Mining Sequential Patterns

Rakesh Agrawal Ramakrishnan Srikant*

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We are given a large database of customer transactions, where each transaction consists of customer-id, transaction time, and the items bought in the transaction. We introduce the problem of mining sequential patterns over such databases. We present three algorithms to solve this problem, and empirically evaluate their performance using synthetic data. Two of the proposed algorithms, AprioriSome and AprioriAll, have comparable performance, albeit AprioriSome performs a little better when the minimum number of customers that must support a sequential pattern is low. Scale-up experiments show that both AprioriSome and AprioriAll scale linearly with the number of customer transactions. They also have excellent scale-up properties with respect to the number of transactions per customer and the number of items in a transaction.

1 Introduction

Database mining is motivated by the decision support problem faced by most large retail organizations. Progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data, referred to as the *basket* data. A record in such data typically consists of the transaction date and the items bought in the transaction. Very often, data records also contain customer-id, particularly when the purchase has been made using a credit card or a frequent-buyer card. Catalog companies also collect such data using the orders they receive.

We introduce the problem of mining *sequential* patterns over this data. An example of such a pattern is that customers typically rent "Star Wars", then "Empire Strikes Back", and then "Return of the Jedi". Note that these rentals need not be consecutive. Customers who rent some other videos in between also support this sequential pattern. Elements of a sequential pattern need not be simple items. "Fitted Sheet and flat sheet and pillow cases", followed by "comforter", followed by "drapes and ruffles" is an example of a sequential pattern in which the elements are sets of items.

Problem Statement We are given a database $\mathcal{D}$ of customer transactions. Each transaction consists of the following fields: customer-id, transaction-time, and the items purchased in the transaction. No customer has more than one transaction with the same transaction-time. We do not consider quantities of items bought in a transaction: each item is a binary variable representing whether an item was bought or not.

An *itemset* is a non-empty set of items. A *sequence* is an ordered list of itemsets. Without loss of generality, we assume that the set of items is mapped to a set of contiguous integers. We denote an itemset i by $(i_1 i_2 ... i_m)$, where i_j is an item. We denote a sequence s by $\langle s_1 s_2 ... s_n \rangle$, where s_j is an itemset.

A sequence $\langle a_1 a_2 ... a_n \rangle$ *is contained in* another sequence $\langle b_1 b_2 ... b_m \rangle$ if there exist integers $i_1 < i_2 < ... < i_n$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, ..., $a_n \subseteq b_{i_n}$. For example, the sequence $\langle (3)\ (4\ 5)\ (8) \rangle$ is contained in $\langle (7)\ (3\ 8)\ (9)\ (4\ 5\ 6)\ (8) \rangle$, since $(3) \subseteq (3\ 8)$, $(4\ 5) \subseteq (4\ 5\ 6)$ and $(8) \subseteq (8)$. However, the sequence $\langle (3)\ (5) \rangle$ is not contained in $\langle (3\ 5) \rangle$ (and vice versa). The former represents items 3 and 5 being bought one after the other, while the latter represents items 3 and 5 being bought together. In a set of sequences, a sequence s is *maximal* if s is not contained in any other sequence.

All the transactions of a customer can together be viewed as a sequence, where each transaction corresponds to a set of items, and the list of

*Also Department of Computer Science, University of Wisconsin, Madison.

Customer Id	TransactionTime	Items Bought
1	June 25 '93	30
1	June 30 '93	90
2	June 10 '93	10, 20
2	June 15 '93	30
2	June 20 '93	40, 60, 70
3	June 25 '93	30, 50, 70
4	June 25 '93	30
4	June 30 '93	40, 70
4	July 25 '93	90
5	June 12 '93	90

Figure 1: Database Sorted by Customer Id and Transaction Time

Customer Id	Customer Sequence
1	$\langle$ (30) (90) $\rangle$
2	$\langle$ (10 20) (30) (40 60 70) $\rangle$
3	$\langle$ (30 50 70) $\rangle$
4	$\langle$ (30) (40 70) (90) $\rangle$
5	$\langle$ (90) $\rangle$

Figure 2: Customer-Sequence Version of the Database

transactions, ordered by increasing transaction-time, corresponds to a sequence. We call such a sequence a *customer-sequence*. Formally, let the transactions of a customer, ordered by increasing transaction-time, be T_1, T_2, ..., T_n. Let the set of items in T_i be denoted by itemset(T_i). The customer-sequence for this customer is the sequence $\langle$ itemset(T_1) itemset(T_2) ... itemset(T_n) $\rangle$.

A customer *supports* a sequence s if s is contained in the customer-sequence for this customer. The *support for a sequence* is defined as the fraction of total customers who support this sequence.

Given a database $\mathcal{D}$ of customer transactions, the problem of mining sequential patterns is to find the maximal sequences among all sequences that have a certain user-specified minimum support. Each such maximal sequence represents a *sequential pattern*.

We call a sequence satisfying the minimum support constraint a *large* sequence.

Example Consider the database shown in Fig. 1. (This database has been sorted on customer-id and transaction-time.) Fig. 2 shows this database expressed as a set of customer sequences.

With minimum support set to 25%, i.e., a minimum support of 2 customers, two sequences: $\langle$ (30) (90) $\rangle$ and $\langle$ (30) (40 70) $\rangle$ are maximal among those satisfying the support constraint, and are the desired sequential patterns. The sequential pattern $\langle$ (30) (90) $\rangle$ is supported by customers 1 and 4. Customer 4 buys items (40 70) in between items 30 and 90, but supports the pattern $\langle$ (30) (90) $\rangle$ since we are looking for patterns that are not necessarily contiguous. The sequential pattern $\langle$ 30 (40 70) $\rangle$ is supported by customers 2 and 4. Customer 2 buys 60 along with 40 and 70, but supports this pattern since (40 70) is a subset of (40 60 70).

