Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1726–1745
The Journal of Systems and Software
www.elsevier.com/locate/jss

SQUIRE: Sequential pattern mining with quantities ☆

Chulyun Kim [a], Jong-Hwa Lim [b], Raymond T. Ng [c], Kyuseok Shim [a,*]

[a] School of Electrical Engineering and Computer Science, Seoul National University, Kwanak, P.O. Box 34, Seoul, Republic of Korea
[b] Department of Computer Science, KAIST, Taejeon, Republic of Korea
[c] Department of Computer Science, University of British Columbia, Vancouver, Canada

Received 23 March 2006; received in revised form 3 December 2006; accepted 7 December 2006
Available online 14 January 2007

Abstract

Discovering sequential patterns is an important problem for many applications. Existing algorithms find qualitative sequential patterns in the sense that only items are included in the patterns. However, for many applications, such as business and scientific applications, quantitative attributes are often recorded in the data, which are ignored by existing algorithms. Quantity information included in the mined sequential patterns can provide useful insight to the users.

In this paper, we consider the problem of mining sequential patterns with quantities. We demonstrate that naive extensions to existing algorithms for sequential patterns are inefficient, as they may enumerate the search space blindly. To alleviate the situation, we propose hash filtering and quantity sampling techniques that significantly improve the performance of the naive extensions. Experimental results confirm that compared with the naive extensions, these schemes not only improve the execution time substantially but also show better scalability for sequential patterns with quantities.

Keywords: Data mining; Knowledge discovery; Sequential pattern mining

1. Introduction

Discovering sequential patterns is a rather well-studied area in data mining and has found many diverse applications, such as basket analysis, telecommunications, etc. (Agrawal and Srikant, 1995; Mannila et al., 1997). While existing studies have considered several variations, which will be discussed later, the basic form of a sequential pattern is $\langle s_1, \ldots, s_m \rangle$, where $s_j = \{i_{j,1}, \ldots, i_{j,n_j}\}$ is a set of items. Note that sequential patterns of this form are essentially qualitative in nature, as there is no quantitative information associated with each item $i_{j,k}$. However, many applications do have quantitative information recorded in the data (e.g., relational data for business, scientific data). Quantitative information is simply ignored in the current form of sequential pattern mining.

In this paper, we formulate the problem of mining sequential patterns with quantities. That is, each item $i_{j,k}$ is represented as a pair $[i_{j,k}, q_{j,k}]$, where the number of units of $i_{j,k}$ is between 0 and $q_{j,k}$. We argue that quantitative sequential patterns of this kind provide meaningful information that is otherwise not reported by qualitative sequential patterns.

In market basket data, assume that we have quantitative sequential pattern, ([*short_pants*, 3])([*jacket*, 2], [*knit_sweater*, 4]) which tells that a customer who purchased up to 3 short pants, may later buy up to 2 jackets and 4 knit sweaters together. Such a quantitative pattern may be very useful for many decisions. For instance, the company may decide to launch a sales promotion by sending coupons to customers who purchased *short_pants*. This quantitative pattern indicates that those customers may be interested in coupons on *jacket* and *knit_sweater*, and the specific quantities

☆ This work was supported by the Ministry of Information and Communication, Korea, under the College Information Technology Research Center Support Program, grant number IITA-2006-C1090-0603-0031.
* Corresponding author. Tel./fax: +82 2 880 7269.
E-mail address: shim@ee.snu.ac.kr (K. Shim).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.12.562

help the company to design the promotion appropriately. In this paper, we make the following contributions:

- We introduce the problem of mining for maximal quantitative sequential patterns. To this end, we propose two naive algorithms; Apriori-QSP is an extension of the Apriori algorithm and PrefixSpan-QSP is an extension of the PrefixSpan (Pei et al., 2001). These algorithms represent the two types of algorithms for finding qualitative sequential patterns. However, these straightforward algorithms are inefficient in several ways.
- We propose two optimizations: hash filtering and quantity sampling. We show that both heuristics are sound. These two optimizations are incorporated into both Apriori-QSP and PrefixSPan-QSP and we call them Apriori-All and PrefixSpan-All respectively. Table 1 summarizes these algorithms for mining maximal sequential patterns with quantities.
- We present preliminary experimental results showing that Apriori-All outperforms Apriori-QSP by at least an order of magnitude. In addition, PrefixSpan-All always performs faster than PrefixSpan-QSP. Between Apriori-All and PrefixSpan-All, we found that the winner varies as the characteristic of data sets changes.

1.1. Patterns from a real-life data set

To demonstrate the usefulness of mining sequential patterns with quantities, we conducted a real-life case study. We used a market basket dataset for the period of six months obtained from a Korean online marketplace company. To restrict our attention to items with multiple quantities, we considered (1) only the items which were purchased in multiple quantities by at least a customer, and (2) only the transactions in which there is at least an item with multiple quantity. In the reduced dataset, the number of customers and the number of items are 559,056 and 157,849 respectively. Below, we show a few interesting frequent patterns with the minimum support 0.01%. To protect confidential and proprietary information of the company, actual product names are encoded.

- $\langle([J,5][T,1])\rangle$: This pattern says that 5 units of J are often co-purchased with one unit of T. According to this pattern, the company may promote a package consisting of 5 items of J and an item T.
- $\langle[B,1][U,3]\rangle$: This pattern says that after one unit of B was bought, it is often that three units of U will be purchased by the same customer. (The maximum elapsed time between the two purchases is upper-bounded by the life span of the dataset.) The company may use this pattern to conduct a target promotion to those customers who bought an item of B.
- $\langle[S,4][S,3][S,4]\rangle$: This pattern shows that the item S is repeatedly purchased with similar quantities by a customer. Beyond its value for sales and promotion, this pattern may be useful for inventory management.

Table 1
Our algorithms for mining maximal sequential patterns with quantities

Algorithms	Sequential patterns with quantities	With hash filtering and quantity sampling
Apriori-style	Apriori-QSP	Apriori-All
PrefixSpan-style	PrefixSpan-QSP	PrefixSpan-All

The paper is organized as follows. In the next section, we present related work. In Section 3, we give preliminary definitions and formally introduce the problem of mining for maximal quantitative sequential patterns. In Section 4 and Section 5, we introduce Apriori-QSP and PrefixSpan-QSP respectively. We present the hash filtering and quantity sampling optimizations. We also show how to incorporate both optimizations into Apriori-QSP and PrefixSPan-QSP. In Section 6, we present preliminary experimental results. Finally, we make concluding remarks in Section 7.

2. Related work

Existing algorithms mining qualitative sequential patterns can be broadly divided into two categories, depending on the order in which the patterns are generated. If we represent all mined sequential patterns in a prefix tree structure, then these patterns can be generated in a breadth-first manner or a depth-first manner. A breadth-first algorithm discovers patterns of the same depth within the same iteration of the algorithm. In other words, patterns of depth 1 are first computed, followed by patterns of depth 2, and so on. This is the well-known Apriori style. Previous studies following this style include Agrawal and Srikant (1995), Agrawal and Srikant (1996) and Mannila et al. (1997). A depth-first algorithm assumes that there is an ordering among items. This ordering is followed when items are processed and patterns are generated in a depth-first manner. The PrefixSpan algorithm in Pei et al. (2001) belongs to this category. SPADE proposed in Zaki (2001) can use both breadth-first and depth-first methods.

There are other extensions to the qualitative sequential pattern mining problem. In Agrawal and Srikant, 1996, a taxonomy of items are considered and efficient algorithms are proposed. The SPIRIT algorithms in Garofalakis et al. (1999) discover all sequential patterns satisfying regular expression constraints given by the user. Mining sequential patterns with constraints is also studied in Pei et al. (2002). Incremental mining of sequential patterns are investigated in Cheng et al. (2004).

Mining association rules can be viewed as a special case of mining qualitative sequential patterns. Algorithms for association rule mining can also be classified as breadth-first or depth-first. The Apriori algorithm initially proposed in Agrawal and Srikant (1994) and Srikant and

Agrawal (1995) is an example of a breadth-first algorithm. The performance of the algorithm can be enhanced by hashing candidate itemsets of size two (Park et al., 1995). Examples of depth-first algorithms include DepthProject (Aggarwal et al., 2000) and FP-Tree (Han et al., 2000). While all these algorithms find all frequent itemsets, one way to speed up the execution time is to restrict the discovery to closed or maximal frequent sets. Examples include Bayardo (1998), Pasquier et al. (1999), Gouda and Zaki (2001) and MAFIA (Burdick et al., 2001). Closed sequential patterns are discussed in Yan et al. (2003).

Last but not least, finding association rules for numeric attributes and interval data has been studied. Examples include Srikant and Agrawal (1996), Fukuda et al. (1996) and Miller and Yang (1997). Given the minimum support threshold as the constraint, the key technical objective of these studies is to search for the largest ranges or intervals $[i_{left}, i_{right}]$ satisfying the constraint. To find sequential patterns with quantities, our approach here is similar in that we try to maximize the width of the range. However, unlike numeric association rules, the left boundary of a range is fixed to 1 in our work (i.e., $i_{left} = 1$). Thus, instead of searching for the values of i_{left} and i_{right}, we only have one variable, namely i_{right}, to maximize. We show in this paper that even with this simplification, the processing is non-trivial, and yet there are natural applications. In future work, we will attempt to generalize the current framework to allow the left boundary to vary.

3. Problem formulation

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of all items. An *extended item* is represented by an item and a quantity, i.e., $[i,n]$ where $i \in I$ and the number of units of i is between 1 and n. Note that we can easily extend n to be a positive real number; for simplicity, here we assume that n is an integer.

The number of distinct quantities per item is finite. The quantity values of an item i are represented by $N_i = \{n_{i_1}, n_{i_2}, \ldots, n_{i_k}\}$ in increasing order. We denote a set of extended items by $EI = \{[i,n] | i \in I \wedge n \in N_i\}$. An *extended itemset eis* is a subset of *EI* where there does not exist a pair of elements having the same item.The subset relationship between *eis* and *EI* is denoted by $eis \subseteq_e EI$.

Given extended itemsets $eis_1 = \{a_1, a_2, \ldots, a_n\}$ and $eis_2 = \{b_1, b_2, \ldots, b_m\}$, eis_1 is called a subset of eis_2 (denoted by $eis_1 \subseteq eis_2$) if, for every a_j, there exists b_k such that a_j is of the form $[i, q_a]$, b_k is of the form $[i, q_b]$, and $q_a \leqslant q_b$.

For instance, given $eis_1 = \{[a,3], [d,10]\}$ and $eis_2 = \{[a,4], [c,1], [d,10]\}$, we have $eis_1 \subseteq eis_2$. However, for $eis_1 = \{[a,3], [d,10]\}$ and $eis_2 = \{[a,4], [c,1], [d,5]\}$, we do not have any subset relationship since the quantity of the last item in eis_2 is 5 and is less than the quantity of the last extended item of eis_1.

A sequential pattern is an ordered list of extended itemsets. We denote a sequential pattern s by $\langle s_1 s_2 \ldots s_l \rangle$ where s_j is an extended itemset (i.e. $s_j \subseteq_e EI$ for $1 \leqslant j \leqslant l$) and is also called an element of the sequential pattern. We represent an element of a sequential pattern by $(x_1 x_2 \ldots x_m)$ where $x_k \in EI$ for $1 \leqslant k \leqslant m$. For brevity, when there is only one extended item (x) (i.e. $m = 1$), we represent this as x without parentheses. Let the total number of extended items in a sequential pattern be the length of the sequential pattern and we represent the sequential pattern with the length of l as l-sequential-pattern. Given the sequential patterns $\alpha = \langle a_1 a_2 \ldots a_n \rangle$ and $\beta = \langle b_1 b_2 \ldots b_m \rangle$, $\alpha = \langle a_1 a_2 \ldots a_n \rangle$ is called a subpattern of $\beta = \langle b_1 b_2 \ldots b_m \rangle$ if there exist integers $j_1, j_2, \ldots, j_n$ such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \ldots, a_n \subseteq b_{j_n}$ for $1 \leqslant j_1 < j_2 < \cdots < j_n \leqslant m$. We also call β a superpattern of α and represent this as $\alpha \sqsubseteq \beta$.

A sequence database, S, is a set of tuples that consists of a sequence id *sid* and a sequence s. We denote a sequence s by $\langle s_1 s_2 \ldots s_l \rangle$ where s_j is an element of the sequence. We represent an element of a sequence by $(x_1 x_2 \ldots x_m)$ in which x_k is the form of $[i : qty]$ where i is an item and qty is the quantity sold for $1 \leqslant k \leqslant m$.

We do not use the square-subset definition right here. Instead, we define directly that a sequence element $[d : n]$ supports an extended itemset $[d,m]$ if and only if we have $n \leqslant m$. We can generalize this to multiple elements and items naturally. With this consideration, all sequences in a sequence database can be represented as sequential patterns. We denote the sequential pattern representation of a sequence s as $p(s)$.

If a sequential pattern α is a subpattern of the sequential pattern form $p(s)$ of a sequence s (i.e. $\alpha \sqsubseteq p(s)$), a tuple $\langle sid, s \rangle$ is said to contain a sequential pattern α. The support of a sequential pattern α is the number of tuples that contain α in the sequence database. In other words, we have $support_S(\alpha) = |\{\langle sid, s \rangle | (\langle sid, s \rangle \in S) \wedge (\alpha \sqsubseteq p(s))\}|$. Whenever no confusion arises, we simplify the notation by using $support(\alpha)$.

