JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 21, 109-128 (2005)

Fast Discovery of Sequential Patterns through Memory Indexing and Database Partitioning

MING-YEN LIN AND SUH-YIN LEE*
*Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, 300 Taiwan
E-mail: sylee@csie.nctu.edu.tw
Department of Information Engineering and Computer Science
Feng Chia University
Taichung, 407 Taiwan
E-mail: linmy@fcu.edu.tw

Sequential pattern mining is a challenging issue because of the high complexity of temporal pattern discovering from numerous sequences. Current mining approaches either require frequent database scanning or the generation of several intermediate databases. As databases may fit into the ever-increasing main memory, efficient memory-based discovery of sequential patterns is becoming possible. In this paper, we propose a memory indexing approach for fast sequential pattern mining, named *MEMISP*. During the whole process, *MEMISP* scans the sequence database only once to read data sequences into memory. The find-then-index technique is recursively used to find the items that constitute a frequent sequence and constructs a compact index set which indicates the set of data sequences for further exploration. As a result of effective index advancing, fewer and shorter data sequences need to be processed in *MEMISP* as the discovered patterns get longer. Moreover, we can estimate the maximum size of the total memory required, which is independent of the minimum support threshold, in *MEMISP*. Experimental results indicate that *MEMISP* outperforms both *GSP* and *PrefixSpan* (general version) without the need for either candidate generation or database projection. When the database is too large to fit into memory in a batch, we partition the database, mine patterns in each partition, and validate the true patterns in the second pass of database scanning. Experiments performed on extra-large databases demonstrate the good performance and scalability of MEMISP, even with very low minimum support. Therefore, *MEMISP* can efficiently mine sequence databases of any size, for any minimum support values.

Keywords: data mining, sequential patterns, memory indexing, find-then-index, database partitioning

1. INTRODUCTION

A complicated issue in data mining is the discovery of sequential patterns, which involves finding frequent sub-sequences in a sequence database. For example, in the transactional database of an electronic store, each record may correspond to a sequence of a customer's transactions ordered by transaction time. An example sequential pattern might be one in which customers typically bought a *PC* and *printer*, followed by the pur-

Received December 12, 2002; revised June 25 & December 24, 2003; accepted April 27, 2004.
Communicated by Ming-Syan Chen.

chase of a *scanner* and *graphics software*, and then a *digital camera*. The mining technique is useful for many applications, including the analysis of Web traversal patterns, telecommunication alarms, and DNA sequences to name a few.

Sequential pattern mining is more difficult than association rule mining because the patterns are formed not only by combining items but also by permuting item-sets. Enormous patterns can be formed as the length of a sequence is not limited and the items in a sequence are not necessarily distinct. Let the *size* of a sequence be the total number of items in that sequence. Given 100 possible items in a sequence database, the number of potential patterns of size two is $100 * 100 + (100 * 99)/2$, that of size three is $100 * 100 * 100 + 100 * [(100 * 99)/2] * 2 + (100 * 99 * 98)/(2 * 3)$, and so on. Owing to the challenge posed by exponential possible combinations, improving the efficiency of sequential pattern mining has been the focus of recent research in data mining [2, 3, 7, 9, 11, 13, 17, 18, 20, 21].

In general, the mining approaches are either of the generate-and-test framework or pattern-growth types for sequence databases with horizontal layouts. Typifying the former approaches [2, 8, 16], the *GSP* (**G**eneralized **S**equential **P**attern) algorithm [16] generates potential patterns (called *candidates*), scans each data sequence in the database to compute the frequencies of candidates (called *supports*), and then identifies candidates having enough supports as sequential patterns. The sequential patterns in the current database pass become seeds for generating candidates in the next pass. This generate-and-test process is repeated until no more new candidates are generated. When candidates cannot fit into memory in a batch, *GSP* re-scans the database to test the remaining candidates that have not been loaded into memory. Consequently, *GSP* scans the on-disk database at least k times if the maximum size of the discovered patterns is k, and this incurs much disk reading. Despite the fact that *GSP* was good at candidate pruning, the number of candidates is still huge, which might reduce mining efficiency.

The *PrefixSpan* (**Prefix**-projected **S**equential **pa**ttern mining) algorithm [11], representing the pattern-growth methodology [4, 6, 11], finds the frequent items after scanning the sequence database once. The database is then projected, according to the frequent items, into several smaller databases. Finally, the complete set of sequential patterns is found by recursively growing subsequence fragments in each projected database. Two optimizations for minimizing disk projections were described in [11]. The *bi-level projection* technique, for huge databases scans each data sequence twice in the (projected) database so that fewer and smaller projected databases are generated. The *pseudo-projection* technique, while avoiding physical projections, maintains the sequence-postfix of each data sequence in a projection by means of a pointer-offset pair. However, according to [11], the maximum mining performance can be achieved only when the database size is reduced to the size accommodable by the main memory by employing *pseudo-projection* after using *bi-level* optimization. Although *PrefixSpan* successfully discovers patterns employing the divide-and-conquer strategy, the cost of disk I/O might be high due to the creation and processing of the projected sub-databases.

Besides the horizontal layout, the sequence database can be transformed into a vertical format consisting of items' id-lists [10, 21]. The id-list of an item is a list of (*sequence-id*, *timestamp*) pairs indicating the occurring timestamps of the item in that *sequence*. Searching in the lattice formed by id-list intersections, the *SPADE* (**S**equential **PA**ttern **D**iscovery using **E**quivalence classes) algorithm [21] completes mining in three

passes of database scanning. Nevertheless, additional computation time is required to transform a database with a horizontal layout into one with a vertical format, which will also require several times more storage space than the original sequence database.

With costs falling and installed memory increasing in size, many small or medium sized databases can now fit into main memory. For example, a platform with 256MB of memory can hold a database with one million sequences and a total size of 189MB. Pattern mining performed directly in memory is now possible. However, current approaches discover patterns either through multiple scans of the database or by means of iterative database projections, thereby requiring a huge number of disk operations. The mining efficiency can be improved if the excessive disk I/O is reduced by enhancing memory utilization in the discovery process.

Here, we propose a memory-indexing approach for fast discovery of sequential patterns, called *MEMISP* (**MEM**ory **I**ndexing for **S**equential **P**attern mining). With *MEMISP*, there is neither candidate generation nor database projection, and both CPU and memory utilization are high. *MEMISP* reads data sequences into memory in the first pass, which is also the sole pass, of database scanning. Through index advancement within an index set composed of pointers and position indices to data sequences, *MEMISP* discovers patterns by using a recursive find-then-index technique. When the database is too large to fit into the main memory, we can still mine patterns efficiently in two database scans by running *MEMISP* using a partition-and-validation technique discussed in section 3.3. The experiments we conducted show that *MEMISP* runs faster than both the *GSP* and *PrefixSpan* (without pseudo-projection optimization) algorithms for databases of any size.

The rest of the paper is organized as follows. The problem is formulated and related works are reviewed in section 2. Section 3 presents the *MEMISP* algorithm. The experimental results of mining memory-accommodable databases and extra-large databases are described in section 4. We discuss the performance factors of *MEMISP* in section 5 and conclude the study in section 6.