Sequential Patterns with support > 25%
$\langle$ (30) (90) $\rangle$
$\langle$ (30) (40 70) $\rangle$

Figure 3: The answer set

An example of a sequence that does not have minimum support is the sequence $\langle$ (10 20) (30) $\rangle$, which is only supported by customer 2. The sequences $\langle$ (30) $\rangle$, $\langle$ (40) $\rangle$, $\langle$ (70) $\rangle$, $\langle$ (90) $\rangle$, $\langle$ (30) (40) $\rangle$, $\langle$ (30) (70) $\rangle$ and $\langle$ (40 70) $\rangle$, though having minimum support, are not in the answer because they are not maximal.

Related Work In [1], the problem of discovering "what items are bought together in a transaction" over basket data was introduced. While related, the problem of finding what items are bought together is concerned with finding intra-transaction patterns, whereas the problem of finding sequential patterns is concerned with inter-transaction patterns. A pattern in the first problem consists of an unordered set of items whereas a pattern in the latter case is an ordered list of sets of items.

Discovering patterns in sequences of events has been an area of active research in AI (see, for example, [6]). However, the focus in this body of work is on discovering the rule underlying the generation of a given sequence in order to be able to predict a plausible sequence continuation (e.g. the rule to predict what number will come next, given a sequence of numbers). We on the hand are interested in finding all common patterns embedded in a database of sequences of sets of events (items).

Our problem is related to the problem of finding text subsequences that match a given regular expression (*c.f.* the UNIX grep utility). There also has been work on finding text subsequences that approximately match a given string (e.g. [5] [12]). These techniques are oriented toward finding matches for one pattern. In our problem, the difficulty is in figuring out what patterns to try and then efficiently finding out which ones are contained in a customer sequence.

Techniques based on multiple alignment [11] have been proposed to find entire text sequences that are similar. There also has been work to find locally similar subsequences [4] [8] [9]. However, as pointed out in [10], these techniques apply when the discovered patterns consist of consecutive characters or multiple lists of consecutive characters separated by a fixed length of noise characters.

Closest to our problem is the problem formulation in [10] in the context of discovering similarities in a database of genetic sequences. The patterns they wish to discover are subsequences made up of consecutive characters separated by a variable number of noise characters. A sequence in our problem consists of list of sets of characters (items), rather than being simply a list of characters. Thus, an element of the sequential pattern we discover can be a set of characters (items), rather than being simply a character. Our solution approach is entirely different. The solution in [10] is not guaranteed to be complete, whereas we guarantee that we have discovered all sequential patterns of interest that are present in a specified minimum number of sequences. The algorithm in [10] is a main memory algorithm based on generalized suffix tree [7] and was tested against a database of 150 sequences (although the paper does contain some hints on how they might extend their approach to handle larger databases). Our solution is targeted at millions of customer sequences.

Organization of the Paper We solve the problem of finding all sequential patterns in five phases: i) sort phase, ii) litemset phase, iii) transformation phase, iv) sequence phase, and v) maximal phase. Section 2 gives this problem decomposition. Section 3 examines the sequence phase in detail and presents algorithms for this phase. We empirically evaluate the performance of these algorithms and study their scale-up properties in Section 4. We conclude with a summary and directions for future work in Section 5.

2 Finding Sequential Patterns

Terminology The *length* of a sequence is the number of itemsets in the sequence. A sequence of length k is called a k-sequence. The sequence formed by the concatenation of two sequences x and y is denoted as $x.y$.

The support for an itemset i is defined as the fraction of customers who bought the items in i in a single transaction. Thus the itemset i and the 1-sequence $\langle i \rangle$ have the same support. An itemset with minimum support is called a large itemset or *litemset*. Note that each itemset in a large sequence must have minimum support. Hence, any large sequence must be a list of litemsets.

Large Itemsets	Mapped To
(30)	1
(40)	2
(70)	3
(40 70)	4
(90)	5

Figure 4: Large Itemsets

2.1 The Algorithm

We split the problem of mining sequential patterns into the following phases:

1. Sort Phase. The database ($\mathcal{D}$) is sorted, with customer-id as the major key and transaction-time as the minor key. This step implicitly converts the original transaction database into a database of customer sequences.

2. Litemset Phase. In this phase we find the set of all litemsets L. We are also simultaneously finding the set of all large 1-sequences, since this set is just $\{\langle l \rangle \mid l \in L\}$.

The problem of finding large itemsets in a given set of customer transactions, albeit with a slightly different definition of support, has been considered in [1] [2]. In these papers, the support for an itemset has been defined as the fraction of transactions in which an itemset is present, whereas in the sequential pattern finding problem, the support is the fraction of customers who bought the itemset in any one of their possibly many transactions. It is straightforward to adapt any of the algorithms in [2] to find litemsets. The main difference is that the support count should be incremented only once per customer even if the customer buys the same set of items in two different transactions.

The set of litemsets is mapped to a set of contiguous integers. In the example database given in Fig. 1, the large itemsets are (30), (40), (70), (40 70) and (90). A possible mapping for this set is shown in Fig.4. The reason for this mapping is that by treating litemsets as single entities, we can compare two litemsets for equality in constant time, and reduce the time required to check if a sequence is contained in a customer sequence.

3. Transformation Phase. As we will see in Section 3, we need to repeatedly determine which of a given set of large sequences are contained in a customer sequence. To make this test fast, we transform each customer sequence into an alternative representation.

In a transformed customer sequence, each transaction is replaced by the set of all litemsets contained in that transaction. If a transaction does not contain any litemset, it is not retained in the transformed sequence. If a customer sequence does not contain any litemset, this sequence is dropped from the transformed database. However, it still contributes to the count of total number of customers. A customer sequence is now represented by a list of sets of litemsets. Each set of litemsets is represented by $\{l_1, l_2, \ldots, l_n\}$, where l_i is a litemset.

This transformed database is called $\mathcal{D}_T$. Depending on the disk availability, we can physically create this transformed database, or this transformation can be done on-the-fly, as we read each customer sequence during a pass. (In our experiments, we physically created the transformed database.)

The transformation of the database in Fig. 2 is shown in Fig. 5. For example, during the transformation of the customer sequence with Id 2, the transaction (10 20) is dropped because it does not contain any litemset and the transaction (40 60 70) is replaced by the set of litemsets {(40), (70), (40 70)}.