Given a minimum support threshold ξ, a sequential pattern α satisfying $support_S(\alpha) \geqslant \xi$ is called a frequent sequential pattern in the sequence database S. A frequent sequential pattern is maximal if it has no frequent superpattern. A frequent sequential pattern with the length of l is called l-pattern. Finally, the problem we are addressing in this paper is formulated as follows:

> Given a sequence database S and a minimum support threshold ξ, find all the frequent maximal sequential patterns satisfying $support_S(\alpha) \geqslant \xi$.

4. Apriori-QSP and the two optimizations

4.1. Apriori-QSP and a running example

In each pass of the conventional Apriori algorithm for sequential pattern mining, we use the frequent patterns from the previous pass to generate the candidate patterns and then measure their support by making a pass over the sequence database. At the end of each pass, the support of the candidates is used to determine the frequent patterns.

Let us consider the case for minimum support of 3 for the sequence database in Fig. 1a. Throughout this paper, we represent each sequence as the sequential pattern form mentioned in previous section. Thus, the notation $[d,3]$ denotes that three units of item d were purchased. Assume that we are in the process of generating candidate sequential 2-patterns from frequent sequential 1-patterns in conventional qualitative sequential pattern mining, i.e., ignoring the quantities for now. If we have two distinct frequent items c and d, we produce 3 candidate sequential patterns: $\langle cd\rangle$, $\langle(cd)\rangle$, $\langle dc\rangle$. The frequent maximal patterns are $\langle b\rangle$, $\langle f\rangle$, $\langle d(ac)\rangle$. The detailed description of Apriori-style algorithms can be found in Agrawal and Srikant (1995) and Agrawal and Srikant (1996).

In order to extend the conventional Apriori algorithm to handle quantities, we have to modify the candidate generation and counting phases to use extended items. Hereafter we call this algorithm Apriori-QSP, which is presented in Fig. 2. Each pass consists of candidate generation and counting in which subsequence pattern checking with extended items is used. The candidate generation phase consists of join and pruning steps. The candidate extended k-patterns C^k are generated with the apriori-gen function by using frequent extended $(k-1)$-patterns F^{k-1} found in the $(k-1)$th pass. Given frequent $(k-1)$-patterns, α and β, if the subpattern obtained by removing the first extended item from α and the subpattern obtained by deleting the last extended item from β are identical, we produce candidate k-patterns by appending α with the last extended item of β. If the appended item is a single element in β, it is appended as a single element in the extended candidate pattern. Otherwise, it is appended to the last element of α.

Any candidate k-pattern that has at least a single unfrequent $(k-1)$-subpattern is pruned. As an example, let us consider the patters $\langle[a,2][b,3]\rangle$ and $\langle[b,3][c,2]\rangle$. Because the last element of the latter pattern is a single item, it is added as a single extended item element in the new extended candidate patterns and results in $\langle[a,2][b,3][c,2]\rangle$. However, for the patterns $\langle[a,2][b,3]\rangle$ and $\langle([b,3][c,2])\rangle$, $[c,2]$ in the latter pattern is located as an item in $([b,3][c,2])$, the extended candidate pattern becomes $\langle[a,2]([b,3][c,2])\rangle$.

Procedure Apriori-QSP(DB)
begin
1. F^1 := { Frequent 1-length patterns }
2. **for** ($k=2$; $F^{k-1}\neq\emptyset$; k++) **do** {
3. C^k := apriori-gen(F^{k-1})
4. counting-support(DB, C^k)
5. F^k := { $x | x\in C^k \wedge x.\text{sup}\geq \text{minsup}$ }
6. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
7. }
8. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
9. **return** $\cup_k F^k$
end

Fig. 2. The Apriori-QSP.

Next, the sequence database is scanned and the support of candidates in C^k is counted. The candidate patterns whose support are at least the given minimum support become the frequent extended patterns F^k. We then prune non-maximal extended patterns from F^{k-1} by *prune-non-maximal* function using F^k. In the *prune-non-maximal* function, we remove subpatterns in F^{k-1} using both F^{k-1} and F^k. For fast pruning, we use a hash table and a hash tree in the *prune-non-maximal*.

Consider the minimum support of 3 with the sequence database in Fig. 1a. The detailed steps of Apriori-QSP including candidate patterns, frequent patterns and frequent maximal patterns for each iteration are illustrated in Fig. 3. We also put the number of candidate patterns generated in the figure. For example, there are 198 candidate patterns with the length of two.

Though Apriori-QSP handles quantities and extended items, it is inefficient in many ways in how candidate patterns are generated. In the rest of this section, we show the heuristics to optimize candidate generation. In Section 4, depth-first approaches, which do not require the generation of candidates to compute quantitative sequential patterns, will be developed. And in Section 5, the effectiveness of all the heuristic algorithms will be evaluated.

a

Sequence id	p(Sequence)
1	$\langle[d,3]([a,3][c,1])[f,1]\rangle$
2	$\langle[a,2]([b,2][e,2][f,5])([d,2][f,2])\rangle$
3	$\langle([b,2][d,1])[c,3]\rangle$
4	$\langle([c,4][d,3])([a,2][c,4][f,5])\rangle$
5	$\langle([d,5][f,3])([a,2][b,3])[e,1][c,4]\rangle$
6	$\langle([c,6][f,5])[d,5]([a,3][c,4])\rangle$

b

Item	Quantities	Item	Quantities
a	2,3	b	2,3
c	1,3,4,6	d	1,2,3,5
e	1,2	f	1, 2, 3, 5

Fig. 1. A running example. (a) Sequence database, and (b) Items and their quantities.

4.2. Hash filtering

Since the number of candidate patterns generated by Apriori-QSP is significantly large, we look for the ways of reducing the number of candidate patterns. One specific weakness of the candidate generation of Apriori-QSP is that it includes all possible quantity values for an item even though the item with a single quantity may not be frequent.

For example, consider Fig. 3. The candidate pattern $\langle([a,2][b,2])\rangle$ is generated because $\langle[a,2]\rangle$ and $\langle[b,2]\rangle$ are frequent patterns of length 1. However, notice that, if we somehow know that the pattern $\langle([a,1][b,1])\rangle$ is not frequent, then we could prune the generation of the candidate $\langle([a,2][b,2])\rangle$. To support this kind of pruning, we propose to use hash-filtering technique. The hash-filtering technique runs the conventional Apriori algorithm, first ignoring

Length	Candidate Patterns	Frequent Patterns	Freq. Max. Patterns
1	N/A	12 $\langle[a,2]\rangle$, $\langle[b,2]\rangle$, $\langle[c,1]\rangle$, $\langle[c,3]\rangle$, $\langle[c,4]\rangle$, $\langle[d,1]\rangle$, $\langle[d,2]\rangle$, $\langle[d,3]\rangle$, $\langle[f,1]\rangle$, $\langle[f,2]\rangle$, $\langle[f,3]\rangle$, $\langle[f,5]\rangle$	$\langle[b,2]\rangle$, $\langle[f,2]\rangle$
2	198 $\langle[a,2][a,2]\rangle$, $\langle[a,2][b,2]\rangle$, $\langle([a,2][b,2])\rangle$, $\langle[b,2][a,2]\rangle$,... , $\langle[d,3][f,5]\rangle$, $\langle([d,3][f,5])\rangle$, $\langle[f,5][d,3]\rangle$	13 $\langle([a,2][c,1])\rangle$, $\langle[d,1][a,2]\rangle$, $\langle[d,2][a,2]\rangle$, $\langle[d,3][a,2]\rangle$, $\langle[d,1][c,1]\rangle$, $\langle[d,1][c,3]\rangle$, $\langle[d,1][c,4]\rangle$, $\langle[d,2][c,1]\rangle$, $\langle[d,2][c,3]\rangle$, $\langle[d,2][c,4]\rangle$, $\langle[d,3][c,1]\rangle$, $\langle[d,3][c,3]\rangle$, $\langle[d,3][c,4]\rangle$	$\langle[d,3][c,4]\rangle$
3	3 $\langle[d,1]([a,2][c,1])\rangle$, $\langle[d,2]([a,2][c,1])\rangle$, $\langle[d,3]([a,2][c,1])\rangle$	3 $\langle[d,1]([a,2][c,1])\rangle$, $\langle[d,2]([a,2][c,1])\rangle$, $\langle[d,3]([a,2][c,1])\rangle$,	$\langle[d,3]([a,2][c,1])\rangle$

Fig. 3. Candidate and frequent patterns by Apriori-QSP.

quantities to find out sequential patterns. This step is called the *filtering step*. The extended item $[i, n_{i_1}]$ is called a basic extended item if i is an item and n_{i_1} is the smallest value of quantities for i. If we ignore quantities in the sequence database and use the conventional sequential pattern mining algorithm, the set of frequent patterns found is the same as the set of frequent patterns with basic extended items. Then we enumerate candidate patterns by putting all possible quantity values to the frequent patterns generated in the filtering step. This step is called the *quantity-counting step*.

As an example, the frequent basic patterns with the length of 2 for the sequence database in Fig. 1 are $\langle(ac)\rangle$, $\langle da\rangle$ and $\langle dc\rangle$. The sets of quantity values of frequent patterns for the items a, c and d are $\{2\}$, $\{1,3,4\}$ and $\{1,2,3\}$ respectively. Thus, with the frequent basic patterns $\langle dc\rangle$, the set of candidate patterns generated are $\langle[d,1][c,3]\rangle$, $\langle[d,1][c,4]\rangle$, $\langle[d,2][c,1]\rangle$, $\langle[d,2][c,3]\rangle$, $\langle[d,2][c,4]\rangle$, $\langle[d,3][c,1]\rangle$, $\langle[d,3][c,3]\rangle$ and $\langle[d,3][c,4]\rangle$. Note that we do not include $\langle[d,1][c,1]\rangle$ here since we have generated $\langle dc\rangle$ as one of the frequent basic patterns in the filtering step, and $\langle dc\rangle$ represents $\langle[d,1][c,1]\rangle$ implicitly.

The algorithm Apriori-Hash that uses hash-filtering is illustrated in Fig. 4. In a subpattern pass k, we first generate the candidate basic extended patterns $\tilde{C}^k$ using only the frequent basic extended patterns $\tilde{F}^{k-1}$, and count their support. Then, we enumerate candidate patterns by putting all possible quantity values to the frequent basic extended patterns $\tilde{F}^k$. We call these patterns enumerated from x the proper superquantity patterns of x. By counting the support of each candidates, we can find all frequent extended patterns. We also prune all non-maximal patterns from F^{k-1} at each pass, thus Apriori-Hash returns all frequent maximal extended patterns.

Let us consider the example illustrated in Fig. 1 with the minimum support of 3. The candidate patterns in the filtering step and the counting step are presented in Fig. 5. When we compute the patterns with the length of 1, we do not use the filtering step, but compute them by a single I/O scan. For the length of 2, we produce 35 candidate patterns to find out frequent 2-patterns, and 12 candidates in the counting step, resulting in 47 candidate patterns as

Procedure Apriori-Hash(DB)
begin
1. F^1 := { Frequent 1-length patterns }
2. $\tilde{F}^1$:= $\{x | x \in F^1 \wedge x$ is a basic extended pattern }
3. **for** ($k = 2$; $F^{k-1} \neq \emptyset$; k++) **do** {
4. $\tilde{C}^k$:= apriori-gen($\tilde{F}^{k-1}$)
5. counting-support(DB, $\tilde{C}^k$)
6. $\tilde{F}^k$:= { $x | x \in \tilde{C}^k \wedge x.\text{sup} \geq \text{minsup}$ }
7. C^k := { $y | y$ is x's proper superquantity pattern where $x \in \tilde{F}^k$ }
8. counting-support(DB, C^k)
9. F^k := $\tilde{F}^k \cup$ { $x | x \in C^k \wedge x.\text{sup} \geq \text{minsup}$ }
10. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
11. }
12. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
13. **return** $\cup_k F^k$
end

Fig. 4. The Apriori-Hash.

shown in Fig. 5. We generate 3 candidate patterns for the length of 3. Note that the number of candidates generated for the length of 2 with hash-filtering is reduced significantly from 198 to 47.

Generalizing from the example, the following lemma shows that hashed filtering cannot increase the number of candidate patterns.