2. PROBLEM STATEMENT AND RELATED WORK

2.1 Problem Statement

A *sequence s*, denoted by $<e_1e_2 \dots e_n>$, is an ordered set of *n* elements, where each *element* e_i is an itemset. An *itemset*, denoted by $(x_1, x_2, \dots, x_q)$, is a nonempty set of *q* items, where each *item* x_j is represented by a literal. Without loss of generality, items in an element are assumed to be in lexicographic order. The *size* of sequence *s*, written as $|s|$, is the total number of items in all the elements in *s*. Sequence *s* is a *k-sequence* if $|s| = k$. For example, <(a)(c)(a)>, <(a, c)(a)>, and <(b)(a, e)> are all 3-sequences. A sequence $s = <e_1e_2 \dots e_n>$ is a *subsequence* of another sequence $s' = <e_1'e_2' \dots e_m'>$ if there exist $1 \le i_1 < i_2 < \dots < i_n \le m$ such that $e_1 \subseteq e_{i_1}'$, $e_2 \subseteq e_{i_2}'$, …, and $e_n \subseteq e_{i_n}'$. Sequence *s'* *contains* sequence *s* if *s* is a subsequence of *s'*. For example, <(**b**, c)(c)(**a**, c, **e**)> contains <(b)(a, e)>.

Each sequence in the sequence database *DB* is referred to as a *data sequence*. The *support* of sequence *s*, denoted as *s.sup*, is the number of data sequences containing *s* divided by the total number of data sequences in *DB*. *Minsup* is the user specified mini-

mum support threshold. A sequence *s* is a *frequent sequence*, also called a *sequential pattern*, if *s.sup* ≥ *minsup*. Given *minsup* and the sequence database *DB*, the problem of sequential pattern mining is to discover *the set of all sequential patterns*.

An example database *DB* having 6 data sequences is listed in the first column in Table 1. Take the data sequence *C*6 as an example. It has three elements (i.e., three itemsets), the first having items *b* and *c*, the second having item *c*, and the third having items *a*, *c*, and *e*. The support of <(b)(a)> is 4/6 since all the data sequences, except *C*2 and *C*3, contain <(b)(a)>. <(a, d)(a)> is a subsequence of both *C*1 and *C*4; thus, <(a, d)(a)>.*sup* = 2/6. Given *minsup* = 50%, <(b)(a)> is a sequential pattern while <(a, d)(a)> is not. The set of all sequential patterns is shown in the second column of Table 1.

Table 1. Example sequence database *DB* and the sequential patterns.

Sequence	Sequential patterns (*minsup*=50%)
*C*1 = <(a, d)(b, c)(a, e)> *C*2 = <(d, g)(c, f)(b, d)> *C*3 = <(a, c)(d)(f)(b)> *C*4 = <(a, b, c, d)(a)(b)> *C*5 = <(b, c, d)(a, c, e)(a)> *C*6 = <(b, c)(c)(a, c, e)>	<(a)>, <(a)(a) >, <(a)(b)>, <(a, c)>, <(a, e)>, <(b)>, <(b)(a)>, <(b)(a, e)>, <(b)(e)>, <(b, c)>, <(b, c)(a)>, <(b, c)(a, e)>, <(b, c)(e)>, <(b, d)>, <(c)>, <(c)(a)>, <(c)(a, e)>, <(c)(b)>, <(c)(e)>, <(d)>, <(d)(a)>, <(d)(b)>, <(d)(c)>, <(e)>

2.2 Related Works

The problem of sequential pattern mining was first described and solved in [2] with the *AprioriAll* algorithm. In a subsequent work, the same authors proposed the *GSP* algorithm [16], which outperformed *AprioriAll*. The *GSP* algorithm makes multiple passes over the database and finds frequent *k*-sequences at the *k*-th database scan. Initially, each item is a candidate 1-sequence for the first pass. Frequent 1-sequences are determined after all the data sequences in the database are checked. In succeeding passes, frequent (*k* – 1)-sequences are self-joined to generate candidate *k*-sequences, and then any candidate *k*-sequence having a non-frequent sub-sequence is deleted. Again, the supports of candidate *k*-sequences are counted by examining all the data sequences, and then those candidates having the minimum supports become frequent sequences. This process terminates when there are no more candidate sequences. Owing to the generate-and-test nature of this approach, the number of candidates often dominates the overall mining time. However, the total number of candidates increases exponentially as *minsup* decreases, even when effective pruning techniques are used. The *PSP* (**P**refix **S**equential **P**attern) algorithm [8] is similar to *GSP* except that the placement of candidates is improved through a prefix tree arrangement to speed up discovery.

The *FreeSpan* (**Fre**qu**e**nt pattern-projected **S**equential **Pa**ttern Mining) algorithm was proposed to mine sequential patterns using a database projection technique [4]. *FreeSpan* first finds the frequent items after scanning the database once. The sequence database is then projected, according to the frequent items, into several smaller databases. Finally, all sequential patterns are found by recursively growing subsequence fragments

in each database. Based on a similar projection technique, the authors proposed the *PrefixSpan* algorithm [11]. *PrefixSpan* outperforms *FreeSpan* in that only effective postfixes are projected. The *bi-level* and *pseudo-projection* techniques further enhance *PrefixSpan* by enabling it to project fewer sub-databases. However, the combined total size of the projected databases might be several times larger than the size of the original database.

In addition, the *SPADE* algorithm finds sequential patterns using a vertical database layout and join-operations [21]. The vertical database layout transforms data sequences into items' id-lists. The id-list of an item is a list of (*sequence-id*, *timestamp*) pairs indicating the occurring timestamps of the item in that sequence-id. The list pairs are joined to form a sequence lattice, in which *SPADE* searches and discovers patterns [21]. However, transforming the naturally horizontal database into vertical requires additional space since a sequence-id is repeated in several items' id-lists. The gain achieved by using the vertical approach might be diminished owing to the additional space and transforming time required while mining large databases.

In order to boost the mining performance, memory utilization should be increased to minimize the required number of disk operations, especially when dealing ever-larger sequence databases. Therefore, we propose the *MEMISP* algorithm, which will be described next.

3. MEMISP: MEMORY INDEXING FOR SEQUENTIAL PATTERN MINING

In this section, the proposed method for sequential pattern mining, named *MEMISP*, is described. *MEMISP* uses a recursive find-then-index strategy to discover all the sequential patterns from in-memory data sequences. *MEMISP* first reads all the data sequences into memory and counts the supports of 1-sequences (i.e., sequences having only one item). Next, an index set for each frequent 1-sequence is constructed and then frequent sequences are found using the data sequences indicated by the index set. In section 3.1, the mining of an example database is use to explain how the algorithm works. Section 3.2 will present the algorithm. The procedure for dealing with extra-large databases that can not fit into to the main memory space is described in section 3.3. Section 3.4 discusses the differences between *MEMISP* and *PrefixSpan*. Some implementation issues are discussed in section 3.5.

3.1 Mining Sequential Patterns with *MEMISP*: an Example

Definition (Type-1 pattern, type-2 pattern, stem, P-pat) Given a pattern ρ and a frequent item x in the sequence database DB, ρ' is a *type*-1 *pattern* if it can be formed by appending the itemset (x) as a new element to ρ, and is a *type*-2 *pattern* if it can be formed by extending the last element of ρ with x. The frequent item x is called the *stem-item* (abbreviated as *stem*) of the sequential pattern ρ', and ρ is the ***p**refix **pat**tern* (abbreviated as *P-pat*) of ρ'.

For example, given a pattern <(a)> and the frequent item *b*, we obtain the *type*-1 pattern <(a)(b)> by appending (*b*) to <(a)> and the *type*-2 pattern <(a, b)> by extending <(a)> with *b*. <(a)> is the *P-pat*, and *b* is the *stem* of both <(a)(b)> and <(a, b)>. As for a

type-2 pattern <(c)(a, **d**)>, its *P-pat* is <(c)(a)>, and its *stem* is *d*. Note that the null sequence, denoted as <>, is the *P-pat* of any frequent 1-sequence. Clearly, any frequent *k*-sequence is either a *type*-1 pattern or a *type*-2 pattern of a frequent (*k* – 1)-sequence.

