

## Lattice Based Algorithm for Incremental Mining of Association Rules

<sup>1</sup>Ravindra Patel, <sup>2</sup>D. K. Swami and <sup>3</sup>K. R. Pardasani

<sup>1,2</sup>*Department of Computer Applications, Samrat Ashok Technological Institute  
(Engineering College), Vidisha (M.P.) India*

<sup>3</sup>*Department of Mathematics & Computer Applications, Maulana Azad National  
Institute, Bhopal (M.P.) India*

### Abstract

Mining frequent pattern in a large transactional database is a time consuming due to its complexity. While maintaining present frequent is a non-trivial task when some new transactions are appended to the database. Since appended transactions may invalidate old patterns. One characteristic of the Apriori based algorithms is that candidate itemsets are generated in rounds, with the size of the itemsets incremented by one per round. The number of database scans required by Apriori based algorithms thus depends on the size of the largest large itemsets. We devise a new algorithm IA\_Gen, which uses frequent patterns collected during an earlier mining as knowledge base and generates candidate itemsets of multiple sizes during each database scan.

**Keywords:** incremental mining, association rules, frequent itemset, lattice, Apriori.

### 1. Introduction

Data mining and knowledge discovery (KDD) lies at the interface of statistics, database technology, machine learning, high-performance computing and other areas. It is concerned with the computational and data intensive process of deriving interesting and useful information or patterns from massive databases. Association rule mining is an important data mining technique to generate correlations and association rules [1]. The problem of mining association rules could be decomposed into two sub problems, the mining of large itemsets (i.e. frequent itemsets) and the generation of association rules.

Several problems arise in the large itemset discovery task, mostly as a consequence of the large size of the databases involved in this process. Moreover, many organizations today have more than large databases; they have databases that change and grow continuously. For example, retail chains record millions of transactions, telecommunications companies connect thousands of calls, and popular web sites log millions of hits. In all these applications the evolving database is updated with a new block of data at regular time intervals. For large time intervals, we have the common scenario in many data warehouses. For small intervals, we have streaming data. Issues like interactivity and quick response times are paramount. Providing interactivity and quick response times when the database is evolving is a challenging task, since changes in the data can invalidate the model of large itemsets and make data understanding and knowledge discovery difficult. Simply using traditional approaches to update the model can result in an explosion in the computational and I/O resources required. Recognizing the dynamic nature of most operational databases, much effort has been devoted to the problem of mining large itemsets in evolving databases and several researchers [2, 3, 6, 9, 10, 11] have proposed interesting solutions and efficient algorithms. Generally these algorithms are incremental in the sense that they re-use previously mined information and try to combine it with the fresh data to reduce I/O requirements.

In this paper we present a new approach for mining large itemsets in evolving databases. Our approach differs from existing approaches in several ways. To the best of our knowledge, it is the first incremental techniques to provide significant computational and I/O savings. These features are very effective to facilitate data understanding and knowledge discovery in evolving databases.

## 2. Related Works

There has been a lot of research in developing efficient algorithms for mining frequent itemsets. Most of them enumerate all frequent itemsets. There also exist methods which only generate frequent closed itemsets [12] and maximal frequent itemsets [5]. While these methods generate a reduced number of itemsets, they still need to mine the entire database in order to generate the model of frequent itemsets, and therefore these methods are not efficient in mining evolving databases. Much effort has been devoted to the problem of incrementally mining frequent itemsets [2, 3, 4, 6, 9, 10]. Some of these algorithms cope with the problem of determining when to update, while the others simply treat arbitrary insertions and deletions of transactions. Lee [6] proposed the DELI algorithm, which uses statistical sampling methods to determine when to apply the updating process. Another interesting approach is DEMON [4], which helps adapt incremental algorithms to work effectively with evolving data by monitoring changes in the stream. Our approaches are different from all the above approaches in several ways. First, while these approaches need to perform database scans ( is the size of the largest frequent itemset), our approaches require only one scan on the incremental database and only a partial scan on the original database.

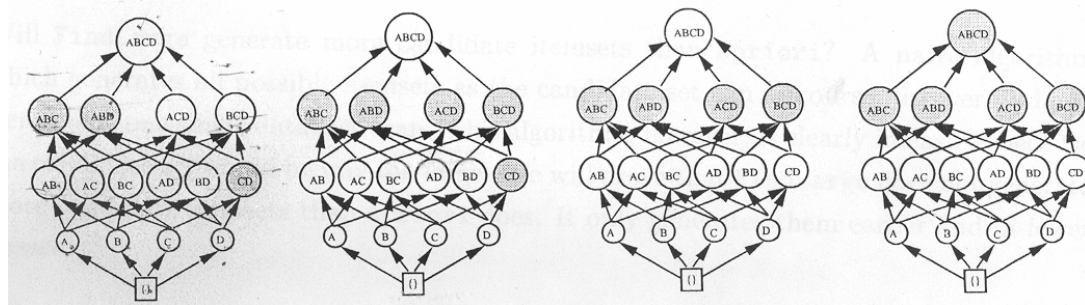
Second, we support selective updates, that is, instead of determining when to update the entire set of frequent itemsets, we determine which particularly itemsets need to be updated.