Lemma 4.1. *The number of candidate patterns examined with hash-filtering is at most the number of candidate patterns examined without it.*

Proof. Let the set of candidate patterns with hash-filtering be C_1 and that without hash-filtering C_2. C_1 can be represented by a union of C_{filter} and C_{count} where C_{filter} is the number of candidate patterns in the filtering step and C_{count} is the number of candidates in the quantity-counting step. Since we do not re-generate candidate patterns considered in the filtering step, we have $C_{filter} \cap C_{count} = \emptyset$. We also have $C_{filter} \subseteq C_2$ and $C_{count} \subseteq C_2$ that results in $C_1 \subseteq C_2$. Thus, we have $|C_{filter}| + |C_{count}| = |C_1| \leqslant |C_2|$. □

4.3. Quantity sampling

The next optimization we propose is called quantity sampling. The idea is to produce coarse-grained candidate

Length	Candidate Patterns (Filtering Step)	Frequent Basic Patterns	Candidate Patterns (Counting Step)	Frequent Patterns
2	35 $\langle aa\rangle$, $\langle ab\rangle$, $\langle (ab)\rangle$, $\langle ba\rangle$,..., $\langle df\rangle$, $\langle (df)\rangle$, $\langle fd\rangle$	3 $\langle (ac)\rangle$, $\langle da\rangle$, $\langle dc\rangle$	12 $\langle([a,2][c,3])\rangle$, $\langle([a,2][c,4])\rangle$, $\langle[d,2][a,2]\rangle$, $\langle[d,3][a,2]\rangle$, $\langle[d,1][c,3]\rangle$, $\langle[d,1][c,4]\rangle$, $\langle[d,2][c,1]\rangle$, $\langle[d,2][c,3]\rangle$, $\langle[d,2][c,4]\rangle$, $\langle[d,3][c,1]\rangle$, $\langle[d,3][c,3]\rangle$, $\langle[d,3][c,4]\rangle$	10 $\langle[d,2][a,2]\rangle$, $\langle[d,3][a,2]\rangle$, $\langle[d,1][c,3]\rangle$, $\langle[d,1][c,4]\rangle$, $\langle[d,2][c,1]\rangle$, $\langle[d,2][c,3]\rangle$, $\langle[d,2][c,4]\rangle$, $\langle[d,3][c,1]\rangle$, $\langle[d,3][c,3]\rangle$, $\langle[d,3][c,4]\rangle$
3	1 $\langle d(ac)\rangle$,	1 $\langle d(ac)\rangle$,	2 $\langle[d,2]([a,2][c,1])\rangle$, $\langle[d,3]([a,2][c,1])\rangle$	2 $\langle[d,2]([a,2][c,1])\rangle$, $\langle[d,3]([a,2][c,1])\rangle$

Fig. 5. Candidate and frequent patterns with hash filtering.

patterns first and use the coarse-grained frequent patterns to examine the candidate patterns with more detailed quantity values in order to find the boundary quantity values occurring in the maximal sequential patterns. A set of these candidate patterns generated from a coarse-grained frequent pattern is called a *grid cell*. The first step is called the sampling step and the second step is called the quantity counting step. The first phase quickly prunes the range of quantities that cannot be a part of frequent maximal patterns, and then the second phase actually finds the frequent maximal patterns.

Let us consider the sequence database in Fig. 1a. We assume the minimum support of 3 and $k = 2$ is used for sampling. We also assume that an item i has L_i distinct quantity values in the sequence database. With k intervals where there are at least k quantity values, we use k sampled points. However, if k is greater than L_i for an item i, we select every quantity value for sampling. Note that if the number of distinct quantity values for an item is at most the number of intervals for sampling, the first phase produces candidate patterns with every quantity value. The first phase for the length of 2 generates 89 candidate patterns as presented in Fig. 6a. By scanning the database and counting, we have 7 frequent patterns and we can delete $\langle[d,1][c,1]\rangle$ by subpattern pruning. We use "underscoring" in the figure to represent the frequent patterns that may be pruned by subpattern pruning. Since the number of distinct quantity values of items a and b is the same as the number of intervals, every quantity value is used to generate candidate patterns. We produce 4 candidate patterns from the frequent patterns obtained by the first phase. This results in 93 candidate patterns. When the length of patterns is 3, we generate 2 and 1 candidate patterns in the first and second phases respectively as illustrated in Fig. 6b.

Lemma 4.2. *Even if we remove proper subpatterns in the frequent patterns in the sampling step, all frequent maximal sequential patterns are found.*

Proof. Assume that a frequent pattern α is a proper subpattern of β. The candidate patterns generated with α in quantity counting step are all proper subpatterns of β and cannot be frequent maximal patterns. Thus the removal of proper subpatterns for frequent patterns does not affect the correctness of Apriori algorithm with quantity sampling. □

a

Phase	Candidate Patterns	Frequent Patterns
1	89 $\langle[a,2][a,2]\rangle$, $\langle[a,2][b,2]\rangle$, $\langle([a,2][b,2])\rangle$, ... , $\langle[b,2][a,2]\rangle$,..., $\langle[d,3][f,3]\rangle$, $\langle([d,3][f,3])\rangle$, $\langle[f,3][d,3]\rangle$	7 $\langle([a,2][c,1])\rangle$, $\langle[d,1][a,2]\rangle$, $\langle[d,3][a,2]\rangle$, $\langle[d,1][c,1]\rangle$, $\langle[d,1][c,4]\rangle$, $\langle[d,3][c,1]\rangle$, $\langle[d,3][c,4]\rangle$
2	4 $\langle([a,2][c,3])\rangle$, $\langle[d,2][a,2]\rangle$, $\langle[d,2][c,4]\rangle$, $\langle[d,3][c,3]\rangle$	3 $\langle[d,2][a,2]\rangle$, $\langle[d,2][c,4]\rangle$, $\langle[d,3][c,3]\rangle$

b

Phase	Candidate Patterns	Frequent Patterns
1	2 $\langle[d,1]([a,2][c,1])\rangle$, $\langle[d,3]([a,2][c,1])\rangle$	2 $\langle[d,1]([a,2][c,1])\rangle$, $\langle[d,3]([a,2][c,1])\rangle$
2	1 $\langle[d,2]([a,2][c,1])\rangle$	1 $\langle[d,2]([a,2][c,1])\rangle$

Fig. 6. Candidate and frequent patterns by quantity sampling. (a) Candidate and frequent patterns for the size of 2, and (b) Candidate and frequent patterns for the size of 3.

Due to the above lemma, we can delete any frequent pattern that is a proper subpattern of any other frequent pattern without affecting the output. This optimization is called subpattern pruning. To incorporate this style of pruning into Apriori-QSP, we insert an extra step immediately after the sampling step, which deletes every frequent pattern that is a proper subpattern of other frequent patterns. Fig. 7 presents the Apriori-Sampling that is Apriori-QSP with quantity sampling. In each pass, we use the coarse-grained frequent patterns with sampled quantity, $\bar{F}^{k-1}$, from the previous pass to generate the coarse-grained candidate patterns $\bar{C}^k$ and then measure their support by making a pass over the database DB. After finding the coarse-grained frequent patterns $\bar{F}^k$, we apply the pruning by Lemma 4.2 to reduce the search space. Notice that $\bar{F}^k$ may not include all frequent extended k-patterns due to Lemma 4.2. Thus, we have to keep distinct quantities for

Procedure Apriori-Sampling(DB)
begin
1. F^1 := { Frequent 1-length patterns }
2. $\bar{F}^1 := \{x | x \in F^1 \wedge x$ is a pattern with sampled quantities }
3. **for** ($k = 2$; $F^{k-1} \neq \emptyset$; k++) **do** {
4. $\bar{C}^k$:= apriori-gen($\bar{F}^{k-1}$)
5. counting-support(DB, $\bar{C}^k$)
6. $\bar{F}^k$:= { $x | x \in \bar{C}^k \wedge x.\text{sup} \geq \text{minsup}$ }
7. $\bar{F}^k$:= pruning-proper-subpattern($\bar{F}^k$)
8. C^k := { $y | y \in x$'s grid cell where $x \in \bar{F}^k$ }
9. counting-support(DB, C^k)
10. $F^k := \bar{F}^k \cup \{ x | x \in C^k \wedge x.\text{sup} \geq \text{minsup} \}$
11. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
12. }
13. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
14. **return** $\cup_k F^k$
end

Fig. 7. The Apriori-Sampling.

each frequent items in order to generate all necessary candidates at the next pass. Next, we make the candidate set C^k with the grid cells of pruned $\bar{F}^k$, and mine the rest of the frequent extended k-patterns. All non-maximal patterns are removed from F^{k-1} at the end of each pass and thus Apriori-Sampling finds all frequent maximal extended patterns.

The following lemma shows that quantity sampling can only reduce the total number of candidates.

Lemma 4.3. *The number of candidate patterns generated by Apriori-QSP with quantity sampling technique is at most the number of candidates produced without quantity sampling.*

Proof. Let the set of candidate patterns with quantity sampling be C_1. Then, C_1 is represented by a union of C_{sample} (the set of candidate patterns in the sampling step) and C_{count} (the set of candidate patterns in the quantity counting step). Let us represent the set of candidate patterns without quantity sampling by C_2. Since the candidate patterns generated in the first step do not need to re-generated in the second phase, we have $C_{sample} \cap C_{count} = \emptyset$. Since $C_{sample} \subseteq C_2$ and $C_{count} \subseteq C_2$, we have $C_1 \subseteq C_2$. Furthermore, we have $|C_{sample}| + |C_{count}| = |C_1| \leqslant |C_2|$. □

4.3.1. Determining the number of sample points

Let us investigate the number of candidate patterns with the length of m. Let k be the number of equi-distance intervals to choose sampling points for each item in the candidate patterns. We select k sampling points. We always select the smallest quantity value, but do not consider the largest quantity value for sampling point in each item. For example, if we have quantities ranging from 1 to 5 and k is 2, we can select two intervals [1,3] and [4,6]. Then, we select 1 and 4 as the sampling points. Thus, k also denotes the actual number of sampling points.

For simplicity, we assume that the number of sampling points is same for every item in the candidate patterns. The algorithm with quantity sampling technique produces $k^m|C|$ candidate patterns in the first step where C is the set of candidate patterns with the length m explored by traditional sequential pattern algorithms ignoring quantity values. Assuming that the frequent patterns discovered after subpattern pruning step is F_1 ($0 \leqslant |F_1| \leqslant k^m|C|$) and an item i has L_i distinct quantity values, the counting step then examines $\sum_{c \in F_1} \prod_{1 \leqslant n \leqslant m} (\frac{L_{c.i_n}}{k} - 1)$ candidate patterns where $c.i_n$ denote the nth item of c ($1 \leqslant n \leqslant m$). We subtract one from $\frac{L_{c.i_n}}{k}$ since we do not re-consider the frequent patterns produced by quantity sampling step. Thus, the total number of candidate patterns is

$$k^m|C| + \sum_{c \in F_1} \prod_{1 \leqslant n \leqslant m} \left(\frac{L_{c.i_n}}{k} - 1 \right) \tag{1}$$

A central issue concerning quantity sampling is the value of k, the number of values initially sampled. The value of k that minimizes the number of candidate patterns is obtained by solving partial differential equation of the formula in Eq. (1). Assuming $|F_1|$ is a constant and perform partial differentiation, we get

$$k^{2m} = \frac{1}{|C|} \sum_{c \in F_1} \prod_{1 \leqslant n \leqslant m} L_{c.i_n} \tag{2}$$

By setting L_i to L, we have

$$k = \sqrt[2m]{\frac{|F_1|}{|C|}} \sqrt{L} \tag{3}$$

4.4. Apriori-all

To conclude this section, we show that we can incorporate all the aforementioned optimizations into the basic Apriori-QSP. We call the algorithm Apriori-All, which is presented in Fig. 8. Apriori-All consists of three parts. The first part (lines (4)–(6)) and the second part (lines (7)–(10)) performs the hash-filtering and the quantity sampling respectively. The first part generates basic extended candidate patterns $\tilde{C}^k$ and count them by scanning the sequence database DB. The second part then produces the coarse-granulated candidate patterns from the basic extended frequent patterns. Note that the candidates generated by hash-filtering phase is a subset of the candidates produced by quantity sampling phase. Thus, we do not need to re-consider the candidates examined by hash-filtering phase. The "proper" in line (7) represents this exclusion. After we compute frequent coarse-granulated patterns with sampled quantities and prune using Lemma 4.2, we generate the rest of remaining frequent patterns in the third part (lines (11)–(13)). The frequent patterns are collected into F^k, and non-maximal patterns are removed (lines (14)–(16)).

