IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 10, Issue 1 (Mar. - Apr. 2013), PP 83-88
www.iosrjournals.org

Mining Approach for Updating Sequential Patterns

Bhagyshri Lachhwani, Mehul P Barot
LDRP Institute of Research and Technology, Gandhinagar

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. The discovery of frequent sequences in temporal databases is an important data mining problem. Most current work assumes that the database is static, and a database update requires rediscovering all the patterns by scanning the entire old and new database. We consider the problem of the incremental mining of sequential patterns when new transactions or new customers are added to an original database. In this paper, we propose novel techniques for maintaining sequences in the presence of a) database updates, and b) user interaction (e.g. modifying mining parameters). This is a very challenging task, since such updates can invalidate existing sequences or introduce new ones.

I. Introduction

Sequential pattern mining can mine only the maximal frequent subsequences, or all frequent subsequences. 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 $(i_1, i_2, i_3, ... i_m)$ where i_j is an item1. We denote a sequence s by $<s_1, s_2, s_3, ... s_n>$ where s_j is an itemset.

A sequence $<a_1, a_2, a_3, ... a_n>$ is contained in another sequence $<b_1, b_2, b_3, ... b_m>$ if there exists integers $k_1 < k_2 < ... < k_n$ such that $a_1 \subseteq b_{k1}, a_2 \subseteq b_{k2}, ... a_n \subseteq b_{kn}$. For example, the sequence <(3) (4 5) (8)> is contained in <(7) (3 8) (9) (4 5 6) (8)>, since (3) $\subseteq$ (3 8), (4 5) $\subseteq$ (4 5 6) and (8) $\subseteq$ (8). However, the sequence <(3) (5)> is not contained in <(3 5)> and vice versa. The former represents items 3 and 5 being bought one after the other, while the latter represent items 3 and 5 being bought together.

In this paper our goal is to minimize the I/O and computation requirements for handling incremental updates. Our algorithm accomplishes this goal by maintaining information about "maximally frequent" and "minimally infrequent" sequences. When incremental data arrives, the incremental part is scanned once to incorporate the new information.

The new data is combined with the "maximal" and "minimal" information in order to determine the portions of the original database that need to be re-scanned. This process is aided by the use of a vertical database layout — where attributes are associated with the list of transactions in which they occur. The result is an improvement in execution time by up to several orders of magnitude in practice, both for handling increments to the database, as well as for handling interactive queries.

A record supports a sequence s if s is contained in it. The support count is incremented only once per record. The support for a sequence is defined as the fraction of the whole data set that contains this sequence. If this support $\leq$ min_sup, then the sequence is frequent.

II. Incrementally Mining Sequential Pattern Algorithms

2.1 The SPADE Algorithm

In this section we describe SPADE [1], an algorithm for fast discovery of frequent sequences, which forms the basis for our incremental algorithm. Sequence Lattice: SPADE uses the observation that the subsequence relation $\leq$ defines a partial order on the set of sequences, i.e,if is a frequent sequence, then all subsequences are also frequent. The algorithm systematically searches the sequence lattice spanned by the subsequence relation, from the most general (single items) to the most specific frequent sequences (maximal sequences) in a depth-first manner. For instance, in Figure1, the bold lines correspond to the lattice for the example dataset.

Support Counting: Most of the current sequence mining algorithms assume a horizontal database layout such as the one shown in Figure 1. In the horizontal format, the database consists of a set of customers (cid's). Each customer has a set of transactions (tid's), along with the items contained in the transaction. In contrast, we use a vertical database layout, where we associate with each item X in the sequence lattice its idlist, denoted L(X), which is a list of all customer(cid)and transaction identifier(tid) pairs containing the item. For example, the idlist for the item C in the original database (Figure 1) would consist of the tuples {< 2,20>,< 2,30>}.

Given the per item idlists, we can iteratively determine the support of any k-sequence from the idlists of any two ofits(k -1) length subsequences. In particular, we combine (intersect) 1 the two (k - 1) length subsequences that share a common suffix(the generating sequences)to compute the support of a new k length sequence. A simple check on the support of the resulting idlist tells us whether the new sequence is frequent or not.

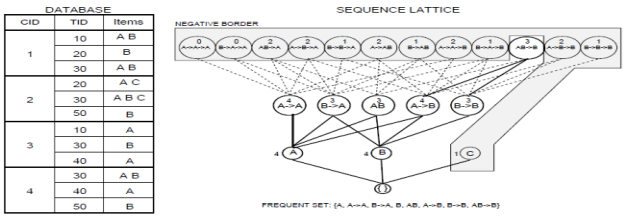

Figure 1 :Original Database

If we had enough main-memory, we could enumerate all the frequent sequences by traversing the lattice, and performing intersections to obtain sequence supports. In practice, however, we only have a limited amount of main-memory, and all the intermediate idlists will not fit in memory. SPADE breaks up this large search space into small, manageable chunks that can be processed independently in memory. This is accomplished via suffix-based equivalence classes (henceforth denoted as a class). We say that two k length sequences are in the same class if they share a common k -1 length suffix.

