

Mining Sequential Patterns Using I-PrefixSpan

Dhany Saputra, Dayang R. A. Rambli, Oi Mean Foong

Abstract—In this paper, we propose an improvement of pattern growth-based PrefixSpan algorithm, called I-PrefixSpan. The general idea of I-PrefixSpan is to use sufficient data structure for Seq-Tree framework and separator database to reduce the execution time and memory usage. Thus, with I-PrefixSpan there is no in-memory database stored after index set is constructed. The experimental result shows that using Java 2, this method improves the speed of PrefixSpan up to almost two orders of magnitude as well as the memory usage to more than one order of magnitude.

Keywords—ArrayList, ArrayIntList, minimum support, sequence database, sequential patterns.

I. INTRODUCTION

THE objective of sequential pattern mining is to find all frequent sequential patterns with a user-specified minimum support. The problem of sequential patterns discovery was inspired by retailing industry problems. However, the results apply to many scientific and business domains, such as stocks and markets basket analysis, natural disasters (e.g. earthquakes), DNA sequence analyses, gene structure analyses, web log click stream analyses, and so forth. Suppose the computer shop database contains list of customers, list of transaction time of each customer, and list of items bought for each customer's transaction time, then "80% of customers buy computer, then earphone, and then digital camera" may be one of the valuable sequential patterns found. This purchasing need not be consecutive. Those customers who buy some other items in between also support this pattern.

Generally, the sequential pattern mining approaches are either generate-and-test (also known as apriori) or pattern growth (also known as divide-and-conquer) or vertical format method approach. *AprioriAll* [1] is the earliest sequential pattern mining algorithm proposed by Agrawal and Srikant in 1995. The following year they generalized the problem to include time constraints, sliding time window, and user-defined taxonomy, and presented an apriori-based, improved algorithm *GSP* [2]. Either *GSP* or *AprioriAll* adopt a multiple-pass, candidate generation-and-testing approach. This approach has three drawbacks. Firstly, a very large set of candidate sequences could be generated in a large database. Secondly, this approach must scan the database many times. Lastly, it generates a combinatorially explosive number of candidates when mining longer sequential patterns. Hence, this approach may not be sufficient in mining large sequence databases having numerous and/or long patterns.

FreeSpan [3] and PrefixSpan [4], [5] are two algorithms adopting divide and conquer approach. In this approach, the sequence database are recursively projected into a set of smaller projected database based on the current frequent

pattern(s), and sequential patterns are grown in each projected databases by exploring only locally frequent fragments [3]. To improve the performance, FreeSpan offers bi-level projection technique, which scans the database twice, instead of level-by-level projection. It has also been showed that FreeSpan runs considerably faster than *GSP*. The following year after the proposal of FreeSpan, Jian Pei et al [4] proposed PrefixSpan algorithm. Since this algorithm projects only the suffixes, the size of the projected database shrinks and redundant checking in every possible location of a potential candidate is reduced. It offers pseudo projection technique that avoids physical projections and maintains the suffix by means of a pointer-offset pair, instead of bi-level projection. In addition, PrefixSpan with pseudo projection outperformed FreeSpan with bi-level projection. Note that pseudo projection does not work effectively on FreeSpan since the next step can be in both forward and backward directions.

Beside the two horizontal layout approaches mentioned earlier, the database can also be transformed into vertical format consisting of items' id-lists. Mohammed Zaki proposed SPADE [6], which follows this approach. It scans the database three times. However, the transformation into vertical format requires both time and more memory space. Although SPADE outperforms FreeSpan, the newer publication of PrefixSpan stated that PrefixSpan still performs better than both of them [4].