Let us consider the example illustrated in Fig. 1 with the minimum support of 3. The candidate patterns in the first

Procedure Apriori-All(DB)
begin
1. F^1 := { Frequent 1-length patterns }
2. $\tilde{F}^1 := \{x | x \in F^1 \wedge x$ is a basic extended pattern }
3. **for** ($k = 2$; $F^{k-1} \neq \emptyset$; k++) **do** {
4. $\tilde{C}^k$:= apriori-gen($\tilde{F}^{k-1}$)
5. counting-support($\tilde{C}^k$)
6. $\tilde{F}^k$:= { $x | x \in \tilde{C}^k \wedge x.\text{sup} \geq$ minsup }
7. $\bar{C}^k$:= { $y | y$ is x's proper superpattern with sampled quantities where $x \in \tilde{F}^k$ }
8. counting-support($\bar{C}^k$)
9. $\bar{F}^k := \tilde{F}^k \cup \{ x | x \in \bar{C}^k \wedge x.\text{sup} \geq \text{minsup} \}$
10. $\bar{F}^k$:= pruning-proper-subpattern($\bar{F}^k$)
11. C^k := { $y | y \in x$'s grid cell, $x \in \bar{F}^k$ }
12. counting-support(C^k)
13. $F^k := \bar{F}^k \cup \{ x | x \in C^k \wedge x.\text{sup} \geq \text{minsup} \}$
14. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
15. }
16. F^{k-1} := pruning-non-maximal(F^{k-1}, F^k)
17. **return** $\cup_k F^k$
end

Fig. 8. The Apriori-All.

filtering, the second sampling and the third counting steps are presented in Fig. 9. For the length of 2, we produce 35 candidate patterns for the frequent basic extended 2-patterns in the first step, and 5 and 4 candidates in the second and the third steps respectively as shown in Fig. 9. Notice that we can delete $\langle dc \rangle$, i.e., $\langle [d,1][c,1] \rangle$, by subpattern pruning as we mentioned in the previous subsection. This results in 44 candidate patterns for the length of 2 with Apriori-All, and the number of candidates for the length of 2 is the smallest among those of proposed algorithms.

a

Phase	Candidate Patterns	Frequent Patterns
1	35 $\langle aa \rangle$, $\langle ab \rangle$, $\langle (ab) \rangle$, $\langle ba \rangle$,..., $\langle df \rangle$, $\langle (df) \rangle$, $\langle fd \rangle$	3 $\langle (ac) \rangle$, $\langle da \rangle$, $\underline{\langle dc \rangle}$
2	5 $\langle ([a,2][c,4]) \rangle$, $\langle [d,3][a,2] \rangle$, $\langle [d,1][c,4] \rangle$, $\langle [d,3][c,1] \rangle$, $\langle [d,3][c,4] \rangle$	4 $\langle [d,3][a,2] \rangle$, $\langle [d,1][c,4] \rangle$, $\langle [d,3][c,1] \rangle$, $\langle [d,3][c,4] \rangle$
3	4 $\langle ([a,2][c,3]) \rangle$, $\langle [d,2][a,2] \rangle$, $\langle [d,2][c,4] \rangle$, $\langle [d,3][c,3] \rangle$	3 $\langle [d,2][c,4] \rangle$, $\langle [d,2][a,2] \rangle$, $\langle [d,3][c,3] \rangle$

b

Phase	Candidate Patterns	Frequent Patterns
1	1 $\langle d(ac) \rangle$	1 $\langle d(ac) \rangle$
2	1 $\langle [d,3]([a,2][c,1]) \rangle$	1 $\langle [d,3]([a,2][c,1]) \rangle$
3	1 $\langle [d,2]([a,2][c,1]) \rangle$	1 $\langle [d,2]([a,2][c,1]) \rangle$

Fig. 9. Candidate and frequent patterns by Apriori-All. (a) Candidate and frequent patterns for the size of 2, and (b) Candidate and frequent patterns for the size of 3.

Let us investigate the number of candidate patterns with the length of m. Let k be the number of equi-distance intervals to choose sampling points for each item in the candidate patterns. We assume that an item i has L_i distinct quantity values in the sequence database. We select k sampling points. For simplicity, we assume that the number of sampling points is the same for every item in the candidate patterns. The algorithm with quantity sampling technique produces $k^m|C|$ candidate patterns. Let us denote the set of candidate patterns and frequent patterns considered by hash filtering as C and F respectively.

The first step of quantity sampling produces $|F|(k^m - 1)$ candidate patterns. Let the set of frequent patterns after subpattern pruning be F_1 ($0 \leqslant |F_1| \leqslant |F|k^m$). The second phase of sampling generates $\sum_{c \in F_1} \left(\prod_{1 \leqslant n \leqslant m} \frac{L_{c.i_n}}{k} - 1 \right)$ candidate patterns where $c.i_n$ denote the nth item of c ($1 \leqslant n \leqslant m$) and m is the length of c in F_1.

Therefore, the total number of candidate patterns examined is

$$|C| + |F|(k^m - 1) + \sum_{c \in F_1} \prod_{1 \leqslant n \leqslant m} \left(\frac{L_{c.i_n}}{k} - 1 \right) \tag{4}$$

which is typically much less than the number generated by Apriori-QSP.

The value of k that minimizes the number of candidate patterns is obtained by solving a partial differential equation of the above formula. Assuming that $|F_1|$ is a constant, $L_i = L$ and we perform partial differentiation, we get

$$k = \sqrt[2m]{\frac{|F_1|}{|F|}} \sqrt{L}. \tag{5}$$

Assumed that k is same for both *Apriori-Sampling* and *Apriori-All*. Since $|C| >= |F|$ holds, the difference between Eq. (1) and Eq. (4) becomes $(k^m - 1)(|C| - |F|) \geqslant 0$. Thus,

we can see that the total number of candidate patterns explored by *Apriori-All* is at most (typically smaller than) that of *Apriori-Sampling*. From Eqs. (3) and (5), we can also show that the k value obtained by Eq. (3) is at most (typically smaller than) that obtained by Eq. (5). In other words, it suggests to use smaller number of sampled points in the candidate patterns for *Apriori-Sampling* than *Apriori-All*. Intuitively, it makes senses because we have higher probability of being frequent with the candidate patterns in F than in C.

In each step of generating frequent patterns with the length of l in Apriori-All, we maintain frequent quantity values to calculate L_i so that it can be used to compute k in the next step of discovering frequent patterns with the length of $l+1$. For each item i, we use $k = \sqrt{L_i}$, which is simpler expression of the above equation. The experimental results to be shown in Section 5 are based on this implementation.

The following lemma is a consequence of the earlier lemmas.

Lemma 4.4. *The number of candidate patterns produced by Apriori-QSP with subpattern pruning, hash filtering and quantity sampling is not more than the one without them.*

In Section 6, we will show experimental results on the effectiveness of the two optimizations, as well as the Apriori-All.

5. PrefixSpan-QSP and the two optimizations

Recall that algorithms for mining conventional qualitative sequential patterns can be either breadth-first or depth-first. So far, we have considered breadth-first Apriori-style algorithms. In this section, we consider depth-first prefix tree type algorithms. In the next section, we will compare these two types of algorithms for mining quantitative sequential patterns.

In the subsections, we first introduce PrefixSpan in Pei et al. (2001), and present PrefixSpan-QSP and the two optimizations, hash-filtering and sampling, which are similar to those for Apriori-QSP. The difference is that the former concerns in projection and recursive call, while the latter concerns the candidate generation and support counting.

5.1. PrefixSpan algorithm

Assume that all items in every element of the sequential patterns are ordered alphabetically. Given a sequential pattern $\alpha = \langle e_1 e_2 \dots e_n \rangle$, a sequential pattern $\beta = \langle e'_1 e'_2 \dots e'_m \rangle$ $(m \leqslant n)$ is called a *prefix* of α if and only if: (i) $e'_i = e_i (i \leqslant m-1)$; (ii) $e'_m \subseteq e_m$; and (iii) every item in $(e_m - e'_m)$ is in alphabetical order after the items in e'_m. For example, $\langle a \rangle$, $\langle aa \rangle$, $\langle a(ab) \rangle$, $\langle a(abc) \rangle$ is the prefix of all sequential patterns $\langle a(abc)(ac)d(cf) \rangle$. However, $\langle ab \rangle$ or $\langle a(bc) \rangle$ is not a prefix.

A subsequence pattern α' of a sequential pattern α is called a projection of α with respect to the prefix β if (1) α' has prefix β and (2) there exists no proper supersequence pattern α' of α' such that α' is a subsequence pattern of α and has prefix β. Let $\alpha' = \langle e_1 e_2 \dots e_n \rangle$ be the projection of α with respect to $\beta = \langle e_1 e_2 \dots e_{m-1} e'_m \rangle$ $(m \leqslant n)$. A sequential pattern $\gamma = \langle e''_m e_{m+1} \dots e_n \rangle$ where $e''_m = (e_m - e'_m)$ is called the postfix of α with respect to the prefix β and is represented as $\gamma = \alpha/\beta$. We also represent it as $\alpha = \beta \cdot \gamma$. If e''_m is not an empty set, the prefix is represented as $\langle (\_\text{the items of} e''_m) e_{m+1} \dots e_n \rangle$. For example, $\langle (abc)(ac)d(cf) \rangle$ is the postfix of $\langle a(abc)(ac)d(cf) \rangle$ on the prefix $\langle a \rangle$ and $\langle (\_bc)(ac)d(cf) \rangle$ is the postfix of $\langle a(abc)(ac)d(cf) \rangle$ on the prefix $\langle aa \rangle$. If β is not a subsequence pattern of α, the projection and postfix on β are all null-pattern $\langle\rangle$. If α is a sequential pattern for a sequence database S, α-projected database is the set of postfixes on α in S.

Given a prefix α and its projected database, *PrefixSpan* computes all frequent patterns that start with the prefix α. (Thus, if a null-pattern is given as a prefix, it returns all frequent qualitative sequential patterns in the sequence database.) The *PrefixSpan* essentially either extends the last element of α by adding a new element or appends a new element to the end of α. When we restrict α to be length-1 frequent patterns, it is called level-by-level projection. If we restrict α to be length-2 patterns, it is called bi-level projection. After we find all frequent items, we generate new prefixes by adding each of the items to the prefix α. These new prefixes and their projected databases are collected and the function *PrefixSpan* is called recursively. The step of extending prefixes is done by depth-first traversal. The detailed description of the PrefixSpan algorithm can be found in Pei et al. (2001).

Let us consider the sequence database and minimum support of 3 in Fig. 10 which was obtained by ignoring quantities from the sequence database in Fig. 1a. The frequent items in the sequence database are a, b, c, d and f. Among them, we first would like to discover sequential patterns having prefix $\langle a \rangle$. In this case, we need to consider only the sequential patterns containing $\langle a \rangle$ that are the postfixes with the prefix $\langle a \rangle$. Sequential patterns in Fig. 10a are projected with $\langle a \rangle$ to calculate the $\langle a \rangle$-projected database, which consists of 5 postfixes shown in the second column in Fig. 10b. Note that when we compute postfixes for $\langle a \rangle$ in the given database, we can discard unfrequent items from the postfixes since they will not be frequent in the projected database either. For instance, the item e does not show up in the postfixes of the second and fifth sequential patterns in the second column in Fig. 10.

By scanning $\langle a \rangle$-projected database once, every sequential patterns with length-2 having prefix $\langle a \rangle$ can be generated. For example, $\_c$ is the only frequent item in $\langle a \rangle$-projected database and thus we have $\langle (ac) \rangle$ only for the frequent patterns with length-2 and the prefix $\langle a \rangle$. Now, we have to decide whether we will recursively mine frequent patterns having the prefix $\langle (ac) \rangle$ or not. Since we have only a single frequent item $\_c$, $\langle (ac) \rangle$-projected database will consist of $\_c$ only. However, the duplicated items

a

Sequence id	Sequence
1	$\langle d(ac)f\rangle$
2	$\langle a(bef)(df)\rangle$
3	$\langle (bd)c\rangle$
4	$\langle (cd)(acf)\rangle$
5	$\langle (df)(ab)ec\rangle$
6	$\langle (cf)d(ac)\rangle$

b

sequence id	prefix $\langle a\rangle$	prefix $\langle d\rangle$	prefix $\langle da\rangle$
1	$\langle (\_c)f\rangle$	$\langle (ac)f\rangle$	$\langle (\_c)\rangle$
2	$\langle (bf)(df)\rangle$	$\langle (\_f)\rangle$	
3		$\langle c\rangle$	
4	$\langle (\_cf)\rangle$	$\langle (acf)\rangle$	$\langle (\_c)\rangle$
5	$\langle (\_b)c\rangle$	$\langle (\_f)(ab)c\rangle$	$\langle c\rangle$
6	$\langle (\_c)\rangle$	$\langle (ac)\rangle$	$\langle (\_c)\rangle$

Fig. 10. An example for PrefixSpan. (a) Sequence database, and (b) Projected database.

are not allowed in an itemset and so we do not need to calculate the frequent patterns having the prefix $\langle (ac)\rangle$ any more. In contrast, if $\langle ac\rangle$ were a single frequent pattern with prefix $\langle a\rangle$, we should invoke prefixSpan recursively to discover the frequent patterns with the prefix $\langle ac\rangle$ since we may have the frequent patterns such as $\langle acc\rangle$. The processing of $\langle a\rangle$-projected database is terminated resulting two frequent patterns $\langle a\rangle$ and $\langle ac\rangle$. Likewise, we can produce the sequential patterns with the prefix $\langle b\rangle$, $\langle c\rangle$, $\langle d\rangle$ and $\langle f\rangle$, respectively, by constructing $\langle b\rangle$-, $\langle c\rangle$-, $\langle d\rangle$- and $\langle f\rangle$-projected databases and processing them respectively. Frequent patterns having prefix $\langle d\rangle$ are $\langle da\rangle$, $\langle dc\rangle$ and $\langle d(ac)\rangle$ while we cannot find frequent patterns having $\langle b\rangle$, $\langle c\rangle$ and $\langle f\rangle$. The projected databases used to discover these frequent patterns are presented in the third and forth columns in Fig. 10.

