapted GST* algorithm. Given that there was no independent occurrence of pattern "L" it can only be inferred from "LQ". Assuming that in batch B2 pattern "L" occurs independently with a count of 2, it now becomes challenging to obtain the total count of "L" across the two batches because this would require searching through the patterns from batch B1 to extract the highest count for "L". Fortunately, this challenge is addressed in the AC-NAP tree.

3.2. Maintaining patterns within a window

Recall that we are working with streams of navigational sequences. The navigational sequences are grouped together into batches representing sequences that arrived within a base time unit. A sliding window is used to specify which batches participate in reporting the current set of navigational patterns. Consider the example used earlier in Section 3. Given that the sliding window size is four batches and the following batches of navigational sequences arrived in order: B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, & B12. Our methodology mines the navigational sequences in each batch independently. Thus, when batch B1 arrives, the navigational patterns in B1 are mined and stored in a database table with the following schema:

Current_patterns (pattern-id, batch-id, pattern-name, support-count)

1097-8585/05 $20.00 © 2005 **IEEE**

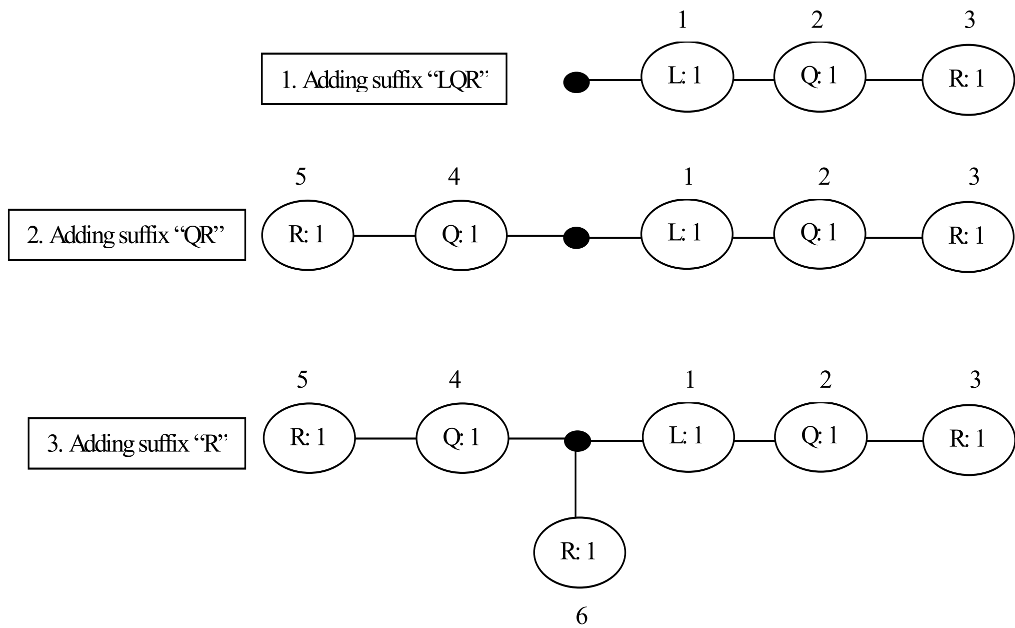

Figure 1. Creating the base AC-NAP tree using the navigational sequence "LQR"

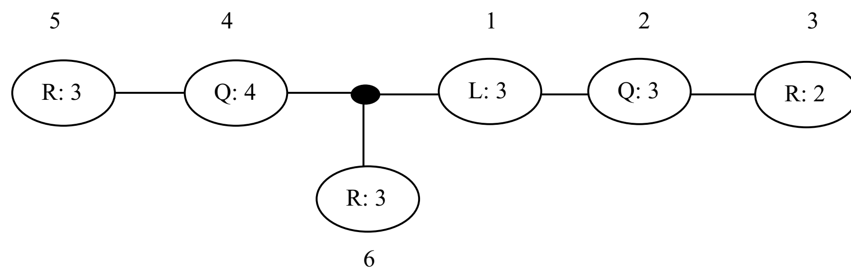

Figure 2. AC-NAP Tree for the navigational sequences "LQR", "LQR", "LQ" and "QR"