A memory-indexing approach for fast discovery of sequential patterns, named MEMISP [7] was introduced in 2005. With MEMISP, there is neither candidate generation nor database projection. Both CPU and memory utilization are low. Through index advancement within an index set composed of pointers and position indices to data sequences, MEMISP discovers sequential patterns by using a recursive find-then-index technique. Through partition-and-validation technique of MEMISP, one can still mine an extremely large database, which is too large to fit into main memory, in two or more database scans. In order to facilitate fast index construction and speed up searching from specific positions, MEMISP uses a variable-length array to store the in-memory data sequence. Nevertheless, if variable-length array is used to store the items bought and transaction time only, then one loses the list of customer. Yet, it is impossible to use 2D or 3D variable-length array, since one cannot create an array with non-fix size of either dimension. A specific framework is required to be proposed to store index set and in-memory database.

In this paper, we introduce an improved version of PrefixSpan named I-PrefixSpan. This algorithm improves PrefixSpan in two ways: (1) it implements sufficient data structure for Seq-Tree framework to build the in-memory database sequence and to construct the index set, and (2) instead of keeping the whole in-memory database, it

implements separator database to store the transaction alteration signs.

The remainder of the paper is organized as follows. The sequential pattern mining terms are explained and PrefixSpan algorithm is illustrated in Section II. Section III presents the I-PrefixSpan algorithm. The experimental results and analysis of the proposed I-PrefixSpan algorithm are described in Section IV. We conclude the study in Section V.

II. DEFINITION OF TERMS AND THE PREFIXSPAN ALGORITHM

A. Terms Definition

Let $I = \{i_1, i_2, \dots, i_n\}$ be a set of all **items**. **Itemset** is a non-empty subset of I . A **sequence** is an ordered list of itemsets. A sequence s is denoted by $\langle s_1 s_2 \dots s_l \rangle$, where s_j is an *itemset*. s_j is also called an **element** of the sequence, and denoted as $(x_1 x_2 \dots x_m)$, where x_k is an item. If an element has only one item, the brackets can be omitted. For example, element (x) is written as x . An item can occur at most once in an element of a sequence, but it can occur multiple times in different elements of a sequence. The number of items in a sequence is called **the length of a sequence**. A sequence $\alpha = \langle a_1 a_2 \dots a_n \rangle$ is called a subsequence of another sequence $\beta = \langle b_1 b_2 \dots b_m \rangle$ and β is a supersequence of α , denoted as $\alpha \sqsubseteq \beta$, if there exists integers $1 \leq j_1 < j_2 < \dots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \dots, a_n \subseteq b_{j_n}$.

A **sequence database** S is a set of tuples $\langle sid, s \rangle$ where sid is a **sequence id** and s is a **sequence**. A tuple $\langle sid, s \rangle$ is said to **contain** sequence α , if α is a subsequence of s . The **support** of a sequence α in a sequence database S is defined as the number of tuples in the database containing α , denoted as:

$$\text{support}(\alpha) = \left| \left\{ \langle sid, s \rangle \mid \langle sid, s \rangle \in S \wedge (\alpha \sqsubseteq s) \right\} \right| \quad (1)$$

Given a positive integer $min_support$ as the support threshold, a sequence α is called a **sequential pattern** for a **large sequence** in a sequence database S if $support(\alpha) \geq min_support$.

Sequential pattern mining is defined as finding complete set of sequential patterns in the sequential database, given $min_support$ threshold.

B. The PrefixSpan

As mentioned in the analysis of FreeSpan algorithm in [4], one may consider two pitfalls of implementing FreeSpan: (1) redundant checking at every possible position of a potential candidate sequence and (2) the large size of projected database. To avoid the former pitfall, items within an itemset must first be ordered. We can assume that they are always ordered alphabetically without loss of generality. To avoid the

latter pitfall, projection can be done just by following the order of the prefix of a sequence and project only the suffix.