5.2. PrefixSpan-QSP

In order to modify PrefixSpan to work with quantities, we generate a projected database with respect to extended itemsets instead of itemsets. Recall that level-by-level and bi-level projections restrict α to be length-1 and length-2 frequent patterns respectively. To generate those αs, we can apply Apriori-All developed in the previous section to generate frequent *extended itemsets*. However, to adapt to the projection-based approaches such as PrefixSpan, Apriori-All needs to be modified as follows:

- *Incorporating levels*: In order to perform level-by-level or bi-level projections, we augment Apriori-All with *level* as input arguments. The value of *level* is the maximum length of frequent patterns that we are interested in. To perform level-by-level or bi-level projection, we set *level* to one or two respectively.
- *Incorporating* α*s*: Because PrefixSpan is a projection based approach, Apriori-All also needs α and *DB* as input arguments. The reason is that frequent itemsets are produced from the α-projected database *DB* that is projected by the prefix α. With this argument, the procedure Apriori-All returns all frequent maximal patterns up to the length of *level*; we call it F_α.
- *Frequent 1-length pattern generation*: Suppose that we have a prefix $\alpha = \langle [b,2]\rangle$, and two sequential patterns such as $\langle (\_[a,1])\rangle$ and $\langle ([b,3][c,1])\rangle$ with the minimum support count of one. We must produce four frequent 1-length patterns: $\langle (\_[a,1])\rangle$, $\langle [b,3]\rangle$, $\langle [c,1]\rangle$ and $\langle \_[c,1]\rangle$. Note that the last one, $\langle \_[c,1]\rangle$ cannot be discovered without knowing the prefix $\langle [b,2]\rangle$. To do so in Apriori-All,given some prefix $\alpha = \langle e_1e_2\ldots e_n\rangle$ and some sequential pattern $\beta = \langle e'_1e'_2\ldots e'_m\rangle$, we first produce every item in e'_i of β as 1-length pattern. In addition, we compute the postfixes w.r.t. the prefix e_n (i.e. last element) in α on every element e'_i in β and produce $\_a$ as 1-length pattern for each item a in the postfixes. When α is a null-pattern, we do not perform this step of producing the patterns of $\_a$.
- *Candidate generation*: To deal with continued items (e.g., $\_[a,2]$), Apriori-All needs to treat $[a,2]$ and $\_[a,2]$ as different items in the join step of candidate generation and in pruning. In general, even when a continued itemset and a non-continued itemset have the same item and quantity, we still need to treat them as different itemsets. However, a continued itemset needs to be treated specially in the join step of candidate generation. For example, consider the sequential patterns $\langle (\_[a,2])\rangle$ and $\langle [b,3]\rangle$; we produce the candidates, $\langle (\_[a,2])[b,3]\rangle$ and $\langle [b,3][b,3]\rangle$. However, we do not generate the candidate $\langle [b,3](\_[a,2])\rangle$.
- *Support counting*: In order to count supports of candidate patterns, we need to check whether a candidate pattern c is a subsequence pattern of each sequential pattern t in *DB*. If the first element of c does not contain "_", we simply check whether c is a subsequence pattern of t. However, if it does, we concatenate the last element e_n in $\alpha = \langle e_1e_2\ldots e_n\rangle$ in front of both c and t and then check whether $e_n \cdot c$ is a subsequence pattern of $e_n \cdot t$.

Fig. 11 presents the PrefixSpan-QSP algorithm. It accepts α, *DB* and *level* as input arguments. *DB* is a projected database by the prefix α. As discussed above, F_α contains all frequent maximal patterns up to the length of *level*. The next step is to generate new frequent patterns by concatenating α with every pattern in F_α. Among patterns in F_α, the patterns with the length of *level* are called F_α^{level}. We compute G_α^{level} by adding non-maximal frequent patterns generated by enumerating all possible smaller quantity values to the maximal patterns in F_α^{level}. We next perform projections for every element in G_α^{level} and call PrefixSpan-QSP recursively. Note that we consider F_∞^{level} as an empty set. Thus, if we set *level* to ∞, PrefixSpan-QSP behaves as an Apriori style algorithm.

Procedure PrefixSpan-QSP(α, DB, $level$)
begin
1. F_α := Apriori-All(α, DB, $level$)
2. $F := \{x | x := \alpha \cdot y$ for all $y \in F_\alpha\}$
3. $G_\alpha^{level} := \{x | x \in y$'s subquantity sequential patterns where $y \in F_\alpha^{level}\}$
4. **for each** $\beta \in G_\alpha^{level}$ **do** {
5. $DB' :=$ projection($\alpha \cdot \beta$, DB)
6. $F := F \cup$ PrefixSpan-QSP($\alpha \cdot \beta$, DB', $level$)
7. }
8. **return** F
end

Fig. 11. The PrefixSpan-QSP.

Let us consider an example by applying PrefixSpan-QSP to the sequence database in Fig. 1 with the minimum support of 3. If we compute α-projected database with $\alpha = \langle [d,1] \rangle$ which is one of the frequent extended items, we get the sequential patterns in the second column in Fig. 12. The frequent extended items in the above α-projected database are $[a,2]$, $[c,1]$, $[c,3]$ and $[c,4]$. If we next compute α-projected database with respect to $\alpha = \langle [d,1][a,2] \rangle$, it comes the last column in Fig. 12. We have $\_[c,1]$ only as a frequent extended item in this projected database. Now, we have to decide whether we will recursively invoke PrefixSpan-QSP to find frequent patterns with the prefix $\langle [d,1]([a,2][c,1]) \rangle$ or not. Since we have only a single frequent item $\_[c,1]$, $\langle [d,1]([a,2][c,1]) \rangle$-projected database will consist of extended item with the form of $\_[c,q]$ only where q is quantity values for c. Since duplicated items are not allowed in an extended itemset due to its definition, we do not need to further explore the frequent patterns with the prefix $\langle [d,1]([a,2][c,1]) \rangle$ any more. The recursive processing of $\langle [d,1][a,2] \rangle$-projected database is now terminated. After returning from this recursive call, we repeat the iteration recursively with $\alpha = \langle [d,1][c,1] \rangle$, $\langle [d,1][c,3] \rangle$ and $\langle [d,1][c,4] \rangle$. If we calculate the α-projected database with respect to $\alpha = \langle [d,1]([a,2][c,1]) \rangle$, it is empty. Since we do not have any frequent extended item, we repeat the iteration recursively with $\alpha = \langle [d,1][c,1] \rangle$. After returning from this recursive call, we again repeat with $\langle [d,1][c,3] \rangle$ and $\langle [d,1][c,4] \rangle$.The process of producing prefixes is performed by depth-first traversal. Since the length of α increases one at a time for the projections, we call this strategy level-by-level projection. The projection tree by PrefixSpan-QSP with level-by-level projection is shown in Fig. 13 and there are 24 projections. Since the major cost of PrefixSpan is constructing

sequence id	prefix $\langle [d,1] \rangle$	prefix $\langle [d,1][a,2] \rangle$
1	$\langle ([a,3][c,1])[f,1] \rangle$	$\langle (\_[c,1]) \rangle$
2	$\langle (\_[f,2]) \rangle$	
3	$\langle [c,3] \rangle$	
4	$\langle ([a,2][c,4][f,5]) \rangle$	$\langle (\_[c,4]) \rangle$
5	$\langle (\_[f,3])([a,2][b,3])[c,4] \rangle$	$\langle [c,4] \rangle$
6	$\langle ([a,3][c,4]) \rangle$	$\langle (\_[c,4]) \rangle$

Fig. 12. Projected database.

projected databases, a bi-level projection strategy is introduced in Pei et al., 2001 to reduce the number of projected databases. The main idea is to compute frequent patterns with the length of two and construct α-projected database for each pattern α.

In Pei et al. (2001), the optimization technique called 3-way apriori checking is proposed to prune items in the construction of projected database. For instance, the sequence database in Fig. 10a has $\langle dc \rangle$ but not $\langle df \rangle$ as frequent patterns with ignoring quantities. Since we know that $\langle dcf \rangle$ and any supersequence pattern of it cannot be frequent, we can exclude item f from $\langle dc \rangle$-projected database. In the following, we generalize this optimization technique for the sequential patterns with extended items.

Generalized 3-way apriori checking optimization: To construct $\alpha \cdot \beta$-projected database, where α is a given prefix and $\beta \in F_\alpha^{level}$, let e be the last element of β and β' be the prefix of β such that $\beta = \beta' e$.

- If all level-subpatterns of $\beta([x,1])$ are frequent, then $[x,q]$ for every $q \geqslant 1$ is not excluded from any element of postfixes except the first element that is a superset of e.
- Let e' be formed by unioning a extended item $[x,1]$ where $([x,1]) \not\subseteq e$. If all level-subpatterns of $\beta' e'$ are frequent, then $[x,q]$ for every $q \geqslant 1$ is not excluded from the projection.

The $\langle [d,1][a,2] \rangle$-projected database for the sequential pattern 5 is $\langle [c,4] \rangle$ in which $[b,2]$ is excluded as shown in Fig. 12 because $\langle [d,1][b,2] \rangle$ and $\langle ([d,1][b,2]) \rangle$ are not frequent. $\langle [d,1]([a,2][c,1]) \rangle$-projection is skipped because all extended items will be excluded due to this optimization. Essentially, this algorithm adopts the aforementioned bi-level projection with extended items, and uses the Apriori-All algorithm proposed in the previous section at the same time to compute all the length-2 frequent quantitative sequential patterns.

5.3. Hash filtering

In Section 3.2, we introduce the idea of hash filtering for the breadth-first apriori framework. Here we show that the same idea can also be applied in a depth-first framework.

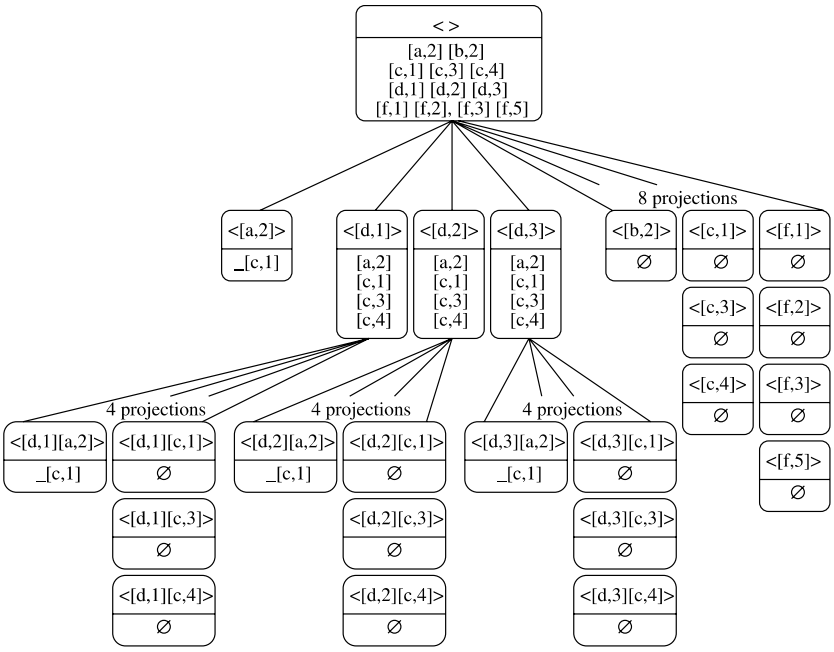

Fig. 13. Projection tree of PrefixSpan-QSP.

Specifically, PrefixSpan-Hash runs the conventional PrefixSpan first ignoring quantities to find out sequential patterns. This step is called the filtering step. Then we enumerate frequent patterns by putting all frequent quantity values to the frequent patterns having frequent postfixes found in the filtering step. Hereafter we generate projection databases of these enumerated frequent patterns and recursively call PrefixSpan-Hash. This step is called the quantity-counting step.

The algorithm PrefixSpan-Hash is presented in Fig. 14. We first find frequent maximal patterns with the length up to *level* in α-projected database DB. We represent these patterns as F_α. In order to produce actual frequent patterns of original database, we concatenate α and every element of F_α. Instead of projecting and recursively calling with all patterns enumerated from F_α^{level} where F_α^{level} is the set of length-*level* patterns in F_α, we probe basic extended patterns of patterns in F_α^{level} called $\tilde{G}_\alpha^{level}$. For patterns $\beta \in \tilde{G}_\alpha^{level}$ whose result obtained by recursive call to PrefixSpan-Hash with $\alpha \cdot \beta$ are not empty, we further project and recursively call with the frequent patterns represented by G_α^{level} which is obtained by replacing frequent bigger quantity values to the quantity value in β.

We will illustrate how PrefixSpan-Hash works by a sequence database in Fig. 1 with the minimum support of 3 and level-by-level. The projection tree with hash filtering is illustrated in Fig. 15. Among all 12 prefixes shown in the root node of Fig. 15, first we compute the projections and recursively invoke PrefixSpan-Hash with the basic extended patterns, $\langle[a,2]\rangle$, $\langle[b,2]\rangle$, $\langle[c,1]\rangle$, $\langle[d,1]\rangle$ and $\langle[f,1]\rangle$. Then, since a has no more quantities to enumerate, there is neither projection nor recursive call for a in the quantity-counting step, and since the results of the recursive calls with $\langle[b,2]\rangle$, $\langle[c,1]\rangle$ and $\langle[f,1]\rangle$ are empty respectively, we do not need to traverse with the superquantity patterns of these patterns such as $\langle[c,3]\rangle$, $\langle[c,4]\rangle$, $\langle[f,2]\rangle$, $\langle[f,3]\rangle$ and $\langle[f,5]\rangle$ in the quantity-counting step. However, the recursive call with $[d,1]$ has non-empty result, therefore we have to find frequent patterns with the superquantity patterns as prefixes, $\langle[d,2]\rangle$ and $\langle[d,3]\rangle$. The gray-colored nodes are traversed in the filtering step, while others are explored in the quantity counting step. Note that the number of projections with hash-filtering is reduced from 24 to 13.

5.4. Quantity sampling

In Section 4.3, we introduce the idea of quantity sampling for the breadth-first apriori framework. Here we show that the same idea can also be applied in a depth-first framework. The idea is to project and recursively call PrefixSpan-Sampling with coarse-grained frequent patterns first and then we project and recursively call PrefixSpan-Sampling with patterns in the grid cells[1] of the coarse-grained frequent patterns having frequent postfixes. The first step is called the sampling step and the second step is called the quantity counting step.