4. Sequence Phase. Use the set of litemsets to find the desired sequences. Algorithms for this phase are described in Section 3.

5. Maximal Phase. Find the maximal sequences among the set of large sequences. In some algorithms in Section 3, this phase is combined with the sequence phase to reduce the time wasted in counting non-maximal sequences.

Having found the set of all large sequences S in the sequence phase, the following algorithm can be used for finding maximal sequences. Let the length of the longest sequence be n. Then,

```
for ( k = n; k > 1; k-- ) do
  foreach k-sequence s_k do
    Delete from S all subsequences of s_k
```

Data structures (the *hash-tree*) and algorithm to quickly find all subsequences of a given sequence are described in [3] (and are similar to those used to find all subsets of a given itemset [2]).

3 The Sequence Phase

The general structure of the algorithms for the sequence phase is that they make multiple passes over the data. In each pass, we start with a seed set of large sequences. We use the seed set for generating new potentially large sequences, called *candidate sequences*. We find the support for these candidate sequences during the pass over the data. At the end of the pass, we determine which of the candidate sequences are actually large. These large candidates become the seed for the next pass. In the first pass, all 1-sequences with minimum support, obtained in the litemset phase, form the seed set.

We present two families of algorithms, which we call *count-all* and *count-some*. The count-all algorithms count all the large sequences, including non-maximal sequences. The non-maximal sequences must then be pruned out (in the maximal phase). We present one count-all algorithm, called *AprioriAll*, based on the Apriori algorithm for finding large itemsets presented in [2].[1]

We present two count-some algorithms: AprioriSome and DynamicSome. The intuition behind these algorithms is that since we are only interested in maximal sequences, we can avoid counting sequences which are contained in a longer sequence if we first count longer sequences. However, we have to be careful not to count a lot of longer sequences that do not have minimum support. Otherwise, the time saved by not counting sequences contained in a longer sequence may be less than the time wasted counting sequences without minimum support that would never have been counted because their subsequences were not large.

Both the count-some algorithms have a *forward phase*, in which we find all large sequences of certain lengths, followed by a *backward phase*, where we find all remaining large sequences. The essential difference is in the procedure they use for generating candidate sequences during the forward phase. As we will see momentarily, AprioriSome generates candidates for a pass using only the large sequences found in the previous pass and then makes a pass over the data to find their support. DynamicSome generates candidates on-the-fly using the large sequences found in the previous passes and the customer sequences read from the

[1] The AprioriHybrid algorithm presented in [2] did better than Apriori for finding large itemsets. However, it used the property that a k-itemset is present in a transaction if any two of its $(k-1)$-subsets are present in the transaction to avoid scanning the database in later passes. Since this property does not hold for sequences, we do not expect an algorithm based on AprioriHybrid to do any better than the algorithm based on Apriori.

Customer Id	Original Customer Sequence	Transformed Customer Sequence	After Mapping
1	$\langle$ (30) (90) $\rangle$	$\langle$ {(30)} {(90)} $\rangle$	$\langle$ {1} {5} $\rangle$
2	$\langle$ (10 20) (30) (40 60 70) $\rangle$	$\langle$ {(30)} {(40), (70), (40 70)} $\rangle$	$\langle$ {1} {2, 3, 4} $\rangle$
3	$\langle$ (30 50 70) $\rangle$	$\langle$ {(30), (70)} $\rangle$	$\langle$ {1, 3} $\rangle$
4	$\langle$ (30) (40 70) (90) $\rangle$	$\langle$ {(30)} {(40), (70), (40 70)} {(90)} $\rangle$	$\langle$ {1} {2, 3, 4} {5} $\rangle$
5	$\langle$ (90) $\rangle$	$\langle$ {(90)} $\rangle$	$\langle$ {5} $\rangle$

Figure 5: Transformed Database

L_1 = {large 1-sequences}; // Result of litemset phase
for ($k = 2$; $L_{k-1} \neq \emptyset$; k++) **do**
  begin
  C_k = New candidates generated from L_{k-1}
    (see Section 3.1.1).
  foreach customer-sequence c in the database **do**
    Increment the count of all candidates in C_k
      that are contained in c.
  L_k = Candidates in C_k with minimum support.
  end
Answer = Maximal Sequences in $\bigcup_k L_k$;

Figure 6: Algorithm AprioriAll

database.

Notation In all the algorithms, L_k denotes the set of all large k-sequences, and C_k the set of candidate k-sequences.

3.1 Algorithm AprioriAll

The algorithm is given in Fig. 6. In each pass, we use the large sequences from the previous pass to generate the candidate sequences and then measure their support by making a pass over the database. At the end of the pass, the support of the candidates is used to determine the large sequences. In the first pass, the output of the litemset phase is used to initialize the set of large 1-sequences. The candidates are stored in *hash-tree* [2] [3] to quickly find all candidates contained in a customer sequence.

3.1.1 Apriori Candidate Generation

The `apriori-generate` function takes as argument L_{k-1}, the set of all large $(k-1)$-sequences. The function works as follows. First, join L_{k-1} with L_{k-1}:

insert into C_k
select p.litemset$_1$, ..., p.litemset$_{k-1}$, q.litemset$_{k-1}$
from L_{k-1} p, L_{k-1} q
where p.litemset$_1$ = q.litemset$_1$, ...,
  p.litemset$_{k-2}$ = q.litemset$_{k-2}$;

Large 3-Sequences	Candidate 4-Sequences (after join)	Candidate 4-Sequences (after pruning)
$\langle$ 1 2 3 $\rangle$	$\langle$ 1 2 3 4 $\rangle$	$\langle$ 1 2 3 4 $\rangle$
$\langle$ 1 2 4 $\rangle$	$\langle$ 1 2 4 3 $\rangle$	
$\langle$ 1 3 4 $\rangle$	$\langle$ 1 3 4 5 $\rangle$	
$\langle$ 1 3 5 $\rangle$	$\langle$ 1 3 5 4 $\rangle$	
$\langle$ 2 3 4 $\rangle$		

Figure 7: Candidate Generation

Next, delete all sequences $c \in C_k$ such that some $(k-1)$-subsequence of c is not in L_{k-1}.