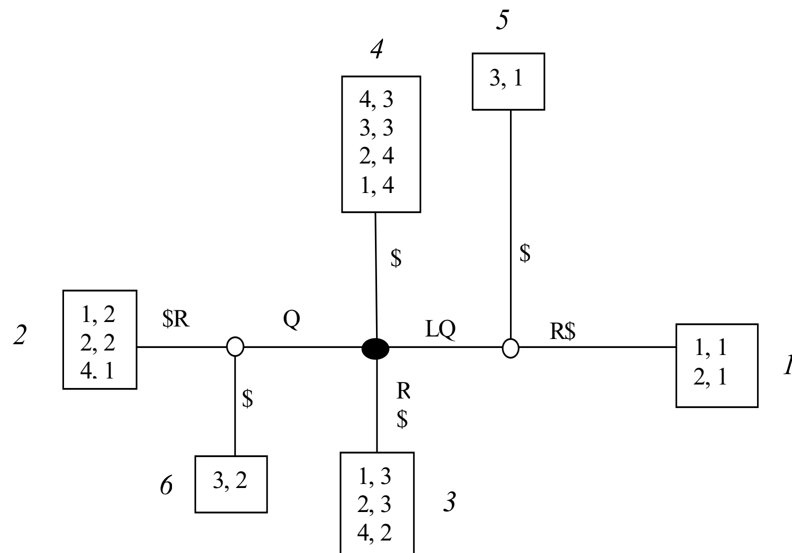

Figure 3. Adapted GST with leaf-node relevance for the navigational sequences "LQR", "LQR", "LQ" and "QR" with "$" appended to each sequence as a stop symbol [8]

We utilize a memory-based DBMS table for this purpose. Our examples in this section are based on memory-based DBMS concepts from the MySQL[1] DBMS. The Current_patterns table within MySQL is created in main memory using the following SQL statement:

```
CREATE TABLE Current_patterns
        (pattern-id INT, batch-id INT, pattern-name
        VARCHAR, support-count INT, PRIMARY
        KEY (pattern-id))
    ENGINE = MEMORY;
```

Similarly, the navigational patterns from batch B2 are mined and added to the table. The batches are mined independently and added to the table until the window size is reached (in this case, until batch B4 is processed). Table 1 shows a sample of some navigational patterns for the first window.

Table 1. A sample representation of patterns in a knowledgebase table

Pattern-id	Batch-id	Pattern-name	Support-count
1	1	LQR	20
2	1	L	50
3	1	QR	30
4	2	QLR	40
5	2	QR	5
6	3	L	15
7	3	QR	20
8	4	QR	10
9	4	LQR	5

Notice that some patterns repeat in different batches. For example: pattern "QR" occurs in all the four batches with support-counts 30 for batch 1 (i.e., B1); 5 for batch B2; 20 for batch B3; and 10 for batch B4. To obtain the unique navigational patterns within a window and their corresponding occurrence frequencies, the following SQL statement is used:

```
SELECT pattern-name, SUM (support-count)
FROM Current_patterns
GROUP BY pattern-name;
```

[1] http://www.mysql.com

At the arrival of the next batch of navigational sequences (i.e. batch B5), the sliding window shifts its start position to batch B2. This means that patterns from batch 1 (i.e., B1) must be removed from the Current_patterns table. This is achieved using the following SQL statement:

```
DELETE * FROM Current_patterns
    WHERE batch-id = 1;
```