Assume that we are in the process of invoking PrefixSpan-Sampling recursively with the sampled prefixes α and β. Let us define the postfixes w.r.t the prefix θ of the results, which are obtained from recursive call to PrefixSpan-Sampling with the prefix θ and θ-projection, as P_θ. Every element in the grid cell of a sampled prefix α can be pruned for projections and recursive calls whenever α

[1] Defined in subsection 4.3.

Procedure PrefixSpan-Hash(α, DB, $level$)
begin
1. F_α := Apriori-All(α, DB, $level$)
2. $F := \{x | x := \alpha \cdot y$ for all $y \in F_\alpha\}$
3. $\tilde{G}_\alpha^{level} := \{x | x \in F_\alpha^{level} \wedge x$ is a basic extended pattern $\}$
4. **for each** $\beta \in \tilde{G}_\alpha^{level}$ **do** {
5. $DB' :=$ projection($\alpha \cdot \beta$, DB)
6. $\tilde{F} :=$ PrefixSpan-Hash($\alpha \cdot \beta$, DB', $level$)
7. **if** $\tilde{F} \neq \emptyset$ **then** {
8. $G_\alpha^{level} := G_\alpha^{level} \cup \{x | x \in \beta$'s proper superquantity patterns$\}$
9. $F := \tilde{F} \cup F$
10. }
11. }
12. **for each** $\beta \in G_\alpha^{level}$ **do** {
13. $DB' :=$ projection($\alpha \cdot \beta$, DB)
14. $F := F \cup$ PrefixSpan-Hash($\alpha \cdot \beta$, DB', $level$)
15. }
16. **return** F
end

Fig. 14. The PrefixSpan-Hash.

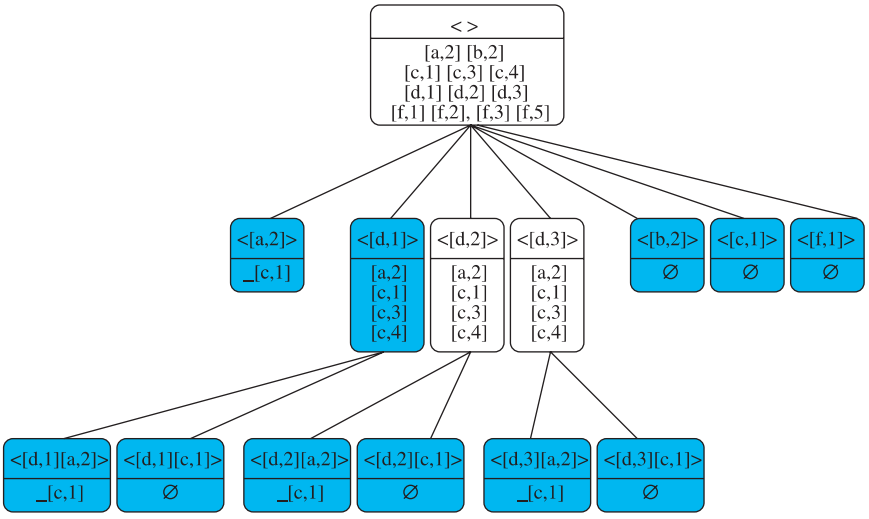

Fig. 15. Projection Tree of PrefixSpan-Hash.

has a proper supersequence pattern β satisfying $P_\alpha = P_\beta$, due to the following lemma.

Lemma 5.1. *Assume that we are in the process of invoking PrefixSpan-Sampling recursively with the prefixes* $\alpha \sqsubseteq \gamma_1 \sqsubseteq \gamma_2 \sqsubseteq \cdots \sqsubseteq \gamma_n \sqsubseteq \beta$. *When the set of postfixes w.r.t.* α *of the results obtained from recursive call to PrefixSpan-Sampling with* α *and* α*-projection are identical to the postfixes w.r.t.* β *of those with* β *and* β*-projection, all frequent maximal sequential patterns are found even though we prune the recursive calls with the prefixes* γ_i *for all* $i = 1, \ldots, n$.

Proof. Let us define the postfixes w.r.t θ of the results obtained from recursive call to PrefixSpan-Sampling with θ and θ-projection as P_θ. For a given database, if we have $\alpha \sqsubseteq \beta$ for the prefixes α and β, we have $P_\alpha \subseteq P_\beta$. It implies $P_\alpha \supseteq P_{\gamma_1} \supseteq P_{\gamma_2} \supseteq \cdots \supseteq P_{\gamma_n} \supseteq P_\beta$. When we have $P_\alpha = P_\beta$, the condition of $P_\alpha = P_{\gamma_1} = P_{\gamma_2} = \cdots = P_{\gamma_n} = P_\beta$ must hold. This results that the recursive call with γ_i does not add any frequent maximal sequential patterns. □

The algorithm PrefixSpan-Sampling algorithm is shown in Fig. 16. We first find frequent maximal patterns with the length up to *level* in α-projected database DB. We represent these patterns as F_α. In order to produce actual frequent patterns of original database, we concatenate α and every element of F_α. Instead of projecting and recursively calling with all patterns enumerated from F_α^{level} where F_α^{level} is the set of length-*level* patterns in F_α, we probe patterns with sampled quantities of F_α^{level} called $\tilde{G}_\alpha^{level}$. For patterns $\beta \in \tilde{G}_\alpha^{level}$ whose result obtained by recursive call to PrefixSpan-Sampling with $\alpha \cdot \beta$ are not empty, we further project and recursively call to PrefixSpan-Sampling with the

Procedure PrefixSpan-Sampling(α, DB, $level$)
begin
1. F_α := Apriori-All(α, DB, $level$)
2. $F := \{x | x := \alpha \cdot y$ for all $y \in F_\alpha\}$
3. $\tilde{G}_\alpha^{level} := \{x | x \in F_\alpha^{level} \wedge x$ is a pattern with sampled quantities $\}$
4. **for each** $\beta \in \tilde{G}_\alpha^{level}$ **do** {
5. $DB' :=$ projection($\alpha \cdot \beta$, DB)
6. $\tilde{F} :=$ PrefixSpan-Sampling($\alpha \cdot \beta$, DB', $level$)
7. **if** $\tilde{F} \neq \emptyset$ **and** $\neg\exists\gamma$ s.t. $(\gamma \sqsupset \beta \wedge P_\gamma = P_\beta)$ **then** {
8. $G_\alpha^{level} := G_\alpha^{level} \cup \{x | x \in \beta$'s grid cell$\}$
9. $F := \tilde{F} \cup F$
10. }
11. }
12. **for each** $\beta \in G_\alpha^{level}$ **do** {
13. $DB' :=$ projection($\alpha \cdot \beta$, DB)
14. $F :=$ PrefixSpan-Sampling($\alpha \cdot \beta$, DB', $level$) $\cup F$
15. }
16. **return** F
end

Fig. 16. The PrefixSpan-Sampling.

frequent patterns in β's grid cell. Note that we do not need to consider grid cells pruned by Lemma 5.1. To decide efficiently whether a grid cell is pruned or not, we sequentially probe sampled patterns with the same items from the largest quantities to the smallest quantities. The unioned set of grid cells we actually process is denoted by G_α^{level}.

Consider a sequence database in Fig. 1 with the minimum support of 3. Suppose that we use level-by-level projection and we sample two intervals from possible quantities of an extended item. First, we compute the projections and recursively invoke PrefixSpan-Sampling with the sampled extended patterns, $\langle[a,2]\rangle$, $\langle[b,2]\rangle$, $\langle[c,1]\rangle$, $\langle[c,4]\rangle$, $\langle[d,1]\rangle$, $\langle[d,3]\rangle$, $\langle[f,1]\rangle$ and $\langle[f,3]\rangle$ from all possible 12 prefixes. For a and b, since the number of possible quantities is only one, a single, actually all, quantity is sampled. For others, two quantities are sampled with equi-width. Then, since the grid cell of a is empty, we cannot further explore for a. Furthermore, because the results of the recursive calls with $\langle[b,2]\rangle$, $\langle[c,1]\rangle$, $\langle[c,4]\rangle$, $\langle[f,1]\rangle$ and $\langle[f,3]\rangle$ are all empty, we do not need to traverse with patterns in the grid cells of these patterns in the quantity-counting step. In the case of d, now that $P_{\langle[d,1]\rangle}$ is the same as $P_{\langle[d,3]\,\rangle}$, we are sure that $P_{\langle[d,2]\rangle}$ is the same as $P_{\langle[d,1]\rangle}$ and $P_{\langle[d,3]\,\rangle}$ by Lemma 5.1. Thus, we do not need to invoke a recursive call, nor do we need to perform a projection with the prefix $\langle[d,2]\rangle$. Finally, since the grid cell of $\langle[d,3]\rangle$ is empty, the process of PrefixSpan-Sampling is terminated now. The projection tree with quantity sampling is presented in Fig. 17. Because we do not need to perform the quantity counting step with this example, all nodes are explored in sampling step and are gray-colored. Note that the number of projections with sampling is reduced from 24 to 14.

5.5. PrefixSpan-all

We can incorporate all the aforementioned optimizations into the PrefixSpan-QSP and call it PrefixSpan-All that is presented in Fig. 18. After finding frequent maximal patterns with the length up to *level* in α-projected database DB, the rest of steps in the PrefixSpan-All are composed of the following three steps.

First, we perform projections on DB with the prefix of $\alpha \cdot \beta$ for every value of β in $\tilde{G}_\alpha^{level}$ where $\tilde{G}_\alpha^{level}$ represents length-*level* basic extended patterns of frequent postfix patterns in F_α. Then, if the recursively invoked PrefixSpan-All returns more than one frequent pattern, we gather super-quantity patterns of β which have sampled quantities including β. The collection, $\bar{G}_\alpha^{level}$, is used in the second step below.

Secondly, we perform projections on DB with the prefix of $\alpha \cdot \beta$ for every value of β in $\bar{G}_\alpha^{level}$ with the exception of basic extended patterns that already have their projected database and frequent postfixes. Then, if the recursively invoked PrefixSpan-All returns more than one frequent

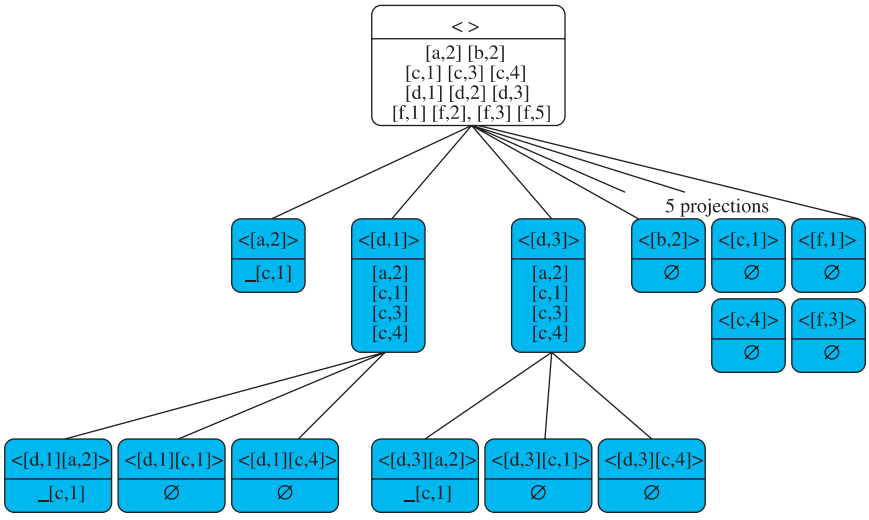

Fig. 17. Projection tree of PrefixSpan-Sampling.

```
Procedure PrefixSpan-All(α, DB, level)
begin
1.  F_α := Apriori-All(α, DB, level)
2.  F := {x|x := α · y for all y ∈ F_α}
3.  G̃_α^level := {x|x ∈ F_α^level ∧ x is a basic extended pattern }
4.  for each β ∈ G̃_α^level do {
5.     DB' := projection(α · β, DB)
6.     F̃ := PrefixSpan-All(α · β, DB', level)
7.     if F̃ ≠ ∅ then {
8.        Ḡ_α^level := Ḡ_α^level ∪ {x|x ∈ β's super quantity patterns with sampled quantities}
9.        F := F̃ ∪ F
10.    }
11. }
12. for each β ∈ Ḡ_α^level do {
13.    if β ≠ a basic extended pattern then {
14.       DB' := projection(α · β, DB)
15.       F̄ := PrefixSpan-All(α · β, DB', level)
16.    }
17.    if F̄ ≠ ∅ and ¬∃γ s.t. (γ ⊐ β ∧ P_γ = P_β) then {
18.       G_α^level := G_α^level ∪ {x|x ∈ β's grid cell}
19.       F := F̄ ∪ F
20.    }
21. }
22. for each β ∈ G_α^level do {
23.    DB' := projection(α · β, DB)
24.    F := PrefixSpan-All(α · β, DB', level)∪F
25. }
26. return F
end
```

Fig. 18. The PrefixSpan-All.

pattern and β has no proper supersequence pattern γ satisfying $P_\gamma = P_\beta$, we collect the patterns in *grid cell* of β into G_α^{level}.

Lastly, we perform projections on DB with the prefix of $\alpha \cdot \beta$ for every value of β in G_α^{level}. we invoke PrefixSpan-All recursively and obtain frequent patterns. These frequent patterns are unioned and returned at the end of PrefixSpan-All algorithm.

