A GSP-based Efficient Algorithm for Mining Frequent Sequences

Article · August 2002

Source: CiteSeer

CITATIONS
32

READS
148

4 authors, including:

Ben Kao
The University of Hong Kong
142 PUBLICATIONS **3,616** CITATIONS
SEE PROFILE

Chi Lap Yip
The University of Hong Kong
40 PUBLICATIONS **425** CITATIONS
SEE PROFILE

A GSP-based Efficient Algorithm for Mining Frequent Sequences

Minghua Zhang Ben Kao Chi-Lap Yip David Cheung

Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong.
{*mhzhang, kao, clyip, dcheung*}*@csis.hku.hk*

Abstract

This paper studies the problem of mining frequent sequences in transactional databases. In [3], Agrawal and Srikant proposed the GSP algorithm for extracting frequently occurring sequences. GSP is an iterative algorithm. It scans the database a number of times depending on the length of the longest frequent sequences in the database. The I/O cost is thus substantial if the database contains very long frequent sequences. In this paper, we extend the candidate generating function used by GSP and propose a new two-stage algorithm MFS. Our algorithm first mines a sample of the database to obtain a rough estimate of the frequent sequences and then refines the solution. Experiment results show that MFS saves I/O cost significantly compared with GSP.

keywords: data mining, sequence, GSP, MFS

1 Introduction

One of the many data mining problems is mining frequent sequences from transactional databases. The goal is to discover frequent sequences of events. The problem was first introduced by Agrawal and Srikant [1]. In their model, a database is a collection of transactions. Each transaction is a set of items (or an itemset) and is associated with a customer ID and a time ID. If one groups the transactions by their customer IDs, and then sorts the transactions of each group by their time IDs in increasing value, the database is transformed into a number of customer sequences. Each customer sequence shows the order of transactions a customer has conducted. Roughly speaking, the problem of mining frequent sequences is to discover "subsequences" (of itemsets) that occur frequently enough among all the customer sequences.

Based on this model, several algorithms have been proposed. Among them, GSP [3] is a very efficient one. GSP is a multi-phase iterative algorithm. It scans the database a number of times. During the i-th iteration, frequent sequences of *length* i are discovered. The number of database scans GSP requires is thus determined by the length of the longest frequent sequences in the database. If the database is huge and if it contains very long frequent sequences, the I/O cost of GSP is substantial.

In this paper, we propose a new algorithm called MFS, which reduces the I/O requirement of GSP significantly. MFS achieves the goal by checking candidates (sequences that are potentially frequent) of various lengths in each database scan. The core of MFS is its candidate generation function — MGen, which is a modification of GSP_Gen (the candidate generation function of GSP). Through experiment, we show that, in many cases, MFS reduces the I/O cost of GSP significantly.

```
1    Algorithm GSP(D, ρ_s, I)
2      C_1 := {⟨{i}⟩|i ∈ I}
3      Scan D to get support of every sequence in C_1
4      L_1 := {s|s ∈ C_1, sup(s) >= ρ_s}
5      i := 1
6      while (L_i ≠ ∅)
7        C_{i+1} := GSP_Gen(L_i)
8        Scan D to get support of every sequence in C_{i+1}
9        L_{i+1} := {s|s ∈ C_{i+1}, sup(s) >= ρ_s}
10       i := i + 1
11     Return L_1 ∪ L_2 ∪ ... L_{i-1}
```

Figure 1: Algorithm GSP

2 Problem Definition

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called items. An itemset X is a set of items (hence, $X \subseteq I$). A sequence $s = \langle t_1, t_2, \ldots, t_n \rangle$ is an ordered set of itemsets. The length of s is defined as the number of items contained in s. (If an item occurs several times in different itemsets of a sequence, the item is counted for each occurrence.) We use $|s|$ to represent the length of s. For example, if $s = \langle \{1\}, \{2, 3\}, \{1, 4\} \rangle$, then $|s| = 5$. A sequence of length k is called a k-sequence.

Consider two sequences $s_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $s_2 = \langle b_1, b_2, \ldots, b_l \rangle$. We say that s_2 is a subsequence of s_1 if there exist integers $j_1, j_2, \ldots, j_l$, such that $1 \leq j_1 < j_2 < \ldots < j_l \leq n$ and $b_1 \subseteq a_{j_1}, b_2 \subseteq a_{j_2}, \ldots, b_l \subseteq a_{j_l}$. We represent this relationship by $s_2 \sqsubseteq s_1$.

Given a sequence set V and a sequence s, if there exists a sequence $s' \in V$ such that $s \sqsubseteq s'$, we write $s \vdash V$.

Given a sequence set V, a sequence $s \in V$ is *maximal* if s is not a subsequence of any other sequence in V. That is, s is maximal if $\nexists s' | s' \in V \wedge s' \neq s \wedge s \sqsubseteq s'$.