The algorithm of PrefixSpan is as follows:

Algorithm PrefixSpan

Input : A sequence database S , the minimum support threshold $min_support$

Output : The complete set of sequential patterns

Metode : Call $PrefixSpan(\langle \rangle, 0, S)$

procedure $PrefixSpan(\alpha, L, S|_{\alpha})$

- 1) Scan $S|_{\alpha}$ once, find each frequent item b , such that:
 - a) b can be assembled to the last element of α to form a sequential pattern; or
 - b) $\langle b \rangle$ can be appended to α to form a sequential pattern.
- 2) For each frequent item b , append it to α to form a sequential pattern α' and output α' .
- 3) For each α' , construct α' -projected database $S|_{\alpha'}$.
- 4) Call $PrefixSpan(\alpha', L+1, S|_{\alpha'})$

Instead of generating intermediate projected databases, we create the index set by registering all the position index of the associated customer by means of a (**pointer, offset**) pair, where **pointer** is a pointer to the corresponding sequence and **offset** represent the positions of the projected suffix in the sequence. Offset should be an integer, if there is a single projection point; and a set of integers, if there are multiple projection points. Each offset indicate the starting projection position in a sequence.

It is reported that PrefixSpan and MEMISP are different although they both utilize memory for fast computation [7]. Yet, PrefixSpan with pseudo projection technique seems to work in a similar way with MEMISP, except when the database cannot be held in main memory. For this exceptional case, MEMISP uses partition-and-validation technique which scans database more than once, depends on the memory available and the size of the sequence database.

III. I-PREFIXSPAN

A. Seq-Tree Framework

There are no documentations found for any framework to store the in-memory database and index set, except in MEMISP [7]. It is said that MEMISP uses variable-length arrays to hold the data sequences in memory. Nevertheless, if variable-length array is used to store the items bought and

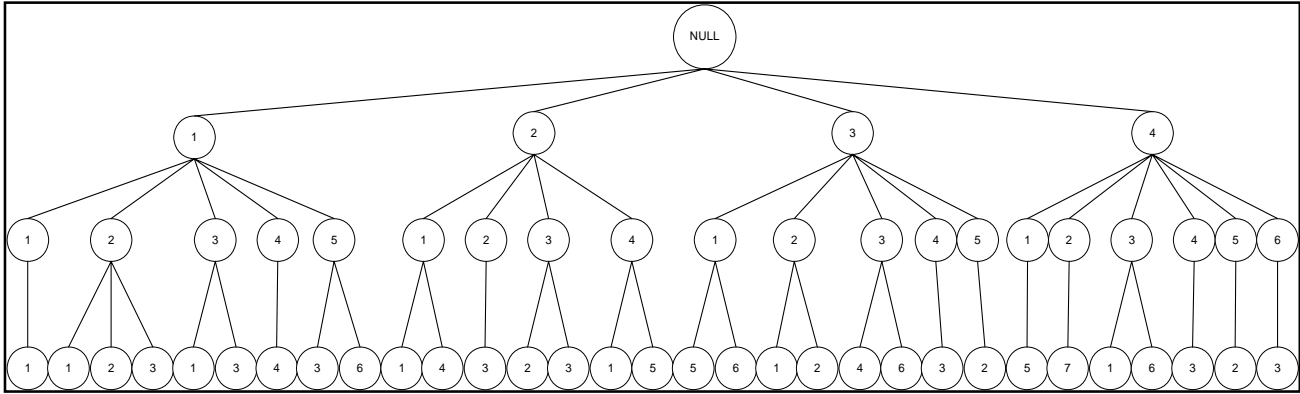


Fig. 1 Seq-Tree Framework

transaction time only, then the list of customers will be missing. Still, it is impossible to use two-dimensional or three-dimensional variable-length array, since it is not possible to create an array with non-fix size of either dimension. Seq-Tree framework with sufficient data structure is the other contribution in I-PrefixSpan which is used to store in-memory sequence database and to construct the index set. Seq-Tree is a general tree with two certain characteristics: (1) all leaves must be located at the same depth and (2) the height of the tree is at least 2.

Definition 1 (Array of Items Bought, Array of Transaction Time, Array of Customers)

Let $T_{c,t} = \{i_1, i_2, \dots, i_n\}$ be the list of all items by a customer on one transaction time of an in-memory sequence database, where c is the customer id of the associated customer, t is the transaction id of the associated transaction time of this customer c on transaction time t . $T_{c,t}$ is stored in a variable-length array named **array of items bought**.