Example 1: Given *minsup* = 50% and the *DB* shown in Table 1, *MEMISP* mines the patterns through the following steps.

Step 1: Read *DB* into memory and find frequent 1-sequences. We accumulate the count of every item while reading data sequences from *DB* into memory. The in-memory *DB* will be referred to as ***MDB*** hereafter. Hence, we have frequent items *a* (count = 5 for appearing in 5 data sequences *C*1, *C*3, *C*4, *C*5, *C*6), *b* (count = 6), *c* (count = 6), *d* (count = 5), and *e* (count = 3). All these frequent items are *stems* of the *type*-1 *patterns* with respect to the *P-pat* = <>. **Loop steps 2 and 3 on each stem to find all the sequential patterns.**

Step 2: Output the sequential pattern ρ formed by the current *P-pat* and stem *x*, and construct the index set ρ-*idx*. We output a sequential pattern ρ generated by the current *P-pat* and stem *x*. Next, we allocate a **(ptr_ds, pos)** pair for each data sequence *ds* in *MDB* if and only if *ds* contains *x*, where ***ptr_ds*** is a pointer to *ds* and ***pos*** is the first occurring position of *x* in *ds*. The set of these **(ptr_ds, pos)** pairs is called **index set ρ-*idx***.

Take stem *x* = *a* as an example. Now, the *P-pat* is <>. We output the *type*-1 sequential pattern ρ = **<(*a*)>** and construct the index set **<(*a*)>-*idx*** as shown in Fig. 1-(1). For instance, the ***pos*** is **1** for *C*1 = <(***a***, d)(b, c)(a, e)> and **4** for *C*6 = <(b, c)(c)(**a**, c, e)>.

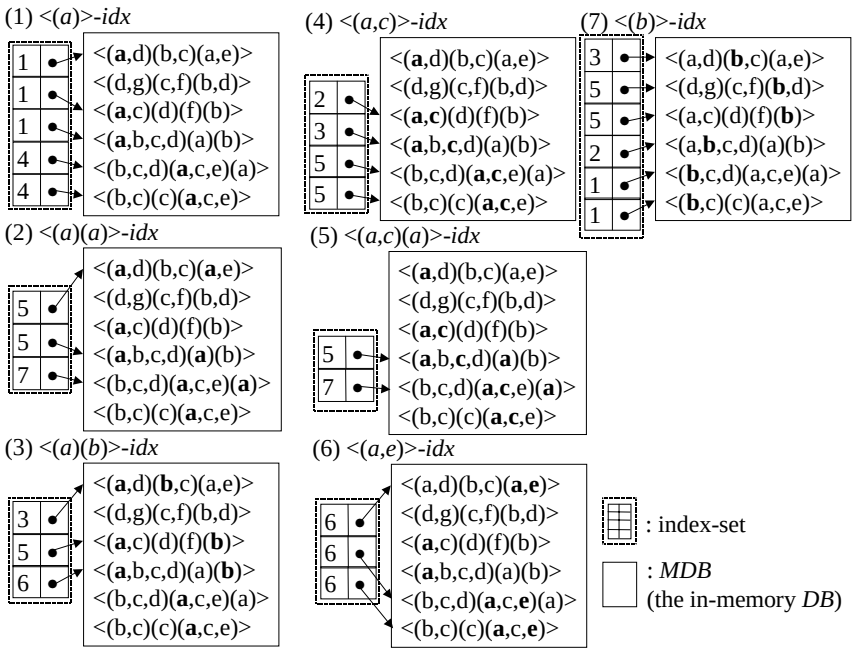

Fig. 1. Some index sets and the in-memory *DB*.

Step 3: Use index set ρ-*idx* and *MDB* to find stems with respect to *P-pat* = ρ. Any sequential pattern having the current pattern ρ as its *P-pat* will be identified in this step. Now, the ***ptr_ds*** of each (**ptr_ds**, **pos**) pair in ***ρ-idx*** points to a data sequence *ds* that contains ρ. Any item appearing after the ***pos*** position in *ds* could be a potential stem (with respect to ρ). Thus, for every *ds* existing in ρ-*idx*, we increase the count of such an item (an item appearing after the ***pos*** in *ds*) by one and then identify the stems having sufficiently large support counts.

Let us continue with <(*a*)>-*idx*. The ***pos*** of the (**ptr_ds**, **pos**) pointing to *C*1 is 1. Only those items occurring after position 1 in *C*1 need to be counted. We increase the count of the potential stem *d* (for the potential *type-2* pattern <(*a*, *d*)>) by one (also the potential stem *e* for <(*a*, *e*)>). We also increase the count of the potential stem *b* (also *c*, *a*, and *e*) for the potential *type*-1 pattern <(*a*)(*b*)> (<(*a*)(*c*)>, <(*a*)(*a*)>, and <(*a*)(*e*)>) by one. Analogously, items occurring after positions 1, 1, 4, and 4 for data sequences *C*3, *C*4, *C*5, and *C*6, respectively, are counted. After validating the support counts, we obtain stems *a*, *b* of *type*-1 patterns and stems *c*, *e* of *type*-2 patterns with respect to *P-pat* = <(*a*)>. Steps 2 and 3 are then recursively applied on the stems *a*, *b*, *c*, and *e* with *P-pat* = <(*a*)>. We continue the mining with stem *a* and *P-pat* = <(*a*)> as follows.

By applying step 2, we generate and output the sequential pattern ρ = **<(*a*)(*a*)>**. Again, a new (**ptr_ds**, **pos**) pair for a data sequence *ds* is inserted into ρ-*idx* (<(*a*)(***a***)>-*idx*) if and only if *ds* contains ρ. While constructing <(*a*)(***a***)>-*idx*, we simply check the data sequences indicated by the current index set, i.e., <(*a*)>-*idx*, rather than those in *MDB*. Assume that a pair (**ptr_ds**, **pos**) in <(*a*)>-*idx* points to *ds*. The search for the occurring position of stem ***a*** (with respect to *P-pat* = <(*a*)>) starts from position **pos + 1** in *ds*. Item ***a*** occurs at *5* in *C*1 and in *C*4, and at 7 in *C*5. No entry is created for *C*3 and *C*6 since item ***a*** cannot be found after positions 1 and 4, respectively. Hence, we have the new index set <(*a*)(*a*)>-*idx* as shown in Fig. 1-(2). Note that the current index set is 'pushed' for later mining before the new index set becomes active.

When step 3 with <(*a*)(*a*)>-*idx* and *MDB* is applied, no stems can form additional sequential patterns. Therefore, mining stops, and the previous index set, i.e., <(*a*)>-*idx*, is popped. Mining continues with stem *b*. Through the creation and mining of <(*a*)(*b*)>-*idx*, pattern **<(*a*)(*b*)>** is output, but no more patterns are found. Next, <(*a*, *c*)>-*idx* is constructed. The result of applying step 2 with <(*a*, *c*)>-*idx* is the generation of **<(*a*, *c*)>** and discovery of the next stem *a*. Thus, <(*a*, *c*)>-*idx* is 'pushed', and <(*a*, *c*)(*a*)>-*idx* is created.

After mining with <(*a*, *c*)(*a*)>-*idx*, which stops when nothing is found and outputs the pattern **<(*a*, *c*)(*a*)>**, the pattern **<(*a*, *e*)>** is generated during mining with <(*a*, *e*)>-*idx*. All the subsequent find-then-index processes regarding stem *a* with *P-pat* = <> now finish.

By collecting the patterns found in the above process, *MEMISP* efficiently discovers all the sequential patterns.