Let us examine how PrefixSpan-All works for a sequence database in Fig. 1 with the minimum support of 3. We again use level-by-level projection for this example and assume that we sample two intervals from possible quantities of an extended item in the second step of PrefixSpan-All. Among all 12 frequent items in the given database, first we compute the projections and recursively invoke PrefixSpan-All with the basic extended patterns, $\langle[a,2]\rangle$, $\langle[b,2]\rangle$, $\langle[c,1]\rangle$, $\langle[d,1]\rangle$ and $\langle[f,1]\rangle$. Then, since a has no more quantities to enumerate, there is neither projection nor recursive call for a. Likewise, since the results of the recursive calls with $\langle[b,2]\rangle$, $\langle[c,1]\rangle$ and $\langle[f,1]\rangle$ are empty respectively, we do not need to traverse with sampled patterns from these patterns. However, the recursive call with $\langle[d,1]\rangle$ has non-empty result, therefore we further perform projection and recursive call with sampled pattern $\langle[d,3]\rangle$ from $\langle[d,1]\rangle$. Although $\langle[d,1]\rangle$ is included in sampled patterns of d, since we already traversed with $\langle[d,1]\rangle$, we do not consider $\langle[d,1]\rangle$ in the second step. The recursive call with $[d,3]$ has non-empty result. However, now that $P_{\langle[d,1]\rangle}$ is the same as $P_{\langle[d,3]\rangle}$, we are sure that $P_{\langle[d,2]\rangle}$ is the same as $P_{\langle[d,1]\rangle}$ and $P_{\langle[d,3]\rangle}$ by Lemma 5.1.Thus, we do not need to invoke a recursive call, nor do we need to perform a projection with the prefix $\langle[d,2]\rangle$. Finally, since the grid cell of $\langle[d,3]\rangle$ is empty, the process of PrefixSpan-All is terminated now. The projection tree is presented in Fig. 19. The gray-colored nodes are traversed in the first step, and the black-colored node is explored in the second step. We do not need to perform the third step with this example. Note that the number of projections is reduced from 24 to 10.

The pseudo-projection proposed in PrefixSpan (Pei et al., 2001) loads database into main memory if it can fit into main memory and then use offsets in order to access projected database. The pseudo-projection method cannot scale well when the size of the database is large.

6. Experimental result

6.1. Algorithms

We conducted a comprehensive performance evaluation of the various algorithms and optimizations proposed in this paper. Specifically, we show the performance Figures of the eight algorithms illustrated in Table 2.

In implementations of PrefixSpan algorithms, whenever a projected database fits into main memory, we use pseudo-projection in Pei et al. (2001) to speed up the cost of projection. We also implemented both level-by-level and bi-level projections.

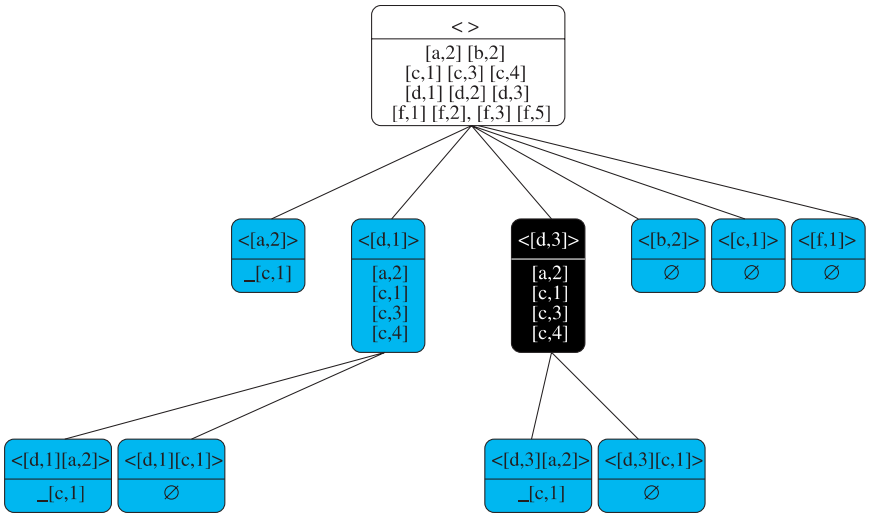

Fig. 19. Projection tree of PrefixSpan-All.

Table 2
Algorithms

Algorithms	Optimization
Apriori-QSP	(Basic handling of extended items)
Apriori-Hash	Hash filtering
Apriori-Sample	Quantity sampling
Apriori-All	Hash filtering + Quantity sampling
PrefixSpan-QSP	(Basic handling of extended items)
PrefixSpan-Hash	Hash filtering
PrefixSpan-Sample	Quantity sampling
PrefixSpan-All	Hash filtering + Quantity sampling

Table 3
Parameters

$\|D\|$	Number of data sequences
$\|S\|$	Average length of maximal potentially frequent Patterns
N_S	Number of maximal potentially frequent Patterns
N_I	Number of maximal potentially frequent Itemsets
N	Number of items
R	Repetition level
Q	Average value of quantities

Table 4
Synthetic data sets

Data set	$\|D\|$	$\|S\|$	N_S	N_I	N	R	Q
data.default	100K	10	1K	10K	100K	0	30
data.rept_0.1	100K	10	1K	10K	100K	0.1	30
data.rept_0.3	100K	10	1K	10K	100K	0.3	30
data.rept_0.5	100K	10	1K	10K	100K	0.5	30
data.exp.30	100K	4	5K	25K	10K	0	30
data.exp.50	100K	4	5K	25K	10K	0	50
data.exp.70	100K	4	5K	25K	10K	0	70
data.ncust_10	10K	10	1K	10K	100K	0	30
data.ncust_50	50K	10	1K	10K	100K	0	30
data.ncust_200	200K	10	1K	10K	100K	0	30
data.ncust_400	400K	10	1K	10K	100K	0	30

All experiments reported in this section were performed on a Pentium-4 2.4 GHz machine with 512 MB of main memory, running a Linux operating system. All of the methods were implemented using GCC compiler of Version 2.95.3.

6.2. Synthetic data sets

We modified a synthetic data set generater[2] for sequential pattern mining by IBM Almaden Research Center in order to add quantities according to the probability distribution. The modified generator takes the parameters shown in Table 3. The dataset parameter settings were summarized in Table 4. We set other parameter values not shown in Table 4 to the default values. For example, the average number of items was set to 2.5. We used exponential distribution to generate quantity values with the average quantity value given in the table.

6.2.1. Behavior of apriori algorithms

In Fig. 20, we compare the performance of variations of our apriori-style algorithms. We plot both the execution time and the number of candidates generated as we vary the minimum support threshold. We use log scale for y-axis in the graphs. As the different variations of our proposed techniques generate much smaller numbers of candidates, the corresponding execution times become smaller as well. The graph for the number of candidates shows that hash filtering and quantity sampling reduce the number of candidate patterns by an order of magnitude. It also presents that hash filtering reduces the number of candidate much more significantly than quantity sampling. Furthermore, the graphs for the number of enumerated candidate patterns conform to the Lemmas presented in Section 4. Our experimental result confirms the effectiveness of our

[2] http://www.almaden.ibm.com/cs/quest/syndata.html#assocSynData.

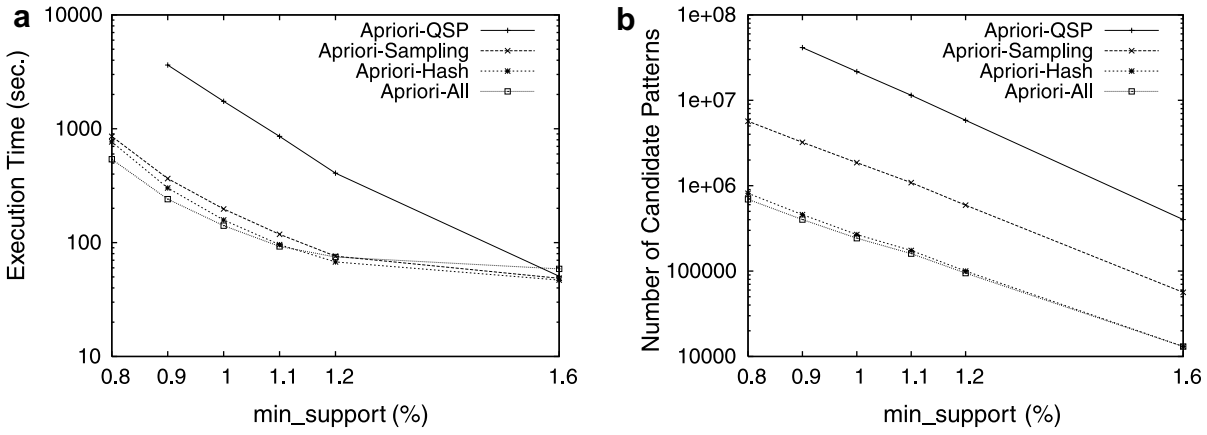

Fig. 20. Apriori-Style algorithms. (a) Execution time (s), and (b) Number of generated candidates.

proposed techniques for apriori style algorithms. Since Apriori-All is the best performer among all variations of apriori algorithms, we use Apriori-All as a representative for apriori algorithms in the rest of this section.

Varying the value of k: We forced Apriori-All algorithm to use a fixed value of k that is the number of sampled quantity values and plot the execution time and number of candidates generated for the minimum support threshold of 0.6% in Fig. 21. As we increase the number of sampled quantity values, the number of candidates and the execution time decreases initially. This shows that quantity sampling is having an immediate impact. However, as more and more intervals are created, there is a diminishing return, and eventually, adding more intervals only leads to extra overhead, causing an increase in the number of candidates and execution time. Moreover, notice that we plot the number of candidates produced as a horizonal line by Apriori-All (that computes the value of k that is the square root of the number of quantity values in each item). As shown in the figure, the lowest number of candidates and execution times are very close to the ones with Apriori-All that predicts the value of k as $\sqrt{L_i}$ as we illustrated in Section 4.4.

6.2.2. Behavior of PrefixSpan algorithms

Fig. 22 plots the execution time of PrefixSpan algorithms with both level-by-level and bi-level projections. The top half graphs present the execution times of level-by-level projection algorithms and the lower half are for bi-level projections. The figure illustrates that hash filtering and sampling improves PrefixSpan algorithms significantly regardless of level-by-level and bi-level projections. Thus, we use PrefixSpan-All as a representative for PrefixSpan algorithms in the rest of this section.

6.2.3. Comparison of apriori and prefix algorithms

We ran the Apriori-All, PrefixSpan (level-by-level) and PrefixSpan (bi-level) with the synthetic dataset that was generated with default values except the repetition level. The repetition level can vary between 0 and 1.0. As we increase the repetition level, it increases the number of candidates and frequent patterns. Furthermore, it increases the average length of the frequent patterns. We present the execution times for the repetition levels of 0.1, 0.3 and 0.5 in Fig. 23. When repetition level was 0.1, Apriori-All was the winner, and PrefixSpan-All (bi-level) was the second best.As we increase the repetition level more and more, Apriori-All becomes the worst performer for small mini-

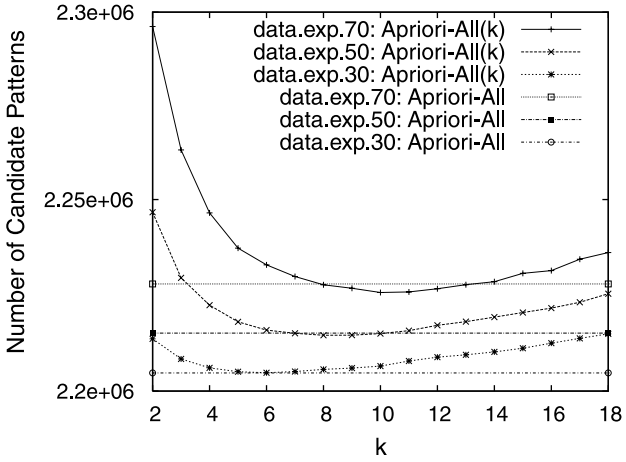

Fig. 21. Number of sampled quantities.

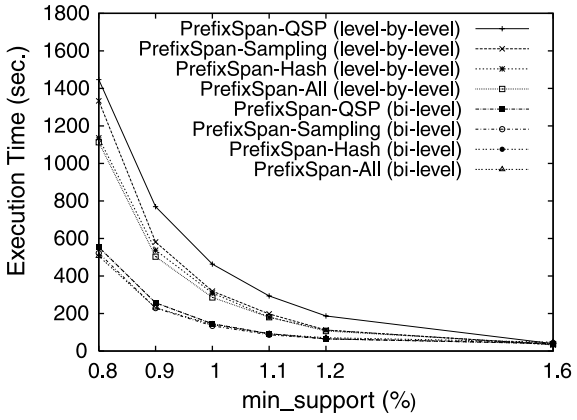

Fig. 22. PrefixSpan algorithms.