For example, consider the set of 3-sequences L_3 shown in the first column of Fig. 7. If this is given as input to the apriori-generate function, we will get the sequences shown in the second column after the join. After pruning out sequences whose subsequences are not in L_3, the sequences shown in the third column will be left. For example, $\langle$ 1 2 4 3 $\rangle$ is pruned out because the subsequence $\langle$ 2 4 3 $\rangle$ is not in L_3. Proof of correctness of the candidate generation procedure is given in [3].

3.1.2 Example

Consider a database with the customer-sequences shown in Fig. 8. We have not shown the original database in this example. The customer sequences are in transformed form where each transaction has been replaced by the set of litemsets contained in the transaction and the litemsets have been replaced by integers. The minimum support has been specified to be 40% (i.e. 2 customer sequences). The first pass over the database is made in the litemset phase, and we determine the large 1-sequences shown in Fig. 9. The large sequences together with their support at the end of the second, third, and fourth passes are also shown in the same figure. No candidate is generated for the

$\langle\{1\ 5\}\ \{2\}\ \{3\}\ \{4\}\rangle$
$\langle\{1\}\ \{3\}\ \{4\}\ \{3\ 5\}\rangle$
$\langle\{1\}\ \{2\}\ \{3\}\ \{4\}\rangle$
$\langle\{1\}\ \{3\}\ \{5\}\rangle$
$\langle\{4\}\ \{5\}\rangle$

Figure 8: Customer Sequences

fifth pass. The maximal large sequences would be the three sequences $\langle 1\ 2\ 3\ 4\rangle$, $\langle 1\ 3\ 5\rangle$ and $\langle 4\ 5\rangle$.

3.2 Algorithm AprioriSome

This algorithm is given in Fig. 10. In the forward pass, we only count sequences of certain lengths. For example, we might count sequences of length 1, 2, 4 and 6 in the forward phase and count sequences of length 3 and 5 in the backward phase. The function *next* takes as parameter the length of sequences counted in the last pass and returns the length of sequences to be counted in the next pass. Thus, this function determines exactly which sequences are counted, and balances the tradeoff between the time wasted in counting non-maximal sequences versus counting extensions of small candidate sequences. One extreme is $\text{next}(k) = k + 1$ (k is the length for which candidates were counted last), when all non-maximal sequences are counted, but no extensions of small candidate sequences are counted. In this case, AprioriSome degenerates into AprioriAll. The other extreme is a function like $\text{next}(k) = 100 * k$, when almost no non-maximal large sequence is counted, but lots of extensions of small candidates are counted.

Let hit_k denote the ratio of the number of large k-sequences to the number of candidate k-sequences (i.e., $|L_k|/|C_k|$). The *next* function we used in the experiments is given below. The intuition behind the heuristic is that as the percentage of candidates counted in the current pass which had minimum support increases, the time wasted by counting extensions of small candidates when we skip a length goes down.

```
function next(k: integer)
begin
    if (hit_k < 0.666)     return k + 1;
    elsif (hit_k < 0.75)   return k + 2;
    elsif (hit_k < 0.80)   return k + 3;
    elsif (hit_k < 0.85)   return k + 4;
    else                   return k + 5;
end
```

We use the `apriori-generate` function given in Section 3.1.1 to generate new candidate sequences. However, in the kth pass, we may not have the large sequence set L_{k-1} available as we did not count the $(k-1)$-candidate sequences. In that case, we use the candidate set C_{k-1} to generate C_k. Correctness is maintained because $C_{k-1} \supseteq L_{k-1}$.

In the backward phase, we count sequences for the lengths we skipped over during the forward phase, after first deleting all sequences contained in some large sequence. These smaller sequences cannot be in the answer because we are only interested in maximal sequences. We also delete the large sequences found in the forward phase that are non-maximal.

In the implementation, the forward and backward phases are interspersed to reduce the memory used by the candidates. However, we have omitted this detail in Fig. 10 to simplify the exposition.

```
// Forward Phase
L_1 = {large 1-sequences}; // Result of litemset phase
C_1 = L_1; // so that we have a nice loop condition
last = 1; // we last counted C_last
for ( k = 2; C_{k-1} ≠ ∅ and L_last ≠ ∅; k++ ) do
  begin
  if (L_{k-1} known) then
     C_k = New candidates generated from L_{k-1};
  else
     C_k = New candidates generated from C_{k-1};
  if ( k == next(last) ) then begin
     foreach customer-sequence c in the database do
        Increment the count of all candidates
           in C_k that are contained in c.
     L_k = Candidates in C_k with minimum support.
     last = k;
     end
  end

// Backward Phase
for ( k--; k >= 1; k-- ) do
  if (L_k not found in forward phase) then begin
     Delete all sequences in C_k contained in
        some L_i, i > k;
     foreach customer-sequence c in D_T do
        Increment the count of all candidates in C_k
           that are contained in c.
     L_k = Candidates in C_k with minimum support.
     end
  else // L_k already known
     Delete all sequences in L_k contained in
        some L_i, i > k.

Answer = ⋃_k L_k;
```

Figure 10: Algorithm AprioriSome

L_1

1-Sequences	Support
$\langle 1 \rangle$	4
$\langle 2 \rangle$	2
$\langle 3 \rangle$	4
$\langle 4 \rangle$	4
$\langle 5 \rangle$	4

L_2

2-Sequences	Support
$\langle 1\ 2 \rangle$	2
$\langle 1\ 3 \rangle$	4
$\langle 1\ 4 \rangle$	3
$\langle 1\ 5 \rangle$	3
$\langle 2\ 3 \rangle$	2
$\langle 2\ 4 \rangle$	2
$\langle 3\ 4 \rangle$	3
$\langle 3\ 5 \rangle$	2
$\langle 4\ 5 \rangle$	2