3.2 The *MEMISP* Algorithm

The central idea of *MEMISP* is to utilize memory for both data sequences and indices in the mining process. A computer memory size of 256MB is very common nowa-

days and can accommodate a sequence database having one million sequences and a size of 189MB as indicated by our experiments. Processing sequences in-memory is more efficient than disk-based processing with either multiple scans or iterative projections. *MEMISP* scans only once over the database and reads data sequences into memory during the whole mining process. Starting with sequential patterns of size one, *MEMISP* then discovers all the frequent sequences of larger sizes recursively by searching the set of in-memory data sequences having common sub-sequences. Fig. 2 outlines the proposed *MEMISP* algorithm.

Algorithm *MEMISP*
Input: *DB* = a sequence database; *minsup* = minimum support.
Output: the set of all sequential patterns.
Method:

1. Scan *DB* into *MDB* (the in-memory *DB*), find the set of all frequent items.
2. For each frequent item *x*,
   (i) form the sequential pattern ρ = <(*x*)> and output ρ.
   (ii) call *IndexSet*(*x*, <>, *MDB*) to construct the index set ρ-*idx*.
   (iii) call *Mine*(ρ, ρ-*idx*) to mine patterns with index set ρ-*idx*.

Subroutine *IndexSet*(*x*, ρ, *range-set*)
Parameters: *x* = a stem-item; ρ = a (*P-pat*) pattern; *range-set* = the set of data sequences for indexing. /* If *range-set* is an index set, then each data sequence for indexing is pointed to by the ***ptr_ds*** of the (**ptr_ds**, **pos**) entry in the index set */
Output: index set ρ'-*idx*, where ρ' denotes the pattern formed by stem-item *x* and *P-pat*.
Method:

1. For each data sequence *ds* in *range-set*,
   (i) if *range-set* = *MDB* then *start-pos* = 0; otherwise *start-pos* = *pos*.
   (ii) starting from position (*start-pos* + 1) in *ds*,
      if the stem-item *x* is first found at position ***pos*** in *ds*, then insert a (***ptr_ds***, ***pos***) pair into the index set ρ'-*idx*, where ***ptr_ds*** points to *ds*.
2. Return index set ρ'-*idx*.

Subroutine *Mine*(ρ, ρ-*idx*)
Parameter: ρ = a pattern; ρ-*idx* = an index set.
Method:

1. For each data sequence *ds* pointed to by the *ptr_ds* of an entry (*ptr_ds*, *pos*) in ρ-*idx*,
   (i) starting from position (*pose*+1) to |*ds*| in *ds*, increase the support count of each potential stem *x* by one.
2. Find the set of stems *x* having a larger enough support count to form a sequential pattern.
3. For each stem *x*,
   (i) form the sequential pattern ρ' with *P-pat* ρ and stem *x*, output ρ'.
   (ii) call *IndexSet*(*x*, ρ, ρ-*idx*) to construct the index set ρ'-*idx*.
   (iii) call *Mine*(ρ', ρ'-*idx*) to mine patterns with index set ρ'-*idx*.

Fig. 2. Algorithm *MEMISP*.

In order to speed up the mining process by means of focused search, we construct a *set* which groups the data sequences to be checked. A data sequence *ds* participates in finding pattern ρ' only when *ds* contains the *P-pat* (prefix-pattern) ρ of pattern ρ'. Consequently, for each *ds* containing ρ, we create a pointer *ptr_ds* pointing to *ds* in the set used for exploring patterns ρ' having *P-pat* ρ. The set is denoted as ρ-*idx*. For each data sequence *ds* pointed to ρ-*idx*, we associate *ptr_ds* with a position index *pos* indicating where (in *ds*) we should begin to find potential stems. That is, ρ-*idx* is the set of (*ptr_ds*, *pos*) pairs for discovering patterns whose *P-pat* = ρ.

Take the data sequence *C*6 = <(**b**, **c**)(c)(**a**, c, e)> in memory as an example. We may find <(b)> occurring at position 1, <(b, c)> occurring at composite position (1, 2), and <(b, c)(a)> occurring at composite position (1, 2, 4). Assume that items *b*, *c*, and *a* are frequent. While mining patterns having *P-pat* <(b)>, we include *C*6 in the index set with **pos = 1**, suggesting that only items appearing after position 1 in *C*6 should engage in mining. Similarly, *C*6 will be included in the index set for patterns having *P-pat* <(b, c)> with **pos** = **2** or *P-pat* <(b, c)(a)> with **pos** = **4**. As the discovered *P-pat* becomes longer, the index set will contain fewer data sequences to be processed. Moreover, the number of items in each data sequence remaining to be processed will decrease. Through recursive finding-then-indexing, the proposed *MEMISP* algorithm efficiently discovers sequential patterns.

3.3 Dealing with Extra-Large Databases by Means of Database Partitioning

With memory sizes increasing, many databases will now fit into the main memory of computers without difficulty. Still, some databases might be too large for the main memory to accommodate in a batch. In this case, the sequential patterns are discovered by using a partition-and-validation technique, as shown in Fig. 3.

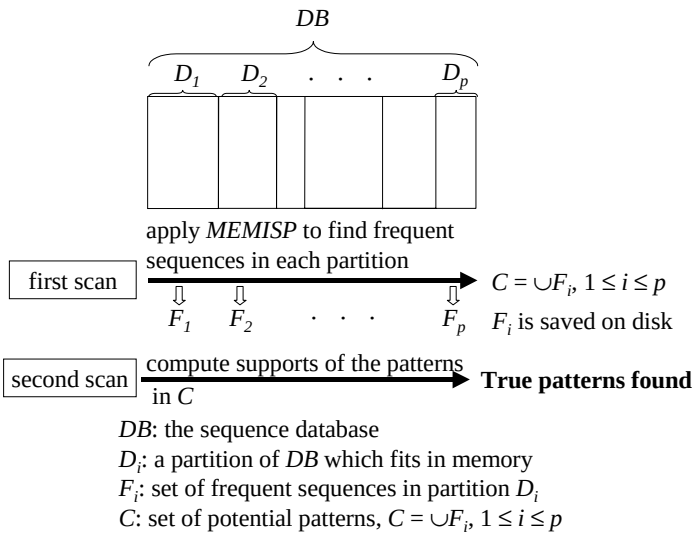

Fig. 3. Partition the database and discover patterns for extra-large databases.

The extra-large database *DB* is partitioned so that each partition can be handled in main memory by *MEMISP*. The number of partitions is minimized by reading as many data sequences into main memory as possible to constitute a partition. The set of potential patterns in *DB* is obtained by collecting the discovered patterns after running *MEMISP* on these partitions. The true patterns can be identified with only one extra database pass through support counting against all the data sequences in *DB* one at a time. Therefore, we can employ *MEMISP* to mine databases of any size, with any minimum support, in just two passes of database scanning.

In comparison with other approaches, *MEMISP* reduces the total number of complete database passes to two without requiring any additional storage space. *SPADE* needs to scan the database three times and demands disk storage space for the transformed vertical database. *GSP* scans at least *k* times to discover the frequent *k*-sequences. *PrefixSpan* often creates and processes projected databases that are several times the original database size.

3.4 Differences between *MEMISP* and *PrefixSpan*

The *PrefixSpan* algorithm proposed in [11] can be optimized with bi-level and pseudo-projection techniques. A pseudo-projection technique keeps redundant pieces of postfixes form being projected when the database/projected database can be held in main memory. *PrefixSpan* and *MEMISP* differ although the two algorithms both utilize memory for fast computation. The differences are illustrated by the following two cases: (1) when the database can be held in main memory; (2) when the database cannot be held in main memory.