A database $\mathcal{D}$ consists of a number of sequences. The support of a sequence s is defined as the fraction of all sequences in $\mathcal{D}$ that contain s. We use $sup(s)$ to denote the support of s. If the support of s is no less than a user specified support threshold ρ_s, s is a frequent sequence. The problem of mining frequent sequences is to find all *maximal* frequent sequences given a sequence database $\mathcal{D}$.

3 GSP

In this section we briefly review the GSP algorithm (Figures 1). We use C_i to denote the set of length-i candidate sequences (i.e., those sequences that are potentially frequent). We use L_i to denote the set of all length-i frequent sequences.

GSP takes 3 parameters: a database D, a support threshold ρ_s, and a set of *itemsets* I. GSP first takes each item i in I to form a length-1 candidate sequence $\langle \{i\} \rangle$. GSP then checks all candidate 1-sequences to get L_1. GSP then generates C_2 from L_1 using GSP_Gen. The database is then scanned to extract all frequent length-2 sequences (L_2). This candidate-generation-verification procedure is repeated until no more frequent sequences are found. In general, GSP_Gen generates length-$(i+1)$ candidate sequences by considering all length-i frequent sequences. Due to space limitation, readers are referred to [3] for details. We remark that if the length of the longest frequent sequence in the database is n, GSP would have to scan the

```
1    Algorithm MFS(D, ρ_s, I, S_est)
2      MFSS := ∅
3      CandidateSet := {⟨{i}⟩|i ∈ I} ∪ {s|s ⊢ S_est, |s| > 1}
4      Scan D to get support of every sequence in CandidateSet
5      NewFrequentSequences := {s|s ∈ CandidateSet, sup(s) >= ρ_s}
6      AlreadyCounted := {s|s ⊢ S_est, |s| > 1}
7      Iteration := 2
8      while (NewFrequentSequences ≠ ∅)
9        //Max(S) returns the set of all maximal sequences is S
10       MFSS := Max(MFSS ∪NewFrequentSequences)
11       CandidateSet := MGen(MFSS, Iteration, AlreadyCounted)
12       Scan D to get support of every sequence in CandidateSet
13       NewFrequentSequences := {s|s ∈ CandidateSet, sup(s) >= ρ_s}
14       Iteration := Iteration+1
15     Return MFSS
```

Figure 2: Algorithm MFS

database at least n times.

4 MFS and MGen

We propose the algorithm MFS(Figure 2) for mining frequent sequences in a transaction database that has a smaller I/O cost when compared with GSP. With GSP, every database scan discovers frequent sequences of the same length. MFS, on the other hand, takes a *sucessive refinement* approach. It first computes a rough estimate, S_{est}, of the set of all frequent sequences. If the database is regularly updated and that frequent sequences are mined periodically, then the result obtained from a previous mining exercise can be used as the estimate. If such an estimate is not readily available, we could mine a small sample (let's say 10%) of the database using GSP to obtain S_{est}. MFS uses the MGen function (Figure 3) for candidate sequence generation. The difference between MGen and GSP_Gen is that while GSP_Gen generates C_{i+1} from L_i (for some i), MGen takes a set of frequent sequences of various lengths to generate a set of candidate sequences of various lengths. Given S_{est}, MFS first scans the database once to determine which sequences in S_{est} are in fact frequent. The *maximal* of these frequent sequences are put into a set *MFSS*. If we apply MGen to *MFSS*, we will generate a set of candidate sequences of various lengths. These candidates are checked against the database to determine which are frequent. The information obtained is then used to *refine MFSS*. The process stops when no more new frequent sequences can be discovered in an iteration. Note that the MFS approach allows the supports of long sequences be checked early. This is the major source of efficiency improvement over GSP.

We can prove that MFS discovers the same set of frequent sequences as does GSP. Hence MFS is correct. We can also prove that the set of candidate sequences ever generated by MFS is a subset of those generated by GSP. Hence, MFS does not generate any unnecessary candidates and waste resources for counting their supports. Due to space limitation, the proofs are not included in this extended abstract.

```
1   Function MGen(MFSS, Iteration, AlreadyCounted)
2     CandidateSet := ∅
3     for each pair of s1, s2 ∈ MFSS such that |s1| > Iteration-2, |s2| > Iteration-2
      and that s1, s2 share at least one common subsequence of length ≥ Iteration-2
4       for each common subsequence s of s1, s2 such that |s| ≥ Iteration-2
5         NewCandidate := {⟨i1, s, i2⟩[1] | ⟨i1, s⟩ ⊑ s1, ⟨s, i2⟩ ⊑ s2}
6         CandidateSet := CandidateSet ∪ NewCandidate
7         NewCandidate := {⟨i2, s, i1⟩ | ⟨i2, s⟩ ⊑ s2, ⟨s, i1⟩ ⊑ s1}
8         CandidateSet := CandidateSet ∪ NewCandidate
9     for each sequence s ∈ CandidateSet
10      if (s ⊢ MFSS) delete s from CandidateSet
11      if s ∈ AlreadyCounted delete s from CandidateSet
12      for any subsequence s' of s with length |s| − 1
13        if (s' ⊬ MFSS) delete s from CandidateSet
14    AlreadyCounted := AlreadyCounted ∪CandidateSet
15    for each sequence s ∈ AlreadyCounted
16      if (|s| = Iteration) delete s from AlreadyCounted
17    Return CandidateSet
```