L_3

3-Sequences	Support
$\langle 1\ 2\ 3 \rangle$	2
$\langle 1\ 2\ 4 \rangle$	2
$\langle 1\ 3\ 4 \rangle$	3
$\langle 1\ 3\ 5 \rangle$	2
$\langle 2\ 3\ 4 \rangle$	2

L_4

4-Sequences	Support
$\langle 1\ 2\ 3\ 4 \rangle$	2

Figure 9: Large Sequences

$\langle 1\ 2\ 3 \rangle$
$\langle 1\ 2\ 4 \rangle$
$\langle 1\ 3\ 4 \rangle$
$\langle 1\ 3\ 5 \rangle$
$\langle 2\ 3\ 4 \rangle$
$\langle 3\ 4\ 5 \rangle$

Figure 11: Candidate 3-sequences

3.2.1 Example

Using again the database used in the example for the AprioriAll algorithm, we find the large 1-sequences (L_1) shown in Fig. 9 in the litemset phase (during the first pass over the database). Take for illustration simplicity, $f(k) = 2k$. In the second pass, we count C_2 to get L_2 (Fig. 9). After the third pass, apriori-generate is called with L_2 as argument to get C_3. The candidates in C_3 are shown in Fig. 11. We do not count C_3, and hence do not generate L_3. Next, apriori-generate is called with C_3 to get C_4, which after pruning, turns out to be the same C_4 shown in the third column of Fig. 7. After counting C_4 to get L_4 (Fig. 9), we try generating C_5, which turns out to be empty.

We then start the backward phase. Nothing gets deleted from L_4 since there are no longer sequences. We had skipped counting the support for sequences in C_3 in the forward phase. After deleting those sequences in C_3 that are subsequences of sequences in L_4, i.e., subsequences of $\langle 1\ 2\ 3\ 4 \rangle$, we are left with the sequences $\langle 1\ 3\ 5 \rangle$ and $\langle 3\ 4\ 5 \rangle$. These would be counted to get $\langle 1\ 3\ 5 \rangle$ as a maximal large 3-sequence. Next, all the sequences in L_2 except $\langle 4\ 5 \rangle$ are deleted since they are contained in some longer sequence. For the same reason, all sequences in L_1 are also deleted.

3.3 Algorithm DynamicSome

The DynamicSome algorithm is shown in Fig. 12. Like AprioriSome, we skip counting candidate sequences of certain lengths in the forward phase. The candidate sequences that are counted is determined by the variable *step*. In the initialization phase, all the candidate sequences of length upto and including *step* are counted. Then in the forward phase, all sequences whose lengths are multiples of *step* are counted. Thus, with *step* set to 3, we will count sequences of lengths 1, 2, and 3 in the initialization phase, and 6,9,12,... in the forward phase. We really wanted to count only sequences of lengths 3,6,9,12,... We can generate sequences of length 6 by joining sequences of length 3. We can generate sequences of length 9 by joining sequences of length 6 with sequences of length 3, etc. However, to generate the sequences of length 3, we need sequences of lengths 1 and 2, and hence the initialization phase.

As in AprioriSome, during the backward phase, we count sequences for the lengths we skipped over during the forward phase. However, unlike in AprioriSome, these candidate sequences were not generated in the forward phase. The intermediate phase generates them. Then the backward phase is identical to the one for AprioriSome. For example, assume that we count L_3 and L_6, and L_9 turns out to be empty in the forward phase. We generate C_7 and C_8 (intermediate phase), and then count C_8 followed by C_7 after deleting non-maximal sequences (backward phase). This process is then repeated for C_4 and C_5. In actual implementation, the intermediate phase is interspersed with the backward phase, but we have omitted this detail in Fig. 12 to simplify exposition.

We use `apriori-generate` in the initialization and intermediate phases, but use `otf-generate` in the forward phase. The `otf-generate` procedure is given in

```
// step is an integer ≥ 1
// Initialization Phase
L_1 = {large 1-sequences}; // Result of litemset phase
for ( k = 2; k <= step and L_{k-1} ≠ ∅; k++ ) do
  begin
  C_k = New candidates generated from L_{k-1};
  foreach customer-sequence c in D_T do
    Increment the count of all candidates in C_k
      that are contained in c.
  L_k = Candidates in C_k
      with minimum support.
  end

// Forward Phase
for ( k = step; L_k ≠ ∅; k += step ) do
  begin
  // find L_{k+step} from L_k and L_step
  C_{k+step} = ∅;
  foreach customer sequences c in D_T do
    begin
    X = otf-generate(L_k, L_step, c); See Section 3.3.1
    For each sequence x ∈ X', increment its count in
      C_{k+step} (adding it to C_{k+step} if necessary).
    end
  L_{k+step} = Candidates in C_{k+step} with min support.
  end

// Intermediate Phase
for ( k--; k > 1; k-- ) do
  if (L_k not yet determined) then
    if (L_{k-1} known) then
      C_k = New candidates generated from L_{k-1};
    else
      C_k = New candidates generated from C_{k-1};

// Backward Phase : Same as that of AprioriSome
```

Figure 12: Algorithm DynamicSome

Section 3.3.1. The reason is that `apriori-generate` generates less candidates than `otf-generate` when we generate C_{k+1} from L_k [2]. However, this may not hold when we try to find $L_{k+\text{step}}$ from L_k and L_{step} [2], as is the case in the forward phase. In addition, if the size of $|L_k| + |L_{\text{step}}|$ is less than the size of $C_{k+\text{step}}$ generated by apriori-generate, it may be faster to find all members of L_k and L_{step} contained in c than to find all members of $C_{k+\text{step}}$ contained in c.

3.3.1 On-the-fly Candidate Generation

The `otf-generate` function takes as arguments L_k, the set of large k-sequences, L_j, the set of large j-sequences, and the customer sequence c. It returns the set of candidate $(k+j)$-sequences contained in c.