When the database can be held in main memory, the two algorithms find the patterns in a similar, but still different way. Both algorithms load the database into memory, but differ in how they process in-memory sequences. The *PrefixSpan* algorithm removes in-frequent items and greatly shrinks projected sequences. Example 3 in [11] clearly demonstrates such projections in that item *g* is not projected in Table 2 in [11]. Pseudo-projection maintains the sequence-postfix of each data sequence in memory by means of a pointer-offset pair. The detailed implementation of *PrefixSpan* with the pseudo-projection technique has not been reported in the literature. However, using just one pointer-offset pair to indicate the occurrence of the prefix is not enough. For example, if C1 = <(**a**, d)(b, c)(**a**, e)> to the pseudo-projected sub-database having prefix <(a)> with just one pointer-offset pair, (*C*1, 1), and if there is no offset specifying 'a' in the third element (a, e), then the 'e' could be missed in support-counting. Thus, each occurrence of the prefix should be recorded in a pointer-offset **list**, rather than by a pointer-offset **pair**.

The *MEMISP* algorithm removes no items from the in-memory sequences. Intermediate in-memory database generation and rearrangement of sequences are not required at all. The sole in-memory sequence database originally loaded is used throughout the whole process. We shift the index without modifying any in-memory sequence to skip the in-frequent items in each iteration. The functionality of the pointer-offset **list** in *PrefixSpan*+pseudo_projection is accomplished by the (*ds_ptr*, *pos*) index pair in *MEMISP*. *MEMISP* matches the prefix from the (*pos* + 1)-th position of the data sequence without using variable-length lists to prevent miscounting. Fast index advancement could eliminate the need to process the in-frequent items.

When the database cannot be held in main memory, *MEMISP* is totally different from *PrefixSpan*. *PrefixSpan*, either with or without pseudo-projection, generates and scans sub-databases that might be several times the original database size. Even with the bi-level projection technique, *PrefixSpan* still might suffer from a low support value when generating many projected sub-databases before pseudo-projection could help. For any support value, *MEMISP* scans the database only twice, and no more times, without generating any intermediate databases.

Bi-level projection is proposed to reduce the number and size of projected databases, at the cost of double scanning to fill the *S-matrix* (see Lemma 3.3 [11]). When extra-large databases are dealt with using bi-level projection, the entire database is scanned at least twice at the beginning. Then, if each projected database can luckily fit into the available memory, then pseudo-projection can be applied. This results in the fewest scans, which is more than twice in total, *PrefixSpan* can do. Otherwise, re-applying bi-level projection could result increasing in the total number of scans to many more than two. Nevertheless, the current version of *PrefixSpan* only uses pseudo-projection optimization since bi-level projection does not always achieve the best performance [23].

MEMISP partitions an extra-large database to several sub-databases; each sub-database can be fit into the available memory. The first scan, which mines each sub-database independently using *MEMISP*, identifies the potential candidates. The second scan verifies whether each candidate has sufficient support to be frequent. *MEMISP* never scans the database, no matter how large it is, more than twice for any value of support. In addition, *MEMISP* never generates any intermediate databases during the mining process. The partition-based approach was used in [14] for association rule mining. However, *MEMISP* is the first algorithm that successfully adapts the partitioning technique to the mining of sequential patterns.

3.5 Implementation Issues

In common implementations, a data sequence is usually represented as a linked list of itemsets in memory. Such a structure might be suitable for algorithms that access a single data sequence at a time for support counting. In order to facilitate fast index construction and speed up searching from specific positions (in data sequences), *MEMISP* uses variable-length arrays to hold the data sequences in memory. Data sequence $C1$ = <(a, d)(b, c)(a, e)>, for instance, is coded as the array = [a, d, $, b, c, $, a, e, $], where $ indicates the end of an element. Therefore, both data sequences and index sets benefit from the array representation in terms of reduced storage space. Efficient searching from specific positions in data sequences is also achieved.

When extra-large databases that require partitioning are mined, a percentage of the main memory (say, 5%) must be reserved for holding variables, index sets, etc. In order to determine whether the main memory can accept any more data sequences, the amount of available physical memory is checked periodically while the database is being loaded into memory. Once the percentage of free memory space falls below a predefined value, *MEMISP* starts to mine the loaded in-memory partition, and the remaining data sequences are handled in subsequent loading runs.

4. EXPERIMENTAL RESULTS

Extensive experiments were conducted to assess the performance of the *MEMISP* algorithm. In the experiments, we used an 866 MHz Pentium-III PC with 256MB memory running Windows NT. Like most studies on sequential pattern mining [2, 4, 11, 15, 16, 21], the synthetic datasets for these experiments were generated using the conventional procedure described in [2]. We will briefly review the generation of experimental data in section 4.1. Section 4.2 will compare the results of mining by the *GSP*, *PrefixSpan*, and *MEMISP* algorithms. To verify that *MEMISP* handles large databases as well, scale-up experiments will be presented in section 4.3.

4.1 Generation of Experimental Data

The procedure described in [2] models a retailing environment, where each customer purchases a sequence of itemsets. Such a sequence is referred to as a *potentially frequent sequence* (abbreviated as *PFS*). Still, some customers might buy only some of the items from a *PFS*. A customer's data sequence may consist of items from several *PFSs*. The *PFS*s are composed of *potentially frequent itemsets* (abbreviated as *PFIs*). A table with a total of N_I *PFIs* (denoted as Γ_I) and a table with a total of N_S *PFSs* (denoted as Γ_S) were generated before items were picked for the transactions of customer sequences.

Table 2. Parameters used in the experiments.

Parameter	Description	Value
$\|DB\|$	Number of data sequences in database *DB*	200K, 500K, 1000K, 10000K
$\|C\|$	Average size (number of transactions) per customer	10, 20
$\|T\|$	Average size (number of items) per transaction	2.5, 5, 7.5
$\|S\|$	Average size of potentially sequential patterns	4, 8
$\|I\|$	Average size of potentially frequent itemsets	1.25, 2.5, 5
N_I	Number of potentially frequent itemsets	25000
N_S	Number of possible sequential patterns	5000
N	Number of possible items	10000
Γ_S	The table of *potentially frequent sequences* (*PFS*s)	
Γ_I	The table of *potentially frequent itemsets* (*PFI*s)	
$corr_S$	Correlation level (sequence), exponentially distributed	$\mu_{corr_S} = 0.25$
$crup_S$	Corruption level (sequence), normally distributed	$\mu_{crup_S} = 0.75$, $\sigma_{crup_S} = 0.1$
$corr_I$	Correlation level (itemset), exponentially distributed	$\mu_{corr_I} = 0.25$
$crup_I$	Corruption level (itemset), normally distributed	$\mu_{crup_I} = 0.75$, $\sigma_{crup_I} = 0.1$

Table 2 summarizes the symbols and the parameters used in the experiments. The procedure for data sequence generation [2] is reviewed here, first the generation of *PFIs* and *PFSs*, and then the customer sequences. The number of itemsets in a *PFS* was generated by picking from a Poisson distribution with mean equal to $|\boldsymbol{S}|$. The itemsets in a *PFS*

are picked from table Γ_I. In order to model that there were common itemsets in frequent sequences, subsequent *PFS*s in Γ_S were related. In the subsequent *PFS*, a fraction of the itemsets were chosen from the previous *PFS*, and the other itemsets were picked at random from Γ_I. The fraction $corr_S$, called *correlation level*, was decided by means of an exponentially distributed random variable with mean equal to μ_{corr_S}. Itemsets in the first *PFS* in Γ_S were randomly picked. The generation of *PFI* and Γ_I was analogous to the generation of *PFS* and Γ_S, with parameter N items, mean $|I|$, *correlation level* $corr_I$, and mean μ_{corr_I} correspondingly.