### 3. Problem Formulations

Most of the algorithms devised to find large itemsets are based on the **Apriori** algorithm [7]. The **Apriori** algorithm finds out the large itemsets iteratively. In the  $i^{th}$  iteration, **Apriori** generates a number of candidate itemset of size  $i$  (The size of an itemset is the number of items the itemset contains). **Apriori** then scans the database to find the support count of each candidate itemset. Itemsets whose support counts are smaller than the minimum support are discarded. **Apriori** terminates when no more candidate set can be generated.

IA\_Gen takes a (partial) set of multiple-sized large itemsets as a knowledge base to generate a set of multiple-sized candidate itemsets. This approach allows us to take advantage of knowledge base generated in earlier mining or obtain by sampling. .

(Example 1) As a simple example [13], suppose the itemset  $\{a, b, c, d\}$  and all its subsets are large in a database. **Apriori** will require four passes over the data to generate the itemset  $\{a, b, c, d\}$ . However, if one already knew that the itemsets  $\{a, b, c\}$ ,  $\{a, b, d\}$ , and  $\{c, d\}$  were large, then we can use this piece of information to help us generate the itemset  $\{a, b, c, d\}$  early. Note that these subsets are large as well. We can then divide these large itemsets into groups according to their sizes, and apply Apriori\_Gen to each group. This strategy will thus generate candidate itemsets of various sizes.



(i) ABC, ABD and CD are maximal large itemsets. (ii) ACD, BCD are generated by IA\_Gen. (iii) ABC, ABD, ACD and BCD Are maximal large itemsets (iv) ABCD is generated by IA\_Gen.

**Figure 1: IA\_Gen**

In our simple example, the itemsets  $\{a, c, d\}$  and  $\{b, c, d\}$  will be generated in the first iteration, among others. The database is then scanned to count the supports of the candidate itemsets. The large ones will be added to the set of large itemsets, and the whole process repeats. Finally, the itemset  $\{a, b, c, d\}$  is generated in the second iteration.

Our candidate set generation function IA\_Gen, based on lattice theory[8], adopts the strategy of generating *large* candidate itemsets as soon as possible, using a suggested set of large itemsets as a knowledge base. However, instead of using all large itemsets known so far to generate a batch of candidate itemsets, IA\_Gen uses only the *maximal* large itemsets for candidate generation. An (already known) large itemset is maximal if it is not a proper subset of another (already known) large itemset.

We present the IA\_Gen, candidate set generation function and the IA\_FindLarge algorithm, which uses IA\_Gen to discover frequent itemsets in updated database. Thus replacing Apriori and Apriori\_Gen by IA\_FindLarge and IA\_Gen allows us to significantly reduce the number of database scan required for mining association rules. Proposed algorithms address the following properties.

- IA\_FindLarge generates less candidate itemsets than Apriori. A naïve algorithm, which generates all possible itemsets as the candidate set, can of course discover all frequent itemsets in one single database scan. This algorithm, however, is clearly infeasible because the candidate set would be too large. IA\_FindLarge only generates candidate itemsets earlier and in fewer passes.
- To apply IA\_FindLarge successfully, one needs to supply it with suggested frequent itemsets as a "knowledge base". Although these suggested itemsets are not necessary, their presence will significantly improve the performance of IA\_FindLarge. To mine frequent itemsets in the updated database, we can use the set of frequent itemsets found from the old database as the suggested set of frequent itemsets and apply IA\_FindLarge. Alternatively, one can take a sample of the database, apply Apriori on the sample to get a rough estimate of the frequent itemsets, and use it as the suggested set of frequent itemsets for IA\_FindLarge. If we take a very small sample, then the frequent itemsets discovered from it would not be a good estimate of the frequent itemsets of the whole database. IA\_FindLarge would not perform much better than Apriori because the "knowledge base" is not good enough. On the other hand, if we take a vary large sample, then the I/O cost of applying Apriori on the large sample to obtain the frequent itemsets estimate would be substantial, essentially wiping out the benefit achieved by IA\_FindLarge. Sampling plus IA\_FindLarge is thus a viable option for fast mining of frequent itemsets.
- The number of database scans saved by IA\_FindLarge over Apriori depends on the *accuracy* of the suggested large itemsets. As an extreme case, if the suggested itemsets cover all the large itemsets in the database, then IA\_FindLarge requires only 2 database scans. This number is independent of the size of the largest large itemsets.