The key observation is that each class is a sub-lattice of the original sequence lattice and can be processed independently. Each suffix class is independent in the sense that it has complete information for generating 1 Described in Zaki[1]. all frequent sequences that share the same suffix. For example, if a class[X] has the elements Y -> X, and Z -> X as the only sequences, the only possible frequent sequences at the next step can be Y -> Z -> X, Z -> Y -> X, and (YZ) -> X. It should be obvious that no other item Q can lead to a frequent sequence with the suffix X, unless (QX)or Q-> X is also in[X].

BEGIN Enumerate-Frequent-Seq($[S]$):
  for all elements $A_i \in [S]$ **do**
    $[A_i] = \emptyset$;
    for all elements $A_j \in [S]$ **do**
      $R = A_j \cup A_i$; /*sequences R formed by generating
        subsequences A_j and A_i with A_i as a suffix*/
      $idlist(R) = idlist(A_j) \cap idlist(A_i)$;
      if $support(idlist(R)) \geq min\_sup$ **then**
        $[A_i] = [A_i] \cup \{R\}$;
  for all $[A_i] \neq \emptyset$ **do** *Enumerate-Frequent-Seq*($[A_i]$);
END Enumerate-Frequent-Seq($[S]$):

Figure 2-: Enumerating Frequent Sequences

SPADE recursively decomposes the sequences at each new level into even smaller independent classes. For instance, at level one it uses suffix classes of length one (X,Y), at level two it uses suffix classes of length two (X ->Y, XY) and so on. We refer to level one suffix classes as parent classes. These suffix classes are processed one-by-one. Figure 2 shows the pseudo-code (simplified for exposition, see[1]for exact details) for the main procedure of the SPADE algorithm. The input to the procedure is a class, along with the idlist for each of its elements. Frequent sequences are generated by intersecting [1] the idlists of all distinct pairs of sequences in each class and checking the support of the resulting idlist against min sup. The sequences found to be frequent at the current level form classes for the next level. This level-wise process is recursively repeated until all frequent sequences have been enumerated. In terms of memory management, it is easy to see that we need memory to store intermediate idlists for at most two consecutive levels. Once all the frequent sequences for the next level have been generated, the sequences at the current level can be deleted. For more details on SPADE, see[1].

2.2 ISM

The ISM algorithm, proposed by [2], is actually an extension of SPADE, which aims at considering the update by means of the negative border and a rewriting of the database. Figure 3 is an example of a database and its update (items in bold characters). We observe that three clients have been updated.

The first iterations of SPADE on DBspade, ended in the lattice given in Figure1 (without the gray section). The main idea of ISM is to keep the negative border (in gray, Figure 1) NB, which is made of j-

candidates, at the bottom of the hierarchy in the lattice. We can observe, in Figure 4 the lattice and negative border for DBspade. Note that hash lines stand for a hierarchy that does not end in a frequent sequence.
The first step of ISM aims at pruning the sequences that become infrequent from the set of frequent sequences after the update. The second step aims at taking into account the new frequent sequences one by one, in order to make the information browse the lattice using the SPADE generating process. The field of observation considered by ISM is thus limited to the new items.

Client	Itemset	Items
1	10	A B
	20	B
	30	A B
	100	**A C**
2	20	A C
	30	A B C
	50	B
3	10	A
	30	B
	40	A
	110	**C**
	120	**B**
4	30	A B
	40	A
	50	B
	140	**C**

Figure 3: DBspade, a database and its update

Let us consider item ‘‘C’’ in DBspade. This item only has a threshold of 1 sequence according to SPADE. After the update given in Figure 3, ISM will consider that support, which is now of four sequences. ‘‘C’’ is now going from NB to the set of frequent sequences. In the same way, the sequences <(A)(A)(B)> and <(A)(B)(B)> become frequent after the update and go from NB to the set of frequent sequences. This is the goal of the first step.