Customer sequences were generated as follows. The number of transactions for the next customer and the average size of the transactions for this customer were determined first. The size of the customer's data sequence was picked from a Poisson distribution with mean equal to $|C|$. The average size of the transactions was picked from a Poisson distribution with mean equal to $|T|$. Items were then assigned to the transactions of the customer. Each customer was assigned a series of *PFS*s from table Γ_S.

The assignment of *PFS*s was based on the weights of *PFS*s. The weight of a *PFS*, representing the probability that this *PFS* would be chosen, was exponentially distributed and then normalized in such a way that the sum of all the weights was equal to one. Since all the itemsets in a *PFS* were not always bought together, each sequence in Γ_S was assigned a *corruption level* $crup_S$. When itemsets were selected from a *PFS* for a customer sequence, an itemset was dropped if a uniformly distributed random number between 0 and 1 was less than $crup_S$. The $crup_S$ value was a normally distributed random variable with mean μ_{crup_S} and variance σ_{crup_S}. The assignment of *PFI*s (from Γ_I) to a *PFS* was conducted analogously with the parameters $crup_I$, mean μ_{crup_I}, and variance σ_{crup_I}.

All the datasets used here were generated by setting N = 10000, N_S = 5000, and N_I = 25000. A dataset created with $|C| = \alpha$, $|T| = \beta$, $|S| = \chi$, and $|I| = \delta$ was denoted as Cα-Tβ-Sχ-Iδ. In addition, μ_{crup_S} and μ_{crup_I} were both set to 0.75, and σ_{crup_S} and σ_{crup_I} were both set to 0.1. μ_{corr_S} and μ_{corr_I} were both set to 0.25.

4.2 Execution Times of the *GSP*, *PrefixSpan*, and *MEMISP* Algorithms

The total execution times of sequence mining with various *minsup* values of algorithms *GSP*, *PrefixSpan*, and *MEMISP* using a horizontal layout were compared in the experiments. The *SPADE* algorithm was not implemented in the comparison because additional storage space and computation time were required to change the database to a vertical format. We have two versions of *PrefixSpan*: *PrefixSpan*-1 is a general implementation without further optimizations; *PrefixSpan*-2, obtained from Prof. Han [23], is optimized using the pseudo-projection technique.

Dataset C*10*-T*2.5*-S*4*-I*1.25* having 200,000 data sequences (37.6MB) was used in the first experiment. Fig. 4 shows that the total execution times of the four algorithms were nearly the same for *minsup* = 2% because only a few (less than 200) patterns had enough supports. In addition, the discovered patterns were all short patterns of size one. However, the performance gaps became clear as *minsup* decreased. In the experiment, *MEMISP*, *PrefixSpan*-1, and *PrefixSpan*-2 were faster than *GSP* for all *minsup* values. *PrefixSpan*-2 was faster than *MEMISP* because prefix checking in *MEMISP* demands more time than the offset-list operation in *PrefixSpan*-2. *MEMISP* was about 3 to 4 times faster than *PrefixSpan*-1 for a low *minsup* value.

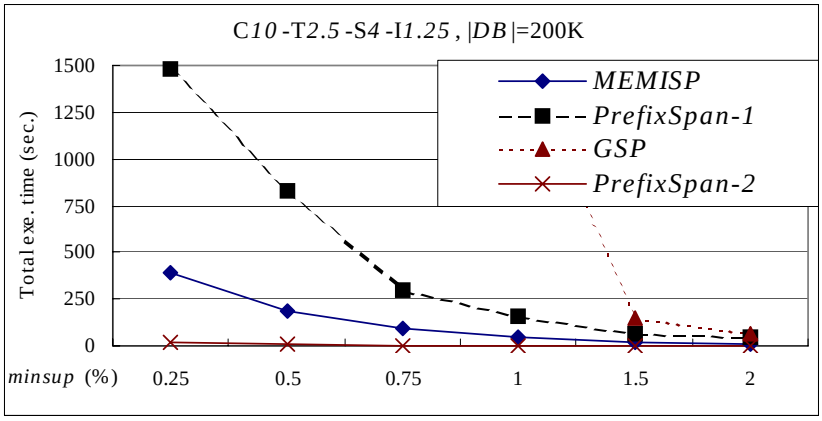

Fig. 4. Total execution times with respect to various *minsup* values.

Next, the characteristics of the datasets were changed. The results of execution on dataset C***20***-T*2.5*-S*4*-I*1.25* (|*DB*| = 200K, 76.3MB) are shown in Fig. 5. The total execution time of running *GSP* was too long to be shown in Fig. 5 and in the subsequent figures. For the same *minsup* value, the doubled |*C*| generated longer data sequences and produced more patterns, thereby requiring more execution time. The total execution time of running *PrefixSpan*-1 was about 5 to 7 times that of running *MEMISP*. The performance of *MEMISP* was comparable to that of *PrefixSpan*-2 for *minsup* values greater than 0.5%. The efficiency of *PrefixSpan*-1 decreased due to the fast growth of the projected databases. For example, the number of data sequences processed by *PrefixSpan*-1 was 4.9 times and 21 times the size of *DB* when *minsup* = 2% and *misup* = 0.75%, respectively. The execution results obtained after changing |*T*| from 2.5 to 5, |*S*| from 4 to 8, and |*I*| from 1.25 to 2.5 showed similar effects. Fig. 6 shows that *MEMISP* outperformed *PrefixSpan*-1. Fig. 7 shows that the performance achieved while running with a bigger |*T*| value and a bigger |*I*| value (|*T*| = 7.5, |*I*| = 5) were consistent with those obtained in the previous experiments. Surprisingly, *PrefixSpan*-2 was slowest when *minsup* = 0.5%. The reason was that it used up to 323.72 MB to discover a large set of patterns. The available memory was not sufficient, so disk-based pseudo-projection slowed the whole process down. In summary, *MEMISP* was faster than *PrefixSpan*-1 for various data characteristics, and the performance gain resulted from in-memory processing of the *MEMISP* algorithm.

4.3 Scale-up Experiments

The maximum size of the datasets used the experiments described in section 4.2 was 76.3MB for the C*20*-T*2.5*-S*4*-I*1.25* dataset with 200,000 sequences. Consequently, all the data sequences could fit into the 256MB main memory. The performance of *MEMISP* was very stable, even when *minsup* was very low for large databases if the database could fit into memory. Given *minsup* = 0.25%, *MEMISP* could perform well in processing one million data sequences of a total size of 189MB and with 256MB main memory in the experiments. Nevertheless, just for the mining of 100K sequences with *minsup* = 0.5%,

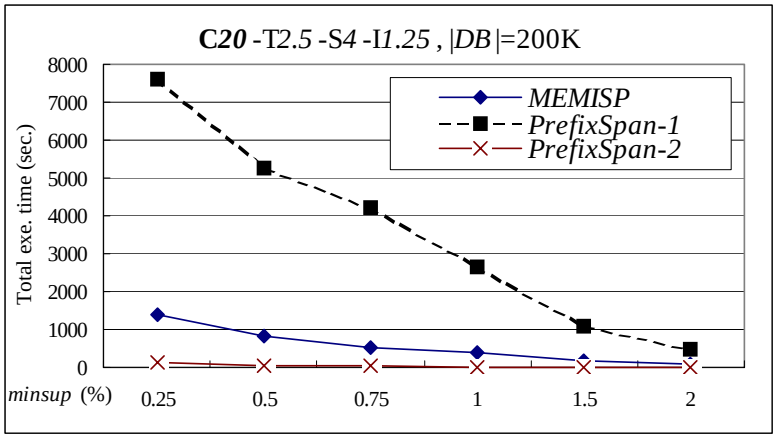

Fig. 5. Comparison of execution times for dataset C*20*-T*2.5*-S*4*-I*1.25*.