Figure 3: Function `MGen`

5 Performance

We performed a number of experiments comparing the performance of `GSP` and `MFS`. Our goals are to study how much I/O cost `MFS` could save, and how effective sampling is in discovering an initial estimate of the set of frequent sequences required by `MFS`. In this section, we present some representative results from our experiments.

We used synthetic data as the test databases. The generator is obtained from the IBM Quest data mining project. Readers are referred to [2] for the details of the data generator.

One way to obtain S_{est} is to apply `GSP` on a database sample. We studied this sampling approach in our experiments. We first applied `GSP` on our synthetic dataset to obtain the number of I/O passes it required. Then a random sample of the database was drawn on which `GSP` was applied to obtain S_{est}. We then executed `MFS` using the S_{est} found. This exercise was repeated a number of times, each with a different random sample.

The average amount of I/O cost required by `MFS` was noted. This I/O cost includes the cost of mining the sample (using `GSP`). Besides I/O cost, we also compare the number of candidate sequences whose supports are counted by the two algorithms. The number of candidates `MFS` counted is measured by the following formula:

$$(\#\text{ of candidates counted to obtain } S_{est}) \times (\text{sample size}) + (\#\text{ of candidates counted in } \texttt{MFS}).$$

Similarly, the amount of CPU time `MFS` took includes the CPU time for mining the sample.

The result of the experiment (with different sample sizes) is shown in Table 1. Note that the average numbers of candidates counted and the average CPU cost of `MFS` are shown in relative quantities (with those of `GSP` set to 1).

[1] $\langle i_1, s, i_2 \rangle$ means the sequence obtained by adding item i_1 to the beginning of sequence s and adding item i_2 to the end of s. Whether i_1 is in a separate itemset in the result sequence is determined by whether i_1 is in a separate itemset in s_1; similarly whether i_2 is in a separate itemset in the result sequence is determined by whether i_2 is in a separate itemset in s_2.

sample size	1/128	1/64	1/32	1/16	1/8	1/4	1/2	0(GSP)
avg. I/O cost	7.895	7.702	7.249	6.760	6.959	7.516	9.344	10
avg. # of cand.	1.130	1.070	1.072	1.084	1.139	1.257	1.510	1
avg. CPU cost	1.576	1.303	1.249	1.198	1.224	1.319	1.544	1

Table 1: Performance of `MFS` vs. sample size ($\rho_s = 0.75\%$)

From Table 1, we see that `MFS` needed fewer I/O passes than `GSP` (10 passes). As the sample size increases, the estimate S_{est} becomes closer to the real set of frequent sequences. Fewer I/O passes are thus needed for `MFS` to refine *MFSS* towards the final result. This accounts for the drop of I/O cost from 7.895 passes to 6.760 passes as the sample size increases from 1/128 to 1/16. As the sample size increases further, however, the I/O cost of mining the sample becomes substantial. The benefit obtained by having a "more accurate" S_{est} is outweighted by the penalty of mining the sample. Hence, the overall I/O cost increases as the sample size increases from 1/16 to 1/2. Also, as the sample size increases, more work is spent on candidate counting, and the CPU cost increases.

In our study, we find that the performance of `MFS` can be further improved by using a slightly smaller support threshold (ρ_{s_sample}) to mine the sample when computing an estimate S_{est}. Through experiments, we find that such a trick can further reduce the I/O requirement by more than 30%. Due to space limitation, we omit the discussion of such a modification to `MFS` in this extended abstract.

6 Conclusion

In this paper we proposed a new I/O efficient algorithm `MFS` to solve the problem of mining frequent sequences. A new candidate generation method `MGen` was proposed which can generate candidate sequences of multiple lengths given a set of suggested frequent sequences.

We performed experiments to compare the performance of `MFS` and `GSP`. We showed that mining a small sample of the database leads to a good S_{est}, and that `MFS` saves I/O passes significantly. By using a smaller support threshold (ρ_{s_sample}) in mining the sample, `MFS` can outperform `GSP` by a wide margin. The I/O saving is obtained, however, at a mild CPU cost.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[2] http://www.almaden.ibm.com/cs/quest/.

[3] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th Conference on Extending Database Technology (EDBT)*, Avignion, France, March 1996.