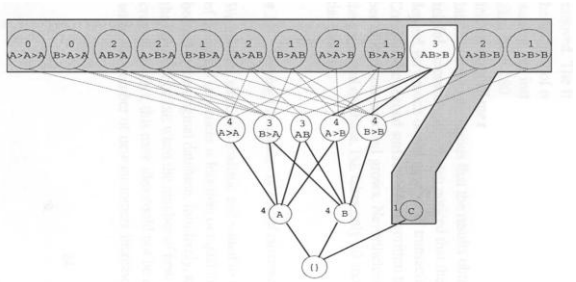

Figure 4: The negative border, considered by ISM after using SPADE on the database from Figure 3, before the update

The second step is intended to consider the generation of candidates, but is limited to the sequences added to the set of frequent sequences during the first step. For instance, sequences <(A)(A)(B)> and <(A)(B)(B)> can generate the candidate <(A)(A)(B)(B)> which will have a support of 0sequences and will be added to the negative border. After the update, the set of frequent sequences will thus be: A, B, C, <(A)(A)>, <(B)(A)>, <(AB)>, <(A)(B)>, <(B)(B)>, <(A)(C)>, <(B)(C)>, <(A)(A)(B)>, <(AB)(B)>, <(A)(B)(B)>, <(A)(A)(C)>, <(A)(B)(C)>.

At the end of the second and last step, the lattice is updated and ISM can give the new set of frequent sequences, as well as a new negative border, allowing the algorithm to take a new update into account. As we observe in Figure 4, the lattice storing the frequent itemsets and the negative border can be very large and memory intensive.

2.3 SPAM

Ayres [4] proposed SPAM algorithm based on the key idea of SPADE. The difference is that SPAM utilizes a bitmap representation of the database instead of {SI D, T I D} pairs used in the SPADE algorithm. Hence, SPAM can perform much better than SPADE and others by employing bitwise operations.

While scanning the database for the first time, a vertical bitmap is constructed for each item in the database, and each bitmap has a bit corresponding to each itemset (element) of the sequences in the database. If an item appears in an itemset, the bit corresponding to the itemset of the bitmap for the item is set to one; otherwise, the bit is set to zero. Figure 5 shows the bitmap vertical table.

\

SID	TID	{a}	{b}	{c}	{d}
1	1	1	0	0	0
1	2	0	0	1	0
1	3	0	1	1	0
1	4	0	0	0	1
1	5	1	1	1	0
1	6	1	0	0	0
1	7	0	0	0	1
2	1	0	1	0	0
2	2	0	0	1	1
2	3	1	0	0	0
2	4	0	0	1	0
2	5	0	1	0	1
3	1	0	0	0	1
3	2	0	1	1	0
3	3	1	0	1	0
3	4	0	0	1	1

Figure 5: Bitmap Vertical Table

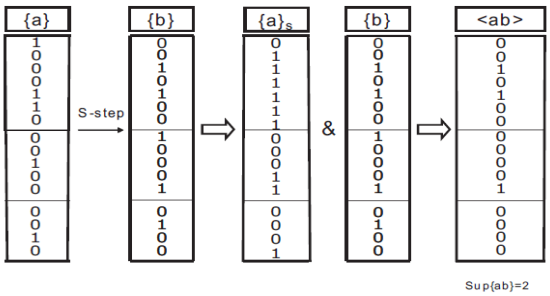

Figure 6: SPAM S-Step join

To generate and test the candidate sequences, SPAM uses two steps, S-step and I- step, based on the lattice concept. To extend a sequence, the S-step appends an item to it as the new last element, and the I-step appends the item to its last element if possible. The resultant bitmap of the S-step can be obtained by doing ANDing operation for the transformed bitmap and the bitmap of the appended item. Figure 6 illustrates how to join two 1-length patterns, a and b, based on the example database. I-step uses the bitmaps of the sequence and the appended item to do ANDing operation to get the resultant bitmap, as shown in Figure 7, which extend the pattern <ab> to the candidate <a(bc)>. The support counting becomes a simple check how many bitmap partitions not containing all zeros.

The main drawback of SPAM is the huge memory space necessary. For example, although an item, α, does not exist in a sequence, *s*, SPAM still uses one bit to represent the existence of α in *s*. This disadvantage restricts SPAM as a best algorithm on mining large datasets in limit resource environments.