#### 4. The Apriori Algorithm

Conceptually, finding large itemsets from database transactions involves keeping a count for every itemset. However, since the number of possible itemsets is

exponential to the number of items in the database, it is impractical to count every subset we encounter in the database transactions. The Apriori algorithm tackles this combinatorial explosion problem by using an iterative approach to count the itemsets. First, itemsets containing only one item (1-itemsets) are counted, and the set of large 1-itemsets ( $L_1$ ) is found. Then, a set of possibly large 2-itemsets is generated using the function Apriori\_Gen. Since for an itemset of size  $n$  to be large, all its size  $n-1$  subsets must also be large, Apriori\_Gen only generates those 2-itemsets whose size one subsets are all in  $L_1$ . This set is the candidate set of size-2 itemsets,  $C_2$ . For example, if  $L_1 = \{\{c\}, \{e\}, \{g\}, \{j\}\}$ ,  $C_2$  would be  $\{\{c, e\}, \{c, g\}, \{c, j\}, \{e, g\}, \{e, j\}, \{g, j\}\}$ .

After  $C_2$  is generated, the database is scanned once again to determine the support counts of the itemsets in  $C_2$ . Those with their support counts larger than the support threshold are put into the set of size-2 large itemsets,  $L_2$ .  $L_2$  is then used to generate  $C_3$  in a similar manner: all size-two subsets of every element in  $C_3$  must be in  $L_2$ . So, if  $L_2$  in our previous example turns out to be  $\{\{c, e\}, \{c, g\}, \{c, j\}, \{g, j\}\}$ ,  $C_3$  would be  $\{\{c, g, j\}\}$ . Note that the itemset  $\{c, e, j\}$  is not generated because not all of its size-two subsets are in  $L_2$ . Again, the database is scanned once more to find  $L_3$  from  $C_3$ . This candidate set generation-verification process is continued until no more candidate itemsets can be generated. Finally, the set of large itemsets is equal to the union of all the  $L_i$ 's.

The iterative nature of the Apriori algorithm implies that at least  $n$  database passes are needed to discover all the large itemsets if the biggest large itemsets are of size  $n$ . Since database passes involve slow access, to increase efficiency, we should minimize the number of database passes during the mining process. One solution is to generate bigger-sized candidate itemsets as soon as possible, so that their supports can be counted early. With Apriori\_Gen, unfortunately, the only piece of information that is useful for generating new candidate itemsets during the  $n$ -th iteration is the size- $(n-1)$  large itemsets,  $L_{n-1}$ . Information from other  $L_i$ 's ( $i < n-1$ ) and  $C_i$ 's ( $i \leq n-1$ ) are not useful because it is already subsumed in  $L_{n-1}$ . As a result, we cannot generate candidate itemsets larger than  $n$ .

## 5. IA\_FindLarge Algorithm

The iterative nature of the Apriori algorithm implies that at least  $n$  database passes are needed to discover all the large itemsets if the biggest frequent itemsets are of size  $n$ . Since database passes involve slow access, to increase efficiency, we should minimize the number of database passes during the mining process. One solution is to generate bigger-sized candidate itemsets as soon as possible, so that their supports can be counted early. With Apriori\_Gen, unfortunately, the only piece of information that is useful for generating new candidate itemsets during the  $n$ -th iteration is the size- $(n-1)$  large itemsets,  $L_{n-1}$ . Information from other  $L_i$ 's ( $i < n-1$ ) and  $C_i$ 's ( $i \leq n-1$ ) are not useful because it is already subsumed in  $L_{n-1}$ . As a result, we cannot generate candidate itemsets larger than  $n$ .

Now, suppose one is given a set of suggested frequent itemsets  $S\_frequent$ . We can use this set as additional information to generate frequent itemsets early. During the first iteration, besides counting the supports of size-1 itemsets, we can also count the supports of the elements in  $S\_frequent$  as well in updated database. After the first iteration, we thus have a (partial) set of frequent itemsets of various sizes,  $N\_frequent$ . These itemsets include all frequent 1-itemsets as well as those itemsets in  $S\_frequent$  that are verified frequent. We can now follow the principle of Apriori to generate candidate itemsets based on  $N\_frequent$ . The only problem remains is how to generalize Apriori\_Gen to compute a set of multiple-sized candidate itemsets from a set of multiple-sized frequent itemsets ( $N\_frequent$ ) *efficiently*.