The intuition behind this generation procedure is that if $s_k \in L_k$ and $s_j \in L_j$ are both contained in c, and they don't overlap in c, then $\langle s_k.s_j \rangle$ is a candidate $(k+j)$-sequence. Let c be the sequence $\langle c_1 c_2 ... c_n \rangle$. The implementation of this function is as shown below:

```
// c is the sequence ⟨ c_1 c_2 ... c_n ⟩
X_k = subseq(L_k, c);
forall sequences x ∈ X_k do
  x.end = min{j | x is contained in ⟨ c_1 c_2 ... c_j ⟩};
X_j = subseq(L_j, c);
forall sequences x ∈ X_j do
  x.start = max{j | x is contained in ⟨ c_j c_{j+1} ... c_n ⟩};
Answer = join of X_k with X_j with the join
  condition X_k.end < X_j.start;
```

For example, consider L_2 to be the set of sequences in Fig. 9, and let `otf-generate` be called with parameters L_2, L_2 and the customer-sequence $\langle \{1\}\ \{2\}\ \{3\ 7\}\ \{4\} \rangle$. Thus c_1 corresponds to $\{1\}$, c_2 to $\{2\}$, etc. The end and start values for each sequence in L_2 which is contained in c are shown in Fig. 13. Thus, the result of the join with the join condition $X_2.end < X_2.start$ (where X_2 denotes the set of sequences of length 2) is the single sequence $\langle 1\ 2\ 3\ 4 \rangle$.

3.3.2 Example

Continuing with our example of Section 3.1.2, consider a *step* of 2. In the initialization phase, we determine L_2 shown in Fig. 9. Then, in the forward phase, we get 2 candidate sequences in C_4: $\langle 1\ 2\ 3\ 4 \rangle$ with support of 2 and $\langle 1\ 3\ 4\ 5 \rangle$ with support of 1. Out of these, only $\langle 1\ 2\ 3\ 4 \rangle$ is large. In the next pass, we find C_6 to be

[2] The apriori-generate procedure in Section 3.1.1 needs to be generalized to generate C_{k+j} from L_k. Essentially, the join condition has to be changed to require equality of the first $k-j$ terms, and the concatenation of the remaining terms.

Sequence	End	Start
$\langle 1\ 2 \rangle$	2	1
$\langle 1\ 3 \rangle$	3	1
$\langle 1\ 4 \rangle$	4	1
$\langle 2\ 3 \rangle$	3	2
$\langle 2\ 4 \rangle$	4	2
$\langle 3\ 4 \rangle$	4	3

Figure 13: Start and End Values

empty. Now, in the intermediate phase, we generate C_3 from L_2, and C_5 from L_4. Since C_5 turns out to be empty, we count just C_3 during the backward phase to get L_3.

4 Performance

To assess the relative performance of the algorithms and study their scale-up properties, we performed several experiments on an IBM RS/6000 530H workstation with a CPU clock rate of 33 MHz, 64 MB of main memory, and running AIX 3.2. The data resided in the AIX file system and was stored on a 2GB SCSI 3.5" drive, with measured sequential throughput of about 2 MB/second.

4.1 Generation of Synthetic Data

To evaluate the performance of the algorithms over a large range of data characteristics, we generated synthetic customer transactions. environment. In our model of the "real" world, people buy sequences of sets of items. Each such sequence of itemsets is potentially a maximal large sequence. An example of such a sequence might be sheets and pillow cases, followed by a comforter, followed by shams and ruffles. However, some people may buy only some of the items from such a sequence. For instance, some people might buy only sheets and pillow cases followed by a comforter, and some only comforters. A customer-sequence may contain more than one such sequence. For example, a customer might place an order for a dress and jacket when ordering sheets and pillow cases, where the dress and jacket together form part of another sequence. Customer-sequence sizes are typically clustered around a mean and a few customers may have many transactions. Similarly, transaction sizes are usually clustered around a mean and a few transactions have many items.

The synthetic data generation program takes the parameters shown in Table 1. We generated datasets

$\|D\|$	Number of customers (= size of Database)
$\|C\|$	Average number of transactions per Customer
$\|T\|$	Average number of items per Transaction
$\|S\|$	Average length of maximal potentially large Sequences
$\|I\|$	Average size of Itemsets in maximal potentially large sequences
N_S	Number of maximal potentially large Sequences
N_I	Number of maximal potentially large Itemsets
N	Number of items

Table 1: Parameters

Name	$\|C\|$	$\|T\|$	$\|S\|$	$\|I\|$	Size (MB)
C10-T5-S4-I1.25	10	5	4	1.25	5.8
C10-T5-S4-I2.5	10	5	4	2.5	6.0
C20-T2.5-S4-I1.25	20	2.5	4	1.25	6.9
C20-T2.5-S8-I1.25	20	2.5	8	1.25	7.8

Table 2: Parameter settings (Synthetic datasets)

by setting $N_S = 5000$, $N_I = 25000$ and $N = 10000$. The number of customers, $|D|$ was set to 250,000. Table 2 summarizes the dataset parameter settings. We refer the reader to [3] for the details of the data generation program.

4.2 Relative Performance

Fig. 14 shows the relative execution times for the three algorithms for the six datasets given in Table 2 as the minimum support is decreased from 1% support to 0.2% support. We have not plotted the execution times for DynamicSome for low values of minimum support since it generated too many candidates and ran out of memory. Even if DynamicSome had more memory, the cost of finding the support for that many candidates would have ensured execution times much larger than those for Apriori or AprioriSome. As expected, the execution times of all the algorithms increase as the support is decreased because of a large increase in the number of large sequences in the result.

DynamicSome performs worse than the other two algorithms mainly because it generates and counts a much larger number of candidates in the forward phase. The difference in the number of candidates generated is due to the `otf-generate` candidate generation procedure it uses. The `apriori-generate` does not count any candidate sequence that contains any subsequence which is not large. The `otf-generate` does not have this pruning capability.