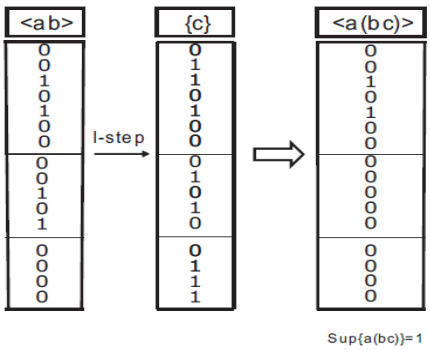

Figure 7: SPAM I-Step join

2.4 Ise

In [2] the main consequence of adding new customers is to verify the support of the frequent sequences in L^{DB}. During the first pass on db, we count the support of individual items and are provided with 1-candExt standing for the set of items occurring at least once in db. This set is called L_1^{db}.The frequent 1-sequences in db to generate new candidates. This candidate generation works by joining L_1^{db} with L_1^{db} and yields the set of

candidate 2-sequences.Such a set is called 2-candExt. This phase is quite different from the GSP approach since we do not consider the support constraint. A candidate 2-sequence is in 2-candExt if and only if it occurs at least once in db. Next, we scan U to find out frequent 2-sequences from 2-candExt. This set is called freqExt and it is achieved by discarding the 2-sequences that do not verify the minimum support from 2-candExt.
During the scan to find 2-freqExt, we also obtain the set of frequent sub-sequences preceding items of db. From this set, by appending the items of db to the frequent sub-sequences we obtain a new set of frequent sequences. This set is called freqSeed.In subsequent iterations we go on to discover the all frequent sequences not yet embedded in freqSeed and 2-freqExt.

Assume that we are at the jth pass. In these subsequent iterations, we start by generating new candidates from the two sets found in the previous pass. The main idea of candidate generation is to retrieve from among sequences of freqSeed and j-freqExt, two sequences (s $\in$ freqSeed; s' $\in$ j-freqExt) such that the last item of s is the first item of s'. When such a condition hold items for a pair (s, s'), a new candidate sequence is built by dropping the last of s and appending s' to the remaining sequence.

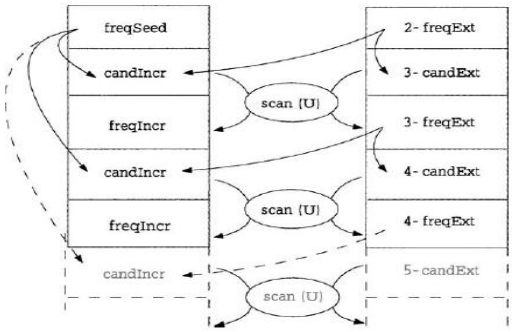

Figure 8: ISE iterations with j ≥ 2

for all candidates are then obtained by scanning U and those with minimum support become frequent sequences. The two sets become, respectively, freqInc and (j + 1)-freqExt. The last one and freqSeed are then used to generate new candidates.

III. Proposed New Algorithm

The original large and pre-large sequences with their counts from preceding runs are retained for later use in maintenance. As new transactions are added, the proposed approach first transforms them into new customer sequences and merges them with the corresponding old sequences existing in the original database. The newly merged customer sequences are then scanned to generate candidate 1-sequences with occurrence increments. These candidate sequences are compared to the large and pre-large 1-sequences which were previously retained.

These candidate sequences are divided into three parts according to whether they are large, pre-large or small in the original database. If a candidate 1-sequence is also among the previously retained large or prelarge 1-sequences, its new total count for the entire updated database can easily be calculated from its current count increment and previous count, since all previous large and pre-large sequences with their counts have been retained. Whether an original large or pre-large sequence is still large or pre-large after new transactions are added is then determined from its new support ratio, which is derived from its total count over the total number of customer sequences.

If a candidate 1-sequence does not exist among the previously retained large or pre-large 1-sequences, then the sequence is absolutely not large for the entire updated database when the number of newly merged customer sequences is within the safety bound. In this situation, no action is needed. When new transaction data are incrementally added and the total number of newly added customer sequences exceeds the safety bound, the original database must be re-scanned to find new large and pre-large sequences.

The proposed approach can thus find all large 1-sequences for the entire updated database. After that, candidate 2-sequences from the newly merged customer sequences are formed, and the same procedure is used to find all large 2-sequences. This procedure is repeated until all large sequences have been found.

Two global variables, c and b, are used to accumulate, respectively, the number of newly added customer sequences and the number of newly added customer sequences belonging to old customers since the last re-scan of the original database.

The Output is a set of final large sequential patterns for the updated database.The procedure works as follows first the value $[(S_u - S_l)d] / (1 - S_u)$ is calculated where d is the number of customer sequences in D , $\mathbf{S_u}$ is the upper support threshold for large sequences,$\mathbf{S_l}$ **is** the lower support threshold for pre-large sequences, $S_l < S_u$.Then, Merge the newly added customer sequences with the old sequences in the original database,after that