Let $C_c = \{T_{c,1}, T_{c,2}, \dots, T_{c,m}\}$ be the list of all transaction time of a customer with customer id c of the same in-memory sequence database, where m is the number of transaction ids of customer c . T_c is stored in a variable-length array named **array of transaction time**.

Let $A = \{C_1, C_2, \dots, C_p\}$ be the list of all customers, where p is the number of customers in the associated sequence database. A is stored in a variable-length array named **array of customer**.

Definition 2 (Array of Offset, Array of Pointers, Array of Prefixes)

Let $P_{pr,pt} = \{o_1, o_2, \dots, o_n\}$ be the list of all offsets by a prefix on one pointer of an index set, where pr is the 1-sequence (prefix) to be mined, pt is the pointer of this prefix, and n is the number of offset for prefix pr of pointer pt . $P_{pr,pt}$ is stored in a variable-length array named **array of items offset**. Let $Pr_{pr} = \{Pt_{pr,1}, Pt_{pr,2}, \dots, Pt_{pr,m}\}$ be the list of all pointers of a prefix c , where m is the number of pointers of prefix pr . Pr_{pr} is stored in a variable-length array named **array of pointers**. Let $B = \{Pr_1, Pr_2, \dots, Pr_p\}$ be the list of all prefixes, where p is

the number of distinct items. B is stored in a variable-length array named **array of prefixes**.

Fig. 1 above describes how the representation of Seq-Tree looks like. This framework will be applied to store the sequence database into memory and to construct the index set. The mathematical definition of Seq-Tree framework is divided into two functionalities: (1) for in-memory sequence database and (2) for constructing index set.

For in-memory sequence database, let $C = \{C_1, C_2, \dots, C_n\}$ be an array of customers, where n is the number of distinct customers in the sequence database. C_f is an array of customers in the sequence database. $C_f = \{C_{f,1}, C_{f,2}, \dots, C_{f,m}\}$, m denotes how many transactions C_f makes, and $1 < f < n$. Then, $C_{f,j}$ is an array of items bought by customer f in transaction time j , where $C_{f,j} = \{B_1, B_2, \dots, B_k\}$, k denotes how many items bought for transaction time $C_{f,j}$, $1 < j < m$, $B_1 \subseteq I$, $B_2 \subseteq I, \dots, B_k \subseteq I$, $I = \{i_1, i_2, \dots, i_p\}$ is the list of distinct items, and p is the number of distinct items. Then the Seq-Tree framework for in-memory sequence database is C .

For index set, let $A = \{A_1, A_2, \dots, A_p\}$ be an array of prefixes, where p is the number of distinct items. A_x is an array of pointers for item i_x , $1 \leq x \leq p$. $A_x = \{A_{x,1}, A_{x,2}, \dots, A_{x,n}\}$, where n is the number of distinct customers in the sequence database. $A_{x,f}$ is the list of offset for index set A_x for customer f , where $1 < f < n$, and $A_{x,f} = \{D_1, D_2, \dots, D_y\}$, where y is the occurrence frequency for item i_x on array of customer C_f , $1 \leq y \leq D_y$, $1 < y < length(C_f)$, where $length(C_2)$ is the number of items bought by customer C_2 + number of transaction time by customer C_2 , and $1 < D_1 < length(C_2)$, $1 < D_2 < length(C_2), \dots, 1 < D_y < length(C_2)$. Then the Seq-Tree framework for index set is A .

For in-memory sequence database, the ArrayIntList is used to store items bought for each customer while an integer sign could act as transaction time separator and ArrayList is used to store those collections of items bought for all customers. For index set, the ArrayIntList stores the offset, while an ArrayList stores those arrays of offset for all customers, and another ArrayList stores this ArrayList of offset of all customers for all length-1 prefixes.

B. Separator Database

Separator Database is proposed to store the list of separator indices on each customer. Previously, PrefixSpan used either disk-based access or in-memory database to build the intermediate index set [4] [5] [7]. I-PrefixSpan does not use any of them to create the intermediate index set since it is time-consuming to recheck all items one by one inside the original sequence database. To put it briefly, I-PrefixSpan only registers the indices of the transaction time-stamp separators, instead of keeping the in-memory sequence database until the mining is finished.