The new navigational patterns discovered from batch B5 are then added to the Current_patterns table and the entire process repeats. We call our approach for maintaining and updating patterns within a sliding window the *batch-update* strategy. An alternative strategy used in previous work [3] is to re-compute the entire set of patterns for all the batches within the window for each update. Our *batch-update* strategy follows a simple intuition: since the patterns within a window are processed using a sliding-window protocol that involves incrementally adding and removing sequences from the window, it makes sense to incrementally update the resulting navigational patterns and knowledgebase in a similar fashion. Thus, the essence of our *batch-update* strategy is to remove the impact of navigational sequences that are no longer in the sliding window from the knowledgebase while adding the impact of the new sequences that just arrived at the window. This update process must be achieved correctly in the most efficient way. It is important to note that the simplicity of update enjoyed in the *batch-update* strategy is only possible when the patterns in the individual batches are discovered without support thresholds.

4. Experimental results and discussions

The major focus of our experiments is the evaluation of the batch-update strategy proposed in this paper. All the experiments reported here were executed on a personal computer running Microsoft Windows XP professional edition, with a 2GHz Intel Pentium 4 processor, and 512 MB RAM. The algorithms are implemented in Java (in the JBuilder 8 personal edition IDE) and the knowledgebase was stored in a MySQL DBMS. Java database connectivity is used for communication between the programs and the knowledgebase. We tested the update strategies (i.e., batch update vs. total re-computation) with publicly available, anonymous, real-world web logs of "msnbc.com" users from the UCI repository [4]. We chose to ignore single-valued navigational sequences (i.e., sequences consisting of a single page visit) as

they do not capture interesting link information. We also ignored sequences with 50 events or longer, as they were more likely to be generated by web crawlers (given that the average sequence length in the original dataset is 5.7 page visits/events). For our experimental evaluations, we set the batch size to 1,000 navigational sequences, and use window sizes of 5, 10, and 20 batches per window. For each window size, we ran 10 updates to evaluate the average performance of the different update strategies. Our results are reported in Tables 2, 3, and 4 for the three window sizes, respectively.

We compare the performance of the two update-strategies (i.e., batch-update vs. total re-computation), using both the apriori-based algorithm [1] and the AC-NAP tree for mining contiguous navigational patterns within each window. (Note: the other tree-based algorithms for discovering navigational patterns [5, 7] cannot differentiate contiguous navigational patterns so we do not include them in our evaluations.) We do not report any execution times for our batch-update strategy with apriori because apriori-based algorithms are extremely expensive if support thresholds are removed and our batch-update strategy requires the removal of support thresholds. The execution times reported in Tables 2 – 4 are the total times required to mine the patterns within each new window and update the knowledgebase. (The knowledgebase is utilized in applications that are based on navigational patterns, such as web recommendation systems, page pre-fetching/prediction, *etc.*). In our implementation, we also use a stored procedure that transfers all patterns arriving at the memory-based table to a regular disk-based table such that a historical trail of patterns is maintained.

The results of our experiments show that our batch-update strategy runs in (near) constant time, irrespective of the window size (i.e., given that we have the same number of navigational sequences in each batch, as is the case in our experimental setup). The results follow our intuitive reasoning because our batch-update strategy only computes the patterns within the new batch entering the window and re-uses the patterns already discovered in the batches previously in the window. Furthermore, the storage and representation scheme used to discover patterns means the final step of removing the impact of patterns from the batch exiting the window is also achieved at a minimal cost.

5. Conclusions

This paper proposes an efficient framework for discovering and maintaining navigational patterns from streaming navigational sequences. Similar to previous work [3], we utilize a sliding window to track the set of current navigational patterns from the data stream. We utilize a sliding *batch-update* strategy for maintaining the patterns within the sliding window. This differs from the earlier approach that required a total re-computation of patterns within windows. However, our methodology requires efficient pattern-discovery techniques that are not based on support thresholds. To achieve this we propose the AC-NAP tree (**A**ll **C**ontiguous – **N**avigational **A**ccess **P**attern tree) for mining contiguous navigational patterns without support thresholds. Our experiments show that our *batch-update* strategy is practical and achieves significant speed-ups important in streaming environments. The path-mining algorithm presented in this work discovers contiguous navigational patterns. An area of future work is to develop a path-mining algorithm for the general case of constrained navigational paths that would not utilize pre-defined support thresholds.