one temporary variable is taken say q, which is the number of the newly added customer sequences belonging to old customers. Then one variable say h is taken and b = b + q and calculate the value of the term as: $h = bS_u / (1 - S_u)$,where b is the accumulative amount of q since the last re-scan.Then one variable say k is taken and initially Set k = 1, where k is used to record the number of itemsets in the sequences currently being processed. Then Find all candidate k-sequences C_k and their count increments from the newly merged customer sequences T. Divide the candidate k-sequences into three parts according to whether they are large, pre-large or small in the original database. In this way find all the large or pre-large sequences of newly added transactions and customers from updated database. For deciding whether the sequences are large or pre-large do the following, for each k-sequence I in the original large k-sequences L_k^D (i)Set the new count $S^U(I) = S^T(I) + S^D(I)$. (ii)If $S^U(I)/(d + c + t - b) \geq S_u$, then assign I as a large sequence, set $S^D(I) = S^U(I)$ and keep I with $S^D(I)$; otherwise, if $S^U(I)/(d + c + t - b) \geq S_l$, then assign I as a pre-large sequence, set $S^D(I) = S^U(I)$ and keep I with $S^D(I)$; otherwise, ignore I. where I is sequence, $S^D(I)$ is the number of occurrences of I in D. $S^T(I)$ is the number of occurrence increments of I in T.$S^U(I)$ is the number of occurrences of I in U. d is the number of customer sequences in D. t is the number of customer sequences in T. Same way Pre-large sequences are decided. Put I in the rescan-set R for each k-sequence I in the candidate k-sequences C_k that is neither in the original large sequences L_k^D nor in the pre-large sequences P_k^D, for use when rescanning in Step 10 is necessary. Now if $c + t \leq f - h$ or R is null, then do nothing; otherwise, rescan the original database to determine whether the sequences in the rescan-set R are large or pre-large. Form candidate (k + 1)-sequences C_{k+1} from finally large and pre-large k-sequences ($L_k^D \cup P_k^D$) that appear in the newly merged transactions. where L_k^D is the set of large k-sequences from D. L_k^T is the set of large k-sequences from T. L_k^U is the set of large k-sequences from U. P_k^D is the set of pre-large k-sequences from D. P_k^T is the set of pre-large k-sequences from T. P_k^U is the set of pre-large k-sequences from U. C_k is the set of all candidate k-sequences from T. Increment value of k by 1. Repeat the above STEPs until no new large or pre-large sequences are found. Modify the maximal large sequence patterns according to the modified large sequences. Now If $c + t > f - h$, then set $d = d + c + t$, $c = 0$ and $b = 0$; otherwise, set $c = c + t$. After all these steps the finally maximal large sequences for the updated database can be determined.

IV. Conclusion

Memory Management SPADE simply requires per item idlists. The vertical database is partitioned into a number of blocks such that each individual block fits in memory. Each block contains the vertical representation of all transactions involving a set of customers. Within each block there exists an item dereferencing array, pointing to the first entry for each item. Given a customer, and an item, we first identify the block containing the customer's transactions using a first level cid-index (hash function). The second item-index then locates the item within the given block. After this we perform a linear search for the exact customer identifier. Using this two level indexing scheme we can quickly jump to only that portion of the database which will be affected by the update, without having to touch the entire database. Note that using a vertical data format we were able to efficiently retrieve all affected item's cids, without having to touch the entire database. This is not possible in the horizontal format, since a given item can appear in any transaction, which is found by scanning the entire data.ISM allows the algorithm to take a new update into account but for frequent itemsets and the negative border can be very large and memory intensive. SPAM utilizes a bitmap representation of the database and perform much better than SPADE but for it the huge memory space necessary. ISE provides all the frequent sequences in the updated database. The assumptions with this new technique are taken that 1) It avoids recomputing large sequences that have already been discovered. 2) It focuses on newly added customer sequences, which are transformed from newly added transactions, thus greatly reducing the number of candidate sequences. 3) It uses a simple check to further filter candidate sequences in newly added customer sequences. 4) It effectively handles the case, in which sequences are small in an original database.

References

[1] M. J. Zaki. Efficient enumeration of frequent sequences. In 7th CIKM, 1998

[2] Florent Masseglia; Pascal Poncelet; Maguelonne Teisseire;, "Incremental mining of sequential patterns in large databases," *Data & Knowledge Engineering*, pp. 97-121, 2003

[3] S. Parthasarathy; M. Zaki; M. Ogihara; S. Dwarkadas; , "Incremental and interactive sequence mining," In Proceedings of the 8th International Conference on Information and Knowledge Management (CIKM' 99), pp. 251–258, 1999

[4] J. Ayres; J. Gehrke; T. Yiu; J. Flannick; , "Sequential pattern mining using a bitmap representation," *In Proceedings of ACM SIGKDD International Confer ence on Knowledge Discovery and Data Mining*, pp. 429–435, 2002