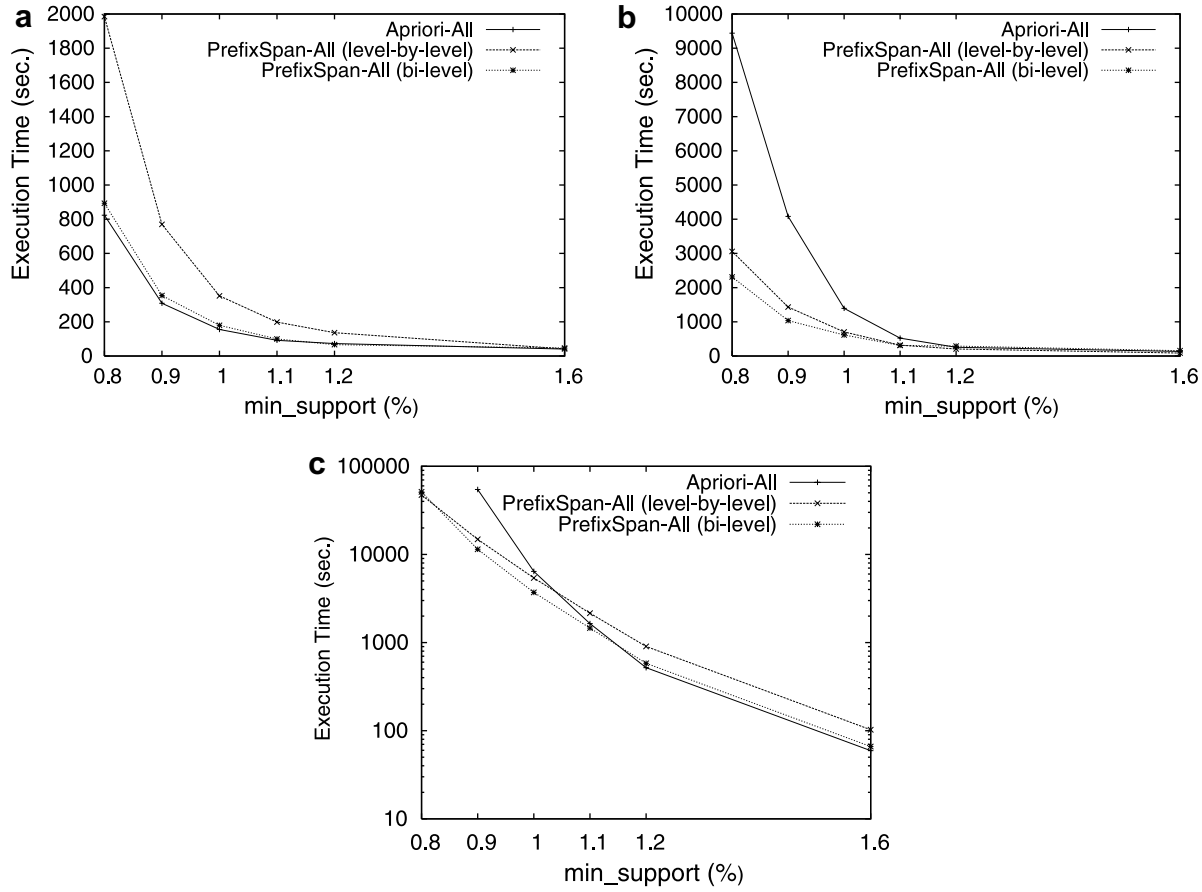

Fig. 23. Comparison of Apriori-All and PrefixSpan-All. (a) Repetition level = 0.1, (b) Repetition level = 0.3, and (c) Repetition level = 0.5.

mum support among the three algorithms. Furthermore, for smaller minimum support, PrefixSpan-All (level-by-level) becomes the best performer.

Scalability: In Fig. 24, we vary the number of sequences for the minimum support threshold of 1.0% and plot the execution times of PrefixSpan-All (level-by-level), PrefixSpan-All (bi-level) and Apriori-All.The graph illustrates that all three algorithms have linear scalability that is desirable for data mining algorithms.

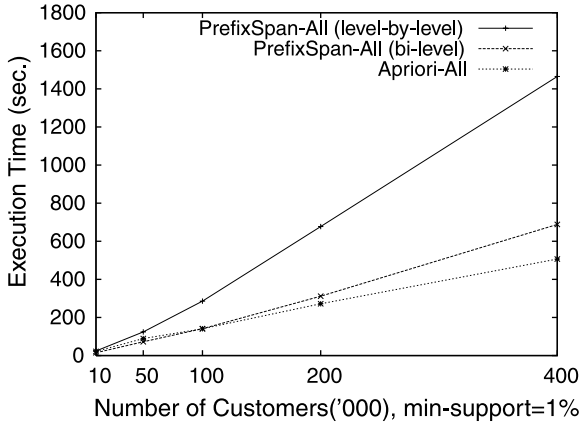

Fig. 24. Number of sequences.

6.3. Real-life data set

For our real-life data experiments, we used a market basket data for the period of six months obtained from a Korean online marketplace company. As we discussed in the introduction, we reduced the dataset to include only the items which were purchased in multiple quantities by at least a customer and only the transactions in which there is at least an item with multiple quantity. In the reduced dataset, the number of customers and the number of items are 559,056 and 157,849 respectively. The average size of sequences is 3.0 and the average quantity of items is 2.9.

Some sample patterns were already given in Section 1.1. Thus, we focus only on performance issues here. We present the execution times of the algorithms in Fig. 25, as the minimum-support is decreased from 0.04% to 0.01%. The shape of the result graph is similar with that of the synthetic datasets.

6.3.1. Behavior of apriori algorithms

As shown in Fig. 25a, Apriori-QSP shows the worst performance. Apriori-Hash is the best and slightly faster than Apriori-All. The reason is that there are a huge number of items compared to the size of the dataset. Since the dataset has 559,056 sequences with the average size 3, an item of

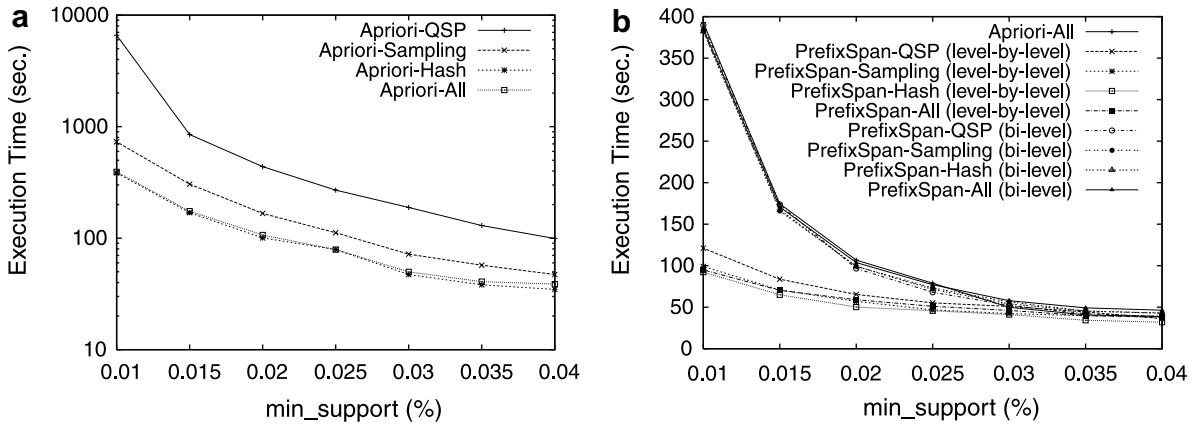

Fig. 25. Execution time for real-life dataset: (a) Apriori and (b) PrefixSpan.

157,849 items appears roughly ten times on the average, and thus only few itemsets can be frequent. This results that the hash-filtering method is very effective. For example, when the minimum-support is 0.01%, 24,613,518 candidates among 24,613,876 candidates with length two are removed through the filtering step by Apriori-Hash. However, Apriori-All additionally performs the sampling step after the filtering step. Since the candidates were mostly removed in the filtering step, the number of candidates reduced in the sampling step is very small, and the saving obtained by reducing the number of candidates in the sampling step is not larger than the time spent by the sampling step itself. Therefore, Apriori-Hash is a little bit faster than Apriori-All in the experiments. In conclusion, the Apriori algorithms using our techniques run faster than Apriori-QSP by an order of magnitude.

6.3.2. Behavior of prefixspan algorithms

In Fig. 25b, PrefixSpan algorithms with level-by-level show better performance than PrefixSpan algorithms with bi-level. It is also because there are a large number of items compared to the size of the dataset, and only few short itemsets can be frequent. Since the length of 95% of patterns is one, the execution time of PrefixSpan algorithms with bi-level mainly depends on the number of candidates while that of level-by-level algorithm depends on the number of projections. In general, performing a projection is more expensive in execution time than checking a candidate. However, in the experiments, since the number of projections by PrefixSpan algorithms with level-by-level is much smaller than that of candidates by PrefixSpan algorithms with bi-level, PrefixSpan algorithms with level-by-level are faster than PrefixSpan algorithms with bi-level. For example, when the minimum-support is 0.01%, the number of projections by Prefix-All with level-by-level is 5,853 while the number of candidates by Prefix-All with bi-level is 24,619,887.

Among PrefixSpan algorithms with level-by-level, PrefixSpan-QSP is the slowest, and other PrefixSpan algorithms with level-by-level show similar performance. PrefixSpan-Hash with level-by-level is the fastest because of the same reason why Apriori-Hash is faster than Apriori-All. All PrefixSpan algorithms with bi-level show nearly the same performance as Apriori-All. Since the length of almost all patterns is at most two, the performance of all PrefixSpan algorithms with bi-level depends on that of Apriori-All.

7. Conclusion

Discovering sequential patterns is an important problem for many applications. For these applications, quantitative attributes are often recorded in the data, which are ignored by existing algorithms. We studied the problem of mining sequential patterns with quantities and proposed naive extensions to Apriori and PrefixSpan algorithms that are introduced for traditional sequential pattern mining. However, these extensions are inefficient, as they may enumerate the search space blindly. To alleviate the situation, we proposed hash filtering and quantity sampling techniques that improve the performance of the naive extended algorithms significantly. Experimental results clearly demonstrate that compared with the naive extensions, these schemes improve the performance substantially.

References

Aggarwal, C., Agarwal, R., Prasad, V.V.V., 2000. Depth-first generation of long patterns. In: International Conference on Knowledge Discovery and Data Mining (KDD), Boston, MA.

Agrawal, Rakesh, Srikant, Ramakrishnan, 1994. Fast algorithms for mining association rules. In: Proceedings of the VLDB Conference, Santiago, Chile, September.

Agrawal, Rakesh, Srikant, Ramakrishnan, 1995. Mining sequential patterns. In: International Conference on Data Engineering, Taipei, Taiwan, March.

Agrawal, Rakesh, Srikant, Ramakrishnan, 1996. Mining sequential patterns: generalizations and performance improvements. In: International Conference on Extending Database Technology (EDBT), Avignon, France, March.

Bayardo, R.J., 1998. Fast subsequence matching in time-series databases. In: Proceedings of the ACM SIGMOD Conference on Management of Data, June.

Burdick, D., Calimlim, M., Gehrke, J., 2001. MAFIA: a maximal frequent itemset algorithm for transactional databases. In: IEEE International Conference on Data Engineering, Heidelberg, Germany.

Cheng, Hong, Yan, Xifeng, Han, Jiawei, 2004. Incspan: incremental mining of sequential patterns in large database. In: KDD, pp. 527–532.

Fukuda, T., Morimoto, Y., Morishita, S., Tokuyama, T., 1996. Mining optimized association rules for numeric attributes. In: Proceedings of the ACM Symposium on Principles of Database Systems, June.

Garofalakis, Minos, Rastogi, Rajeev, Shim, Kyuseok, 1999. SPIRIT: Sequential pattern mining with regular expression constraints. In: Proceedings of the VLDB Conference, Edinburgh, UK, September.

Gouda, K., Zaki, M.J., 2001. Efficiently mining maximal frequent itemsets. In: IEEE International Conference on Data Mining, San Jose, CA, November.

Han, J., Pei, J. Yin, Y., 2000. Mining frequent patterns without candidate generation. In: Proceedings of the ACM SIGMOD Conference on Management of Data.

Mannila, Heikki, Toivonen, Hannu, Inkeri Verkamo, A., 1997. Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery 1 (3).

Miller, R., Yang, Y., 1997. Association rules over interval data. In: Proceedings of the ACM SIGMOD Conference on Management of Data.

Park, Jong Soo, Chen, Ming-Syan, Yu, Philip S., 1995. An effective hash based algorithm for mining association rules. In: Proceedings of the ACM-SIGMOD Conference on Management of Data, San Jose, CA.

Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L., 1999. Discovering frequent closed itemsets for association rules. In: International Conference on Database Thoery, Jerusalem, Israel, January.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.-C., 2001. Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth. In: International Conference on Knowledge Discovery in Databases and Data Mining (KDD), Heidelberg, Germany.

Pei, Jian, Han, Jiawei, Wang, Wei, 2002. Mining sequential patterns with constraints in large databases. In: CIKM, pp. 18–25.

Srikant, Ramakrishnan, Agrawal, Rakesh, 1995. Mining generalized association rules. In: Proceedings of the VLDB Conference, Zurich, Switzerland.

Srikant, Ramakrishnan, Agrawal, Rakesh, 1996. Mining quantitative association rules in large relational tables. In: Proceedings of the ACM SIGMOD Conference on Management of Data, June.

Yan, Xifeng, Han, Jiawei, Afshar, Ramin, 2003. Clospan: mining closed sequential patterns in large databases. In: SDM.

Zaki, Mohammed Javeed, 2001. SPADE: an efficient algorithm for mining frequent sequences. Machine Learning 42 (1/2).