**function IA\_FindLarge (S\_frequent )**

```

{
    G_candidate =  $\phi$ 
    for each  $x \in S\_frequent$  // Canonicalization
    {
        I = {  $p \mid p \subseteq x$  }
        G_candidate =  $I \cup G\_candidate$ 
    }
    G_candidate = { all 1-itemset }  $\cup$  G_candidate
    Scan database and compute support count of every itemset in G_candidate
    N_frequent = {  $I \mid I \in G\_candidate \wedge support(I) \geq min\_sup$  }
    While( N_frequent  $\neq \phi$  )
    {
        for each  $I \in S$ 
        {
            if( $|I| = k$ ) then
            {
                add I to  $F_k$ 
                add k to M // M: set of integer
            }
        }
        r = min(M)
        m = max(M)
        M_frequent =  $\phi$ 
        for(  $p = r+1$  to m )
        {
            Max_Fp =  $\phi$ 
            for each itemset  $I_i \in F_{p-1}$ 
            {
                for each itemset  $I_j \in F_p$ 
                {
                    if( $(I_i \cap I_j) \neq I_i$ ) then

```

```

        Max_Fp = Max_Fp ∪ Ii
        }
    }
    M_frequent = M_frequent ∪ Max_Fp
    }
    N_candidate = IA_Gen(M_frequent, r, m)
    Scan database and compute support count of itemsets in N_candidate
    N_frequent = {I | I ∈ N_candidate ∧ (support(I) ≥ min_sup)}
}
return all subsets of itemsets in M_frequent
}

```

**Figure 2:** Finding frequent itemsets

## 6. IA\_Gen Algorithm

We generalize Apriori\_Gen to a new candidate set generation function called IA\_Gen based on Lattice Theory [8]. The main idea of IA\_Gen is to generate candidate itemsets of bigger sizes early using information provided by a set of suggested frequent itemsets. Before we describe our algorithm formally, let us first illustrate the idea with an Example.

For example [13], suppose we have a database whose frequent itemsets are {a, b, c, d, e, f}, {d, e, f, g}, {e, f, g, h}, {h, i, j} and all their subsets. Assume that the sets {a, b, c, d, e, f}, {e, f, g, h}, and {d, g} are suggested frequent itemsets. During the first iteration, we count the supports of the singletons as well as those of the suggested itemsets in updated database. Assume that the suggested itemsets are verified frequent in updated database also. In the second iteration, since {a, b, c, d, e, f} is frequent, we know that its subset {d, e} is also frequent. Similarly, we can infer from {e, f, g, h} that {e, g} is also frequent. Since {d, g} is also frequent, we can generate the candidate itemset {d, e, g} and start counting its support. Similarly, the candidate itemset {d, f, g} can also be generated this way. Therefore, we have generated some size-3 candidate itemsets before we find out all size-two frequent itemsets.

Our algorithm for finding frequent itemsets, IA\_FindLarge is shown in Figure 2. The method is similar to Apriori except that:

- it takes a set of suggested set of frequent itemsets, S\_frequent as input and counts their supports during the first database scan in updated database, and
- it replaces the Apriori\_Gen function by the more general IA\_GEN function which takes the set of maximal frequent itemsets in updated database M\_frequent as input.

The algorithm consists of two stages. The first stage consists of a single database scan. Candidate 1-itemsets, as well as the suggested frequent itemsets and their subsets are counted. Any itemsets found to be frequent at this first stage is put into the set of newly found frequent itemsets (N\_frequent).

```

function IA_Gen (M_frequent, r, m )
{
  N_candidate =  $\phi$ 
  for( k= r+1 to m)
  {
    Max_Fk-1 = {I | I $\in$  M_frequent  $\wedge$  |I| = k-1}
    Ck = apriori_gen(Max_Fk-1)
    N_candidate = Ck  $\cup$  N_candidate
  }
return N_candidate
}
}

```

**Figure 3:** Generating candidate itemsets

The second stage of IA\_FindLarge is iterative. The iteration continues until no more new large itemsets can be found. At each iteration, IA\_FindLarge generates a set of candidate itemsets based on the frequent itemsets it has already discovered. As we have argued, we could apply Apriori\_Gen on the whole set of large itemsets already found. However, the drawback is that the set of frequent itemsets could be large, and that it would result in the generation of many redundant candidate itemsets. Instead, IA\_FindLarge first *canonicalizes* the set of frequent itemsets into a set of *maximal frequent* itemsets (M\_frequent) passes the maximal frequent itemsets to generate candidate itemsets in updated database ( N\_candidate).