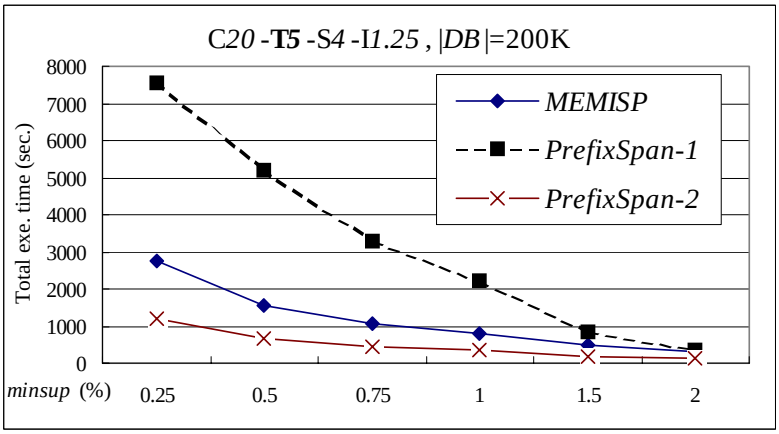

Fig. 6. Comparison of execution times for dataset C*20*-T*5*-S*4*-I*1.25*.

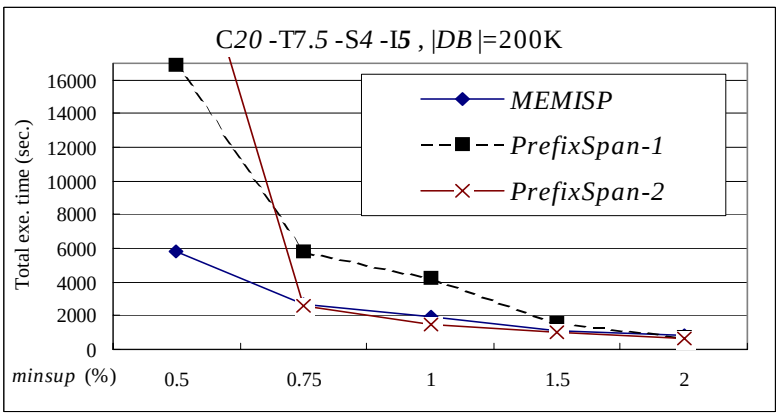

Fig. 7. Comparison of execution times for dataset C*20*-T*7.5*-S*4*-I*5*.

GSP scanned the database 4 times to test the 4.4 million candidates in pass two (more passes to go), and *PrefixSpan*-1 generated sub-databases that, in total, were 9.6 times the size of the original database.

In order to verify the scalability of *MEMISP*, in the following experiments, we increased the number of data sequences from 1000K to 10,000K with C*10*-T*2.5*-S*4*-I*1.25*. As shown in Fig. 8, the total execution times are normalized with respect to the execution time for |*DB*| = 1000K. The size of the dataset having 1000K sequences was 189MB, so *MEMISP* discovered patterns in a single pass without partitioning. *PrefixSpan*-2 used 125.7 seconds for |*DB*| = 1000K and 9635.7 seconds for |*DB*| = 2000K. In addition, *PrefixSpan*-2 could not finish mining in a reasonable amount of time for |*DB*| > 4000K. Although *PrefixSpan*-2 ran very fast when the database could fit into the main memory, its performance worsened dramatically when |*DB*| >= 2000K. That is, once the database could not fit into memory, the cost of disk-based pseudo-projection rose greatly. Note that in Fig. 8, the execution time of *PrefixSpan*-2 is plotted in a different scale.

Given |*DB*| >= 2000K in the experiments, *MEMISP* mined using the partition-and-validation technique described in section 3.3. For example, a dataset of |*DB*| = 10,000K with a size of 1.8GB was mined by means of 10 partitions. Given *minsup* = 0.75% with 10 million sequences, *GSP* could not finish mining in a reasonable amount of time. *PrefixSpan*-1 created the projected databases that together were 11.4 times the original database size. Though Fig. 8 shows that both *PrefixSpan*-1 and *MEMISP* were linearly scalable with the number of data sequences, *MEMISP* showed better scalability.

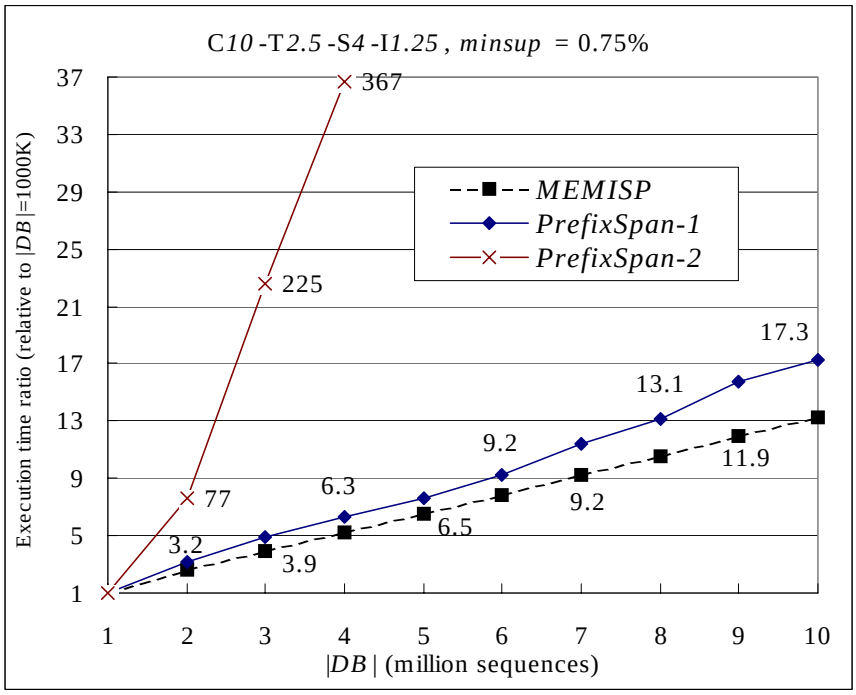

Fig. 8. Linear scalability of *MEMISP* vs. *PrefixSpan*.

5. DISCUSSION

We will summarize the factors contributing to the efficiency of the proposed *MEMISP* algorithm by comparing it with the well-known *GSP* and *PrefixSpan* algorithms.