The major advantage of AprioriSome over AprioriAll is that it avoids counting many non-maximal se-

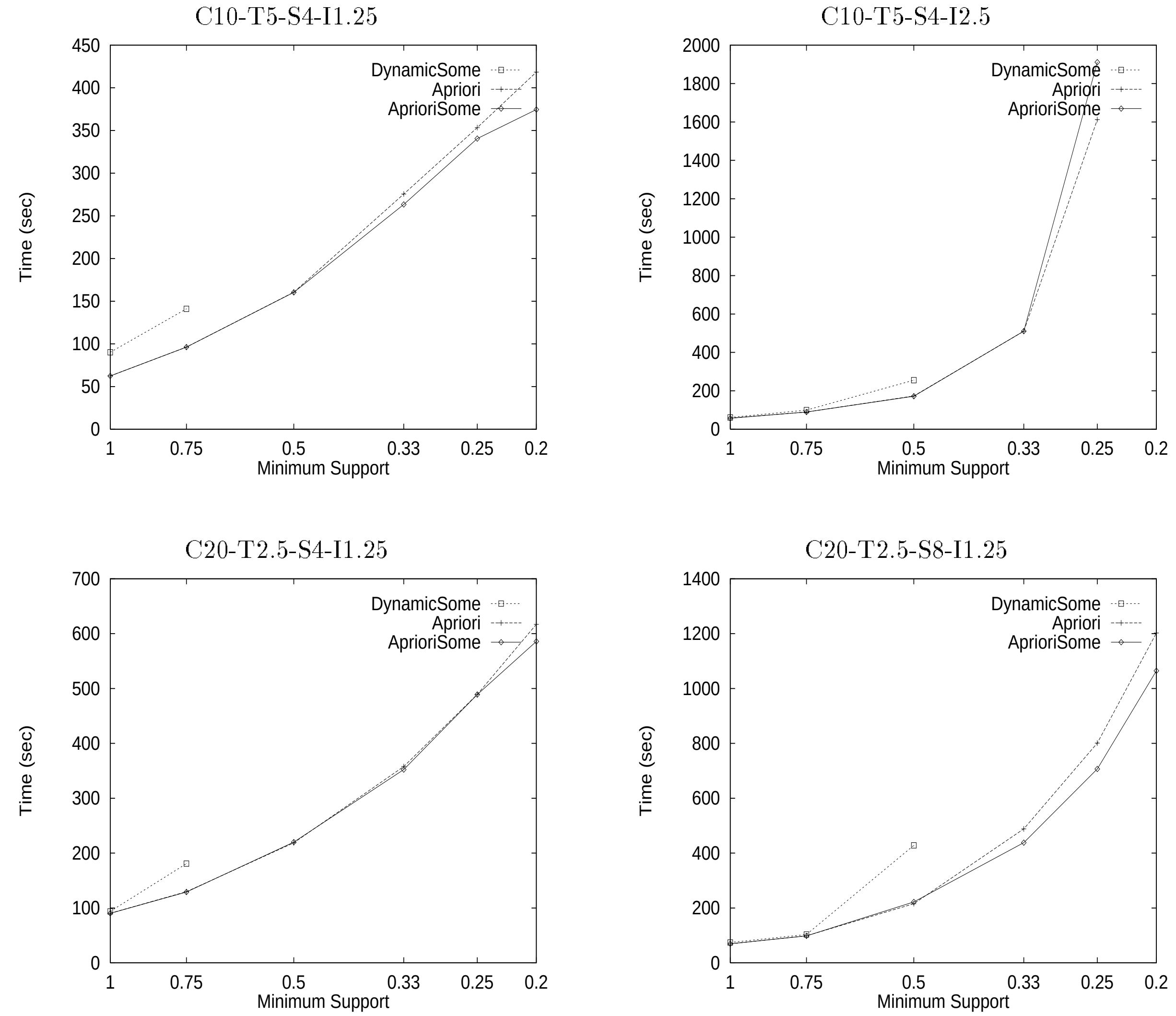

Figure 14: Execution times

quences. However, this advantage is reduced because of two reasons. First, candidates C_k in AprioriAll are generated using L_{k-1}, whereas AprioriSome sometimes uses C_{k-1} for this purpose. Since $C_{k-1} \supseteq L_{k-1}$, the number of candidates generated using AprioriSome can be larger. Second, although AprioriSome skips over counting candidates of some lengths, they are generated nonetheless and stay memory resident. If memory gets filled up, AprioriSome is forced to count the last set of candidates generated even if the heuristic suggests skipping some more candidate sets. This effect decreases the skipping distance between the two candidate sets that are indeed counted, and AprioriSome starts behaving more like AprioriAll. For lower supports, there are longer large sequences, and hence more non-maximal sequences, and AprioriSome does better.

4.3 Scale-up

We will present in this section the results of scale-up experiments for the AprioriSome algorithm. We also performed the same experiments for AprioriAll, and found the results to be very similar. We do not report the AprioriAll results to conserve space. We will present the scale-up results for some selected datasets. Similar results were obtained for other datasets.

Fig. 15 shows how AprioriSome scales up as the number of customers is increased ten times from 250,000 to 2.5 million. (The scale-up graph for increasing the number of customers from 25,000 to 250,000 looks very similar.) We show the results for the

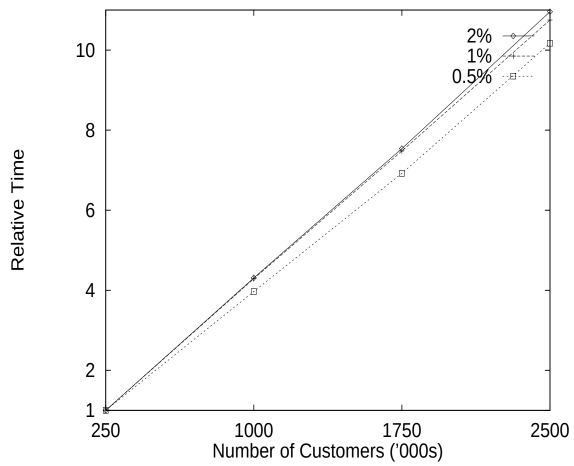

Figure 15: Scale-up : Number of customers

dataset C10-T2.5-S4-I1.25 with three levels of minimum support. The size of the dataset for 2.5 million customers was 368 MB. The execution times are normalized with respect to the times for the 250,000 customers dataset. As shown, the execution times scale quite linearly.