Having created separator database when scanning the database at the first step of PrefixSpan and having the in-memory database deleted, then for each candidate pattern with current pattern's index list *curr_pat*, separator list *s_list*, and candidate item's index list *can_pat*,

```

IF (last element of current pattern is appended to
candidate item) THEN
  for (i=1 to curr_pat.getSize())
    separatorPoint = an s_list number closer but
    not less than curr_pat.getElementAt(i);

    IF (can find any can_pat number after
separatorPoint) THEN
      ++SUPPORT_COUNT;
      Break for;
    ELSE
      Continue for;

ELSE IF (last element of current pattern is
assembled to candidate item) THEN
  for (i=1 to curr_pat.getSize())
    separatorPoint = an s_list number closer but
    not less than curr_pat.getElementAt(i)

    IF (can find any can_pat number before
separatorPoint and after
curr_pat.getElementAt(i)) THEN
      ++SUPPORT_COUNT;
      Break for;
    ELSE
      Continue for;

```

Example: For the sequence $\langle(a)(a,b)(b)(c)(b,c)\rangle$ in a sequence database, I-PrefixSpan stores the index set offset of prefix $\langle(a)\rangle$ as $\{1, 3\}$, $\langle(b)\rangle$ as $\{4, 6, 10\}$, and $\langle(c)\rangle$ as $\{8, 11\}$. The separator list stored is $\{2, 5, 7, 9, 12\}$.

To get pattern $\langle(a)(b)\rangle$, the first index of $\langle(a)\rangle$ is retrieved, that is 1. Then, the separator list with index number closer but higher than 1 is retrieved, that is 2. Lastly, all indices in $\langle(b)\rangle$ with index number closer but higher than 2 is retrieved, that is 4, 6, and 10. The intermediate index set for $\langle(a)(b)\rangle$ is yielded, that is $\{4, 6, 10\}$.

To get pattern $\langle(a,b)\rangle$, the first index of $\langle(a)\rangle$ is retrieved, that is 1. Then, the separator list with index number closer but higher than 1 is retrieved, that is 2. Next, index in $\langle(b)\rangle$ index in $\langle(b)\rangle$ with index number higher than 3 but lower than 5 is

retrieved, that is 4. The intermediate index set for $\langle(a,b)\rangle$ is yielded, that is $\{4\}$.

Similarly, to get pattern $\langle(a)(b,c)\rangle$, the intermediate index set from previous step is retrieved, that is $\langle(a)(b)\rangle = \{4, 6, 10\}$. The first index of $\langle(a)(b)\rangle$ is retrieved, that is 4. Then, the separator list with index number closer but higher than 4 is retrieved, that is 5. Next, an index in $\langle c \rangle$ with index number higher than 4 but lower than 5 is searched, but there is no such number. Proceed to the next index of $\langle a \rangle \langle b \rangle$, which is 6. Then, the separator list with index number closer but higher than 6 is retrieved, that is 7. Next, index in $\langle c \rangle$ with index number higher than 6 but lower than 7 is searched, but there are no such number. Proceed to the next index of $\langle(a)(b)\rangle$, that is 10. Then, the separator list with index number closer but higher than 10 is retrieved, that is 12. Next, index in $\langle c \rangle$ with index number higher than 10 but lower than 12 is retrieved, that is 11. The intermediate index set for $\langle(a)(b,c)\rangle$ is yielded, that is $\{11\}$.

On those three mining cases, i.e. mining $\langle(a)(b)\rangle$, $\langle(a,b)\rangle$, and $\langle(a)(b,c)\rangle$ index set, the support count is added to 1 respectively since the intermediate index can be yielded.

IV. EXPERIMENTAL RESULTS

To test the proposed algorithm, a series of performance studies were conducted. All experiments were conducted on an Intel® Pentium® 4 CPU 3.20 GHz (2 CPUs) with 512 MB RAM, running Microsoft Windows XP Professional (5.1, Build 2600) and implementing Java 2 using JDK 1.6.