Thus canonicalization one of the ways of compressing the information contained in the set of frequent itemsets. Suppose we know that a set  $s$  is frequent, we immediately conclude that all of its subsets are also frequent. Consider the set of itemsets with the subset operator as a lattice [15, 42]. We can then represent the set of all frequent itemsets  $L$  by the set of maximal elements of  $L$ , is defined as maximal ( $L$ ) = {  $x \in L$  |  $\forall y \in L$  [ $x \subseteq y \Rightarrow y \subseteq x$ ] }.

In above Example, where {a, b, c, d, e, f}, {d, e, f, g}, {e, f, g, h}, {h, i, j} and all their subsets are frequent, set of maximal frequent itemsets of  $L$  = {{a, b, c, d, e, f}, {d, e, f, g}, {e, f, g, h}, {h, i, j}}. Hence, only 4 itemsets are needed to represent  $L$ , which contains 86 frequent itemsets.

IA\_Gen uses the set of maximal frequent itemsets to generate candidate itemsets. The crux is how to perform the generation of candidate sets based on the compressed maximals only. We remark that the Apriori algorithm with Apriori\_Gen is in fact displaying a special case of candidate generation with canonicalization.

## 7. Conclusion

This paper described a new algorithm IA\_FindLarge for discovering large itemsets in a transaction database. IA\_FindLarge uses a new candidate set generation algorithm IA\_Gen which takes a set multiple-sized large itemsets to generate multiple-sized

candidate itemsets. Given a reasonably accurate suggested set of large itemsets, IA\_Gen allows big-sized candidate itemsets to be generated and processed early. This results in significant to reducing the number of database scan compared with traditional Apriori-based mining algorithms. To obtain a good suggested set, sampling techniques can be applied. IA\_FindLarge is thus an efficient and practical algorithm for mining association rules.

## References

- [1] Agrawal R., Imielinski T., and Swami A. (1993). Mining association rules between sets of items in large databases. In Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, Washington, USA.
- [2] Cheung D., Han J., Ng V., and Wong C. Y. (1996). Maintenance of discovered association rules in large databases: An incremental updating technique. In Proc. of the 12<sup>th</sup> Intl. Conf. on Data Engineering.
- [3] Cheung D., Lee S., and Kao B. (1997). A general incremental technique for maintaining discovered association rules. In Proc. of the 5<sup>th</sup> Intl. Conf. on Database Systems for Advanced Applications, pages 1–4.
- [4] Ganti V., Gehrke J., and Ramakrishnan R. (2000). Demon: Mining and monitoring evolving data. In Proc. of the 16<sup>th</sup> Int'l Conf. on Data Engineering, pages 439–448,
- [5] Gouda K. and Zaki M. (2001). Efficiently mining maximal frequent itemsets. In Proc. of the 1<sup>st</sup> IEEE Int'l Conference on Data Mining, San Jose, USA.
- [6] Lee S. and Cheung D. (1997). Maintenance of discovered association rules: When to update? In Research Issues on Data Mining and Knowledge Discovery.
- [7] Agrawal R. and Srikant R. (1994). Fast algorithms for mining association rules in large databases. In Proc. Of the Twentieth International Conference on Very Large Databases, pages 487-499.
- [8] Davey B.A. and Priestley H.A. (1990). Introduction to Lattices and Order. Cambridge University Press.
- [9] Thomas S., Bodagala S., Alsabti K., and Ranka S. (1997). An efficient algorithm for the incremental updationof association rules. In Proc. of the 3<sup>rd</sup> ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining.
- [10] Veloso A., Meira Jr W., de Carvalho M. B., Possas B., Parthasarathy S., and Zaki M. (2002). Mining frequent itemsets in evolving databases. In Proc. of the 2<sup>nd</sup> SIAM Int'l Conf. on Data Mining, Arlington, USA.
- [11] Veloso A., Meira Jr. W., de Carvalho M. B., Rocha B., Parthasarathy S., and Zaki M. (2002). Efficiently mining approximate models of associations in evolving databases. In Proc. of the 6<sup>th</sup> Int'l Conf. on Principles and Practices of Data Mining and Knowledge Discovery in Databases, Helsinki, Finland.
- [12] Zaki M. and Hsiao C. (2002). Charm: An efficient algorithm for closed itemset mining. In Proc. of the 2<sup>nd</sup> SIAM Int'l Conf. on Data Mining, Arlington.

- [13] Yip C., Loo K.K., Kao B., Cheung D., and Cheng C.K. (2002). LGen- A lattice-based candidate set generation algorithm for I/O efficient association rule mining. Information system archive vol. 27, issue 1.