Next, we investigated the scale-up as we increased the total number of items in a customer sequence. This increase was achieved in two different ways: i) by increasing the average number of transactions per customer, keeping the average number of items per transaction the same; and ii) by increasing the average number of items per transaction, keeping the average number transactions per customer the same. The aim of this experiment was to see how our data structures scaled with the customer-sequence size, independent of other factors like the database size and the number of large sequences. We kept the size of the database roughly constant by keeping the product of the average customer-sequence size and the number of customers constant. We fixed the minimum support in terms of the number of transactions in this experiment. Fixing the minimum support as a percentage would have led to large increases in the number of large sequences and we wanted to keep the size of the answer set roughly the same. All the experiments had the large sequence length set to 4 and the large itemset size set to 1.25. The average transaction size was set to 2.5 in the first graph, while the number of transactions per customer was set to 10 in the second. The numbers in the key (e.g. 100) refer to the minimum support.

The results are shown in Fig. 16. As shown, the execution times usually increased with the customer-sequence size, but only gradually. The main reason for the increase was that in spite of setting the minimum support in terms of the number of customers, the number of large sequences increased with increasing customer-sequence size. A secondary reason was that finding the candidates present in a customer sequence took a little more time. For support level of 200, the execution time actually went down a little when the transaction size was increased. The reason for this decrease is that there is an overhead associated with reading a transaction. At high level of support, this overhead comprises a significant part of the total execution time. Since this decreases when the number of transactions decrease, the total execution time also decreases a little.

5 Conclusions and Future Work

We introduced a new problem of mining sequential patterns from a database of customer sales transactions and presented three algorithms for solving this problem. Two of the algorithms, AprioriSome and AprioriAll, have comparable performance, although AprioriSome performs a little better for the lower values of the minimum number of customers that must support a sequential pattern. Scale-up experiments show that both AprioriSome and AprioriAll scale linearly with the number of customer transactions. They also have excellent scale-up properties with respect to the number of transactions in a customer sequence and the number of items in a transaction.

In some applications, the user may want to know the ratio of the number of people who bought the first $k+1$ items in a sequence to the number of people who bought the first k items, for $0 < k <$ length of sequence. In this case, we will have to make an additional pass over the data to get counts for all prefixes of large sequences if we were using the AprioriSome algorithms. With the AprioriAll algorithm, we already have these counts. In such applications, therefore, AprioriAll will become the preferred algorithm.

These algorithms have been implemented on several data repositories, including the AIX file system and DB2/6000, as part of the Quest project, and have been run against data from several data. In the future, we plan to extend this work along the following lines:

- Extension of the algorithms to discover sequential patterns across item categories. An example of such a category is that a dish washer *is a* kitchen appliance *is a* heavy electric appliance, etc.

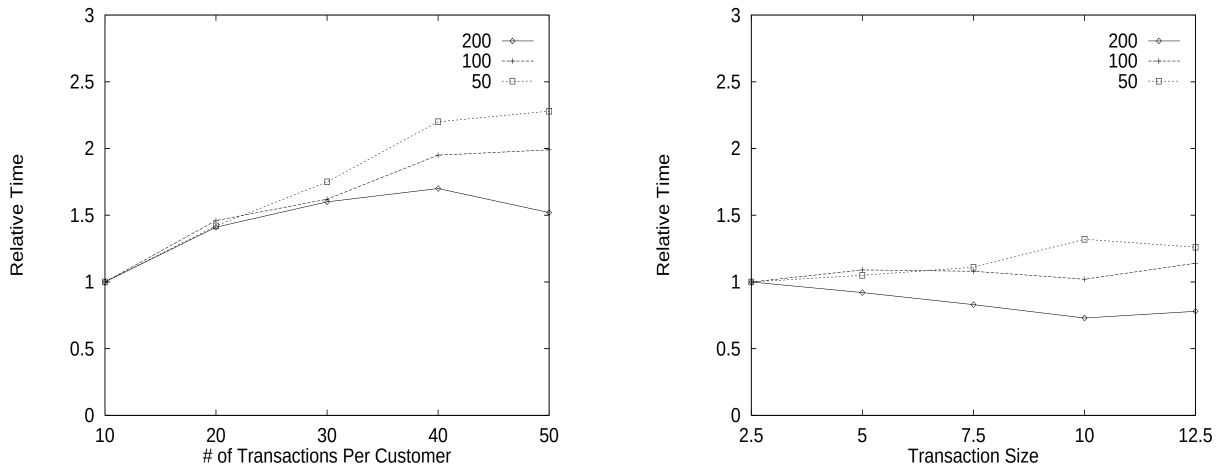

Figure 16: Scale-up : Number of Items per Customer

- Transposition of constraints into the discovery algorithms. There could be item constraints (e.g. sequential patterns involving home appliances) or time constraints (e.g. the elements of the patterns should come from transactions that are at least d_1 and at most d_2 days apart.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the VLDB Conference*, Santiago, Chile, September 1994. Expanded version available as IBM Research Report RJ9839, June 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. Research Report RJ 9910, IBM Almaden Research Center, San Jose, California, October 1994.

[4] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 1990.

[5] A. Califano and I. Rigoutsos. Flash: A fast look-up algorithm for string homology. In *Proc. of the 1st International Converence on Intelligent Systems for Molecular Biology*, Bethesda, MD, July 1993.

[6] T. G. Dietterich and R. S. Michalski. Discovering patterns in sequences of events. *Artificial Intelligence*, 25:187–232, 1985.

[7] L. Hui. Color set size problem with applications to string matching. In A. Apostolico, M. Crochemere, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching, LNCS 644*, pages 230–243. Springer-Verlag, 1992.

[8] M. Roytberg. A search for common patterns in many sequences. *Computer Applications in the Biosciences*, 8(1):57–64, 1992.

[9] M. Vingron and P. Argos. A fast and sensitive multiple sequence alignment algorithm. *Computer Applications in the Biosciences*, 5:115–122, 1989.

[10] J. T.-L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Minneapolis, May 1994.

[11] M. Waterman, editor. *Mathematical Methods for DNA Sequence Analysis*. CRC Press, 1989.

[12] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.