The subsequent three tests compare performance of PrefixSpan and the proposed I-PrefixSpan on small database. The synthetic datasets used in our experiments were generated using Quest Synthetic Data Generation Code for Association and Sequential Patterns software [1]. The same data generator has been used in most studies on sequential pattern mining.

The first test is on Dataset-1 (C1k|N10|T2|S2|t4|i6), which contains 1000 sequences and 10 distinct items. Both the average number of items in a transaction and the average number of transaction in a sequence are set to 2. On average, a frequent sequential pattern consists of four transactions, and each transaction is composed of 6 items. Fig. 2 below shows the processing time of those two algorithms. I-PrefixSpan consistently outperforms PrefixSpan. On average, I-PrefixSpan is about 2.2 faster than PrefixSpan. It is probably not a big deal when the support is 1% (I-PrefixSpan = 11.39 seconds, PrefixSpan = 25.425 seconds), but it will be a considerable problem when the support is 0.2% (I-PrefixSpan = 295.025 seconds, PrefixSpan = 658.24 seconds).

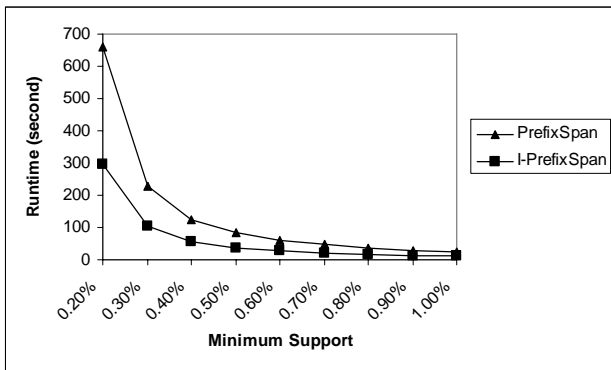


Fig. 2 CPU performance of the two algorithms on Dataset-1

The second test is on Dataset-2 (C1k|N150|T2|S2|t4|i6), which has the same distribution properties with Dataset-1 except that it has 150 distinct items. Fig. 3 below shows the processing time of those two algorithms. I-PrefixSpan persistently outperforms PrefixSpan. The lower the minimum support, the clearer the excellence performance of I-PrefixSpan. When the minimum support is 1%, I-PrefixSpan (7.67 seconds) is almost 4 times faster than PrefixSpan (30.269 seconds). When the minimum support is dwindled to 0.4%, I-PrefixSpan (46.853 seconds) is approximately 4.8 times faster than PrefixSpan (225.302 seconds). Moreover, when the support threshold is 0.2%, I-PrefixSpan (264.98 seconds) runs almost two orders of magnitude faster than PrefixSpan (19,582.97 seconds).

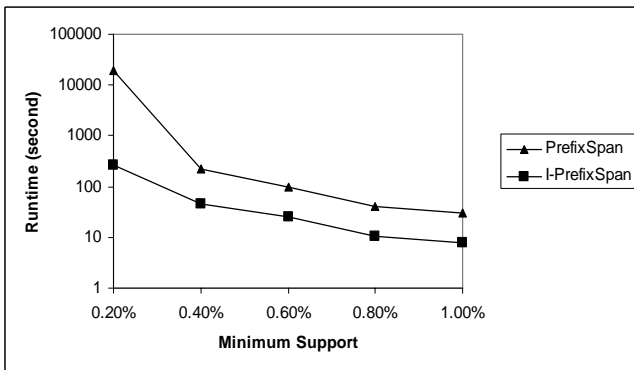


Fig. 3 CPU performance of the two algorithms on Dataset-2

The third test is still on Dataset-2. Fig. 4 below shows the memory usage comparison between those two algorithms. Although I-PrefixSpan just slightly outperforms PrefixSpan when the minimum support is 0.4% or higher, I-PrefixSpan (44.97 MB) dramatically outperforms PrefixSpan (492.36 MB) when the minimum support is 0.2% in more than one order of magnitude.