- **One pass database scanning.** *MEMISP* reads the original database only once, except for extra-large databases as described in section 3.3. In the experiments, a database with one million data sequences could fit into a platform with 256MB memory, so the database was scanned only once by *MEMISP* during mining. However, *GSP* must read the database at least k times, assuming that the maximum size of the discovered patterns is k. *PrefixSpan* reads the original database once, and then writes and reads once for each projected sub-database. In some cases, such as when the *minsup* value is low, the total size of the sub-databases might be several times larger than the size of the original database.
- **No candidate generation.** *MEMISP* discovers patterns directly from data sequences in-memory by means of index advancement. In contrast to *GSP*, *MEMISP* generates no candidates, so the time needed for candidate generation and testing is reduced. Moreover, *MEMISP* works well with a small amount of memory since the unknown sized (and often huge) amount of space for candidate storage is not needed.
- **No database projection.** The pure and simple index advancing approach in *MEMISP* creates no new databases, so the intermediate storage, which *PrefixSpan* needs, is not needed here. Note that *MEMISP* and *PrefixSpan* can achieve similar performance in mining a memory-accommodable database if the *pseudo-projection* technique [11] is used in *PrefixSpan*. However, according to [11], *pseudo-projection* is not efficient if it is used for disk-based accessing and should be employed after *bi-level* optimization is performed [11], which reduces the database size so that it can fit into the main memory.
- **Focused search and effective indexing.** *MEMISP* considers only those data sequences indicated by the current index set instead of searching every data sequence in the database. Furthermore, each position index keeps moving forward along a data sequence as the discovered pattern gets longer. Consequently, fewer and fewer items in a data sequence need to be considered as a prefix pattern gets longer.
- **Compact index storage.** *MEMISP* requires very little storage space for index sets. In an index set, the maximum number of indices required is equal to the number of data sequences, no matter how small the *minsup* value is. Assume that the database has m million sequences. In a 4-byte addressing mode, *MEMISP* demands a maximum of $(4+4) * m$ MB for an index set. The total required memory for discovering the frequent k-sequences will be less than $k * (8 * m)$ MB for any *minsup* value. On the other hand, the memory requirement for storing candidates in *GSP* can hardly be estimated without giving the *minsup* value. Similarly, the total size of the databases projected by *PrefixSpan* increases as the *minsup* value decreases.
- **High CPU and memory utilization.** *PrefixSpan* needs only a little memory space during the mining process. It solves the mining problem successfully by means of sub-database searching, though the CPU maybe idle when sub-databases are projected. *MEMISP*, in contrast, uses all the available memory and maximizes CPU utilization without extra disk operations.

6. CONCLUSIONS

Speeding up the discovery of sequential patterns has been a focus of data mining research. In this paper, we have presented a memory indexing approach for fast discovery of sequential patterns, called *MEMISP*. *MEMISP* mines the set of all sequential patterns without generating candidates or sub-databases. The performance study shows that *MEMISP* is more efficient than both the *GSP* and *PrefixSpan* algorithms, and achieves comparable performance when *PrefixSpan* is optimized using the pseudo-projection technique. *MEMISP* has good linear scalability even under very low minimum supports. Moreover, *MEMISP* can estimate the total memory required, which is independent of the specified *minsup* value. *MEMISP* scans the database at most twice using the partition-and-validation technique even for extra large databases, so the slow disk I/O is minimized. The compact indexing approach and the effective find-then-index technique together make *MEMISP* a promising approach for fast discovery of sequential patterns in sequence databases of any size, even with a small amount of memory and low *minsup* values.

In addition to sequential pattern mining, the technique could be extended to the discovery of maximum patterns [1], constrained/generalized sequential patterns [6, 16, 19], multi-dimensional patterns [12], and incremental sequence discovery after database updating [5, 10, 22]. It would also be interesting to integrate the proposed index sets with database systems to make queries efficient.

REFERENCES

1. R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, "Depth first generation of long patterns," in *Proceedings of 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000, pp. 108-118.
2. R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of 11th International Conference on Data Engineering*, 1995, pp. 3-14.
3. M. N. Garofalakis, R. Rastogi, and K. Shim, "SPIRIT: sequential pattern mining with regular expression constraints," in *Proceedings of 25th International Conference on Very Large Data Bases*, 1999, pp. 223-234.
4. J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. C. Hsu, "FreeSpan: frequent pattern-projected sequential pattern mining," in *Proceedings of 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000, pp. 355-359.
5. M. Y. Lin and S. Y. Lee, "Incremental update on sequential patterns in large databases," in *Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*, 1998, pp. 24-31.
6. M. Y. Lin, S. Y. Lee, and S. S. Wang, "DELISP: efficient discovery of generalized sequential patterns by delimited pattern-growth technology," in *Proceedings of 6th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2002, pp. 198-209.
7. H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, Vol. 1, 1997, pp.

259-289.

8. F. Masseglia, F. Cathala, and P. Poncelet, "The PSP approach for mining sequential patterns," in *Proceedings of 2nd European Symposium on Principles of Data Mining and Knowledge Discovery*, Vol. 1510, 1998, pp. 176-184.
9. T. Oates, M. D. Schmill, D. Jensen, and P. R. Cohen, "A family of algorithms for finding temporal structure in data," in *Proceedings of 6th International Workshop on AI and Statistics*, 1997, pp. 371-378.
10. S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas, "Incremental and interactive sequence mining," in *Proceedings of 8th International Conference on Information and Knowledge Management*, 1999, pp. 251-258.
11. J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M. C. Hsu, "PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth," in *Proceedings of 2001 International Conference on Data Engineering*, 2001, pp. 215-224.
12. H. Pinto, J. Han, J. Pei, K. Wang, Q. Chen, and U. Dayal, "Multi-dimensional sequential pattern mining," in *Proceedings of 10th International Conference on Information and Knowledge Management*, 2001, pp. 81-88.
13. P. Rolland, "FlExPat: flexible extraction of sequential patterns," in *Proceedings of the IEEE International Conference on Data Mining 2001*, 2001, pp. 481-488.
14. A. Sarasere, E. Omiecinsky, and S. Navathe, "An efficient algorithm for mining association rules in large databases," in *Proceedings of 21st International Conference on Very Large Data Bases*, 1995, pp. 432-444.
15. T. Shintani and M. Kitsuregawa, "Mining algorithms for sequential patterns in parallel: Hash based approach," in *Proceedings of 2nd Pacific–Asia Conference on Knowledge Discovery and Data Mining*, 1998, pp. 283-294.
16. R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Proceedings* of *5th International Conference on Extending Database Technology*, 1996, pp. 3-17. (An extended version is the IBM Research Report RJ 9994)
17. S. Thomas and S. Sarawagi, "Mining generalized association rules and sequential patterns using SQL queries," in *Proceedings of 4th International Conference on Knowledge Discovery and Data Mining*, 1998, pp. 344-348.
18. K. Wang, "Discovering patterns from large and dynamic sequential data," *Journal of Intelligent Information Systems*, Vol. 9, 1997, pp. 33-56.
19. M. Wojciechowski, "Interactive constraint-based sequential pattern mining," in *Proceedings of 5th East European Conference on Advances in Databases and Information Systems*, 2001, pp. 169-181.
20. P. H. Wu, W. C. Peng, and M. S. Chen, "Mining sequential alarm patterns in a telecommunication database," in *Proceedings of Databases in Telecommunications 2001*, 2001, pp. 37-51.
21. M. J. Zaki, "SPADE: an efficient algorithm for mining frequent sequences," *Machine Learning Journal*, Vol. 42, 2001, pp. 31-60.
22. M. Zhang, B. Kao, D. Cheung, and C. L. Yip, "Efficient algorithms for incremental update of frequent sequences," in *Proceedings of 6th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2002, pp. 186-197.
23. Private communication with Prof. Jiawei Han.

Ming-Yen Lin (林明言) received the B.S. degree (1988), the M.S. degree (1990), and the Ph.D. degree (2003) in Computer Science and Information Engineering, all from Department of Computer Science and Information Engineering in National Chiao Tung University, Taiwan. His research interests include data mining, data stream management systems, and bioinformatics.

Suh-Yin Lee (李素瑛) received the B.S. degree in Electrical Engineering from National Chiao Tung University, Taiwan, in 1972, the M.S. degree in Computer Science from University of Washington, U.S.A., in 1975, and the Ph.D. degree in Computer Science from Institute of electronics, National Chiao Tung University. Her research interests include multimedia information system, mobile computing, and data mining.