6. References

[1] Chen, M-S., Park, J.S., and Yu, P.S., "Efficient Data Mining for Path Traversal Patterns", *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 2, 1998, pp. 209-221.

[2] Cooley, R., Mobasher, B., and Srivastava, J., "Data Preparation for Mining World Wide Web Browsing Patterns", *Knowledge and Information Systems*, vol. 1, no. 1, 1999.

[3] Giannella, C., Han, J., Pei, J., Yan, X., and Yu, P.S., "Mining Frequent Patterns in Data Streams at Multiple Time Granularities", in H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, (Editors), Chapter 3, Data Mining: Next Generation Challenges and Future Directions, AAAI/MIT Press, 2003.

[4] Hettich, S. and Bay, S.D, "The UCI KDD archive", [http://kdd.ics.uci.edu]. Department of Information and Computer Science, University of California, Irvine, CA, 1999.

[5] Lu, Y. and Ezeife, C.I., "Position Coded Pre-Order Linked WAP-Tree for Web Log Sequential Pattern Mining", *Proc. of the 7th Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, 2003, pp. 337-349.

Table 2. Update Times (in seconds) for ten windows using a window size of 5

	Window										
	1	2	3	4	5	6	7	8	9	10	Average
Batch-update with AC-NAP	2.0	2.2	1.9	1.9	2.1	2.0	2.0	2.1	2.1	2.2	2.05
Total window re-computation with AC-NAP	5.5	5.5	5.7	5.9	5.6	5.7	5.4	6.0	5.5	5.6	5.64
Total window re-computation with Apriori at 0.1% support	19.9	19.7	20.0	19.5	19.8	19.8	19.4	20.1	20.8	20.8	19.98

Table 3. Update Times (in seconds) for ten windows using a window size of 10

	Window										
	1	2	3	4	5	6	7	8	9	10	Average
Batch-update with AC-NAP	1.9	2.0	2.0	2.0	2.1	1.9	1.7	1.9	1.9	1.9	1.93
Total window re-computation with AC-NAP	9.3	9.5	9.5	9.5	9.8	9.6	9.6	9.6	9.0	9.6	9.50
Total window re-computation with Apriori at 0.1% support	36.8	36.3	36.9	37.0	37.4	37.2	36.7	36.8	37.0	37.0	36.91

Table 4. Update Times (in seconds) for ten windows using a window size of 20

	Window										
	1	2	3	4	5	6	7	8	9	10	Average
Batch-update with AC-NAP	1.8	2.0	1.8	2.0	1.8	2.1	2.1	1.9	1.8	2.0	1.93
Total window re-computation with AC-NAP	15.1	15.7	16.0	16.6	15.5	15.7	15.2	15.1	15.7	15.4	15.60
Total window re-computation with Apriori at 0.1% support	69.6	70.0	71.8	70.8	70.4	71.2	70.0	71.0	71.3	72.5	70.86

[6] Nakagawa, M. and Mobasher, B., "A Hybrid Web Personalization Model Based on Site Connectivity", *Proc. of the 5th WEDKDD Workshop*, in conjunction with ACM SIGKDD Conference, Washington DC, USA, August 2003, pp. 59-70.

[7] Pei, J., Han, J., Mortazavi-asl, B., and Zhu, H., "Mining Access Patterns Efficiently from Web Logs", in *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00)*, 2000, pp. 396-407.

[8] Udechukwu, A., Barker, K. and Alhajj, R., "A Framework for Representing Navigational Patterns as Full Temporal Objects", *ACM SIGEcom Exchanges*, vol. 5, no.2, 2004.

[9] Yang, H. and Parthasarathy, S., "On the Use of Constrained Associations for Web Log Mining", *Proc. of the International Workshop on Knowledge Discovery in the Web (WEBKDD)*, 2002.