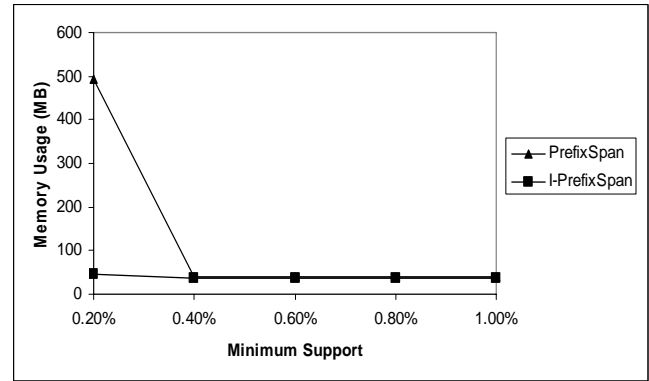


Fig. 4 Memory usage of the two algorithms on Dataset-2

When the number of distinct items grows, Seq-Tree framework with sufficient data structure benefits I-PrefixSpan. ArrayIntList reduced the uncertainty of the List element data type by pre-specifying it as an integer primitive data type, while ArrayList of Integer, which is assumed to be used by PrefixSpan, still required to box and unbox the non-primitive Integer data type. Moreover, ArrayList of Integer needs three times more memory space than ArrayIntList to store an integer value. When creating index set, ArrayIntList benefits I-PrefixSpan from retrieval of all items checking on every data sequence and from appending the pseudoprojection index whenever it finds the searched frequent items. When selecting frequent sequential patterns, ArrayIntList benefits I-PrefixSpan from appending frequent items to current pattern and from iteration to associated index set to search the existence of searched item's offset to decide the support of an item.

Separator database helps I-PrefixSpan to reduce the memory space and in-memory database traversal. Separator database replaces in-memory database with the list of transaction separators. With separator database, there is no need to traverse along all items inside all data sequences. I-PrefixSpan uses separator database to find sequential patterns by comparing the index set of current pattern and the index set of items to be assembled or appended, while PrefixSpan confirms the pointers of the proposed pattern by traversing the index set into in-memory database sequence.

V. CONCLUSION

The experimental results have shown that I-PrefixSpan outperforms PrefixSpan, time-wise and memory-wise. Future research could include the generalization of I-PrefixSpan in handling constrained mining sequential pattern, which extends the ability of I-PrefixSpan on (1) time constraint, i.e. minimum and/or maximum time gaps between adjacent elements in a pattern, (2) sliding windows, i.e. the new definition of time for term "same transaction" as specified by user, and (3) taxonomy, i.e. to include super-category patterns when necessary.

REFERENCES

- [1] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Journal Intelligent Systems*, vol. 9, No.1, 1997, pp. 33 – 56.
- [2] R. Agrawal and R. Srikant, "Mining Sequential Patterns: Generalization and Performance Improvements," Research Report RJ 9994, IBM Almaden Research Center, San Jose, California, December 1995.
- [3] J. Han, and M. Kamber, *Data Mining: Concepts and Techniques*. CA: Prentice Hall, 2002, ch. 5.
- [4] J. Pei, et al, "Mining Sequential Patterns by Pattern Growth: The PrefixSpan Approach," *IEEE Transaction on Knowledge and Data Engineering*, vol. 16, no. 11, pp.14240-1440, Nov. 2004.
- [5] J. Han and J. Pei, "Mining Frequent Patterns by Pattern-Growth: Methodology and Implications", *ACM SIGKDD Explorations (Special Issue on Scalable Data Mining Algorithms)*, vol. 2, no. 2, pp.14-20, December 2000.
- [6] M. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Journal Machine Learning*, vol. 42, nos. 1-2, 2001.
- [7] L.M. Yen and L. S.Y. Lee, "Fast discovery of sequential patterns through memory indexing and database partitioning," *Journal Information Science and Engineering*, vol. 21, pp. 109-128, 2005.
- [8] J. Shirazi, *Java™ Performance Tuning*. CA:O'Reilly, 2003.