ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology
Volume 1, Issue 5, July 2012

Frequent Itemset Mining Technique in Data Mining

Sanjaydeep Singh Lodhi
Department of Computer Application (Software Systems)
S.A.T.I (Govt. Autonomous collage) , Vidisha, (M.P), India

Premnarayan Arya
Asst. Prof. Dept. of CA (Software Systems)
S.A.T.I (Govt. Autonomous collage) , Vidisha, (M.P), India

Dilip Vishwakarma
Department of Computer Application (Software Systems)
S.A.T.I (Govt. Autonomous collage) , Vidisha, (M.P), India

Abstract

In computer science and data mining, Apriori is a classic algorithm for learning association rules. Apriori is designed to operate on databases containing transactions (for example, collections of items bought by customers, or details of a website frequentation). Frequent itemsets play an essential role in many data mining tasks that try to find interesting patterns from databases, such as association rules, correlations, sequences, episodes, classifiers, clusters and many more of which the mining of association rules is one of the most popular problems. In this paper, we take the classic Apriori algorithm, and improve it quite significantly by introducing what we call a vertical sort. We then use the large dataset, web documents to contrast our performance against several state-of-the-art implementations and demonstrate not only equal efficiency with lower memory usage at all support thresholds, but also the ability to mine support thresholds as yet un-attempted in literature. We also indicate how we believe this work can be extended to achieve yet more impressive results. We have demonstrated that our implementation produces the same results with the same performance as the best of the state-of-the art implementations. In particular, we have started with the classic algorithm for this problem and introduced a conceptually simple idea, sorting the consequences of which have permitted us to outperform all of the available state-of-the-art implementations.

Keywords: Data Mining, Apriori Algorithm, Frequent Itemset Mining.

Introduction

Data mining is the automated process of extracting useful patterns from typically large quantities of data. Different types of patterns capture different types of structures in the data: A pattern may be in the form of a rule, cluster, set, sequence, graph, tree, etc. and each of these pattern types is able to express different structures and relationships present in the data. The problem of mining probabilistic is frequent itemsets in uncertain or probabilistic databases, including mining the probability distribution of support. One challenge is to compute the probability distribution efficiently. The other challenge is to mine probabilistic frequent itemsets using this computation. Uses an Apriori style algorithm for the itemset mining task. The first algorithm based on the FP-Growth idea that is able to handle probabilistic data. It solves the problem much faster than the Apriori style approach. Finally, the problem can be solved in the GIM framework, leading to the GIM-PFIM algorithm. This improves the run time by orders of magnitude, reduces the space usage and also allows the problem to be viewed more naturally as a vector based problem. The vectors are real valued probability vectors and the search can benefit from subspace projections. The Generalized Interaction Mining (GIM) methods for solving interaction mining are problems at the abstract level. Since GIM leaves the semantics of the interactions, their interestingness measures and the space in which the interactions are to be mined as flexible components; it creates a layer of abstraction between a problem's definition/semantics and the algorithm used to solve it; allowing both to vary independently of each other. This was achieved by developing a consistent but general geometric computation model based on vectors and vector valued functions. For most problems, the space required is provably linear in the size of the data set. The run-time is provably linear in the number of interactions that need to be examined. These properties allow it to outperform specialist algorithms when applied to specific interaction mining problems [1] and [3].

In this paper, we take the classic algorithm for the problem, Apriori, and improve it quite significantly by introducing what we call a vertical sort. We have demonstrated that our implementation produces the same results with the same performance as the best of the state-of-the art implementations. But, whereas they blow their memory in order to decrease the support threshold, the memory utilization of our implementation remains relatively constant. We believe that this work opens up many avenues for yet more pronounced improvement.

Related Works on Frequent Itemset Mining

The approach proposed by Chui et. al computes the expected support of itemsets by summing all itemset probabilities in their U-Apriori algorithm. Later, they additionally proposed a probabilistic filter in order to prune candidates early.

The UF-growth algorithm is proposed. Like U-Apriori, UF-growth computes frequent itemsets by means of the expected support, but it uses the FP-tree approach in order to avoid expensive candidate generation. In contrast to our probabilistic approach, itemsets are considered frequent if the expected support exceeds minSup. The main drawback of this estimator is that information about the uncertainty of the expected support is lost; ignore the number of possible worlds in which an itemset is frequent. Proposes exact and sampling-based algorithms find likely frequent items in streaming probabilistic data. However, they do not consider itemsets with more than one item. The current state-of the art (and only) approach for probabilistic frequent itemset mining (PFIM) in uncertain databases was proposed. Their approach uses an Apriori-like algorithm to mine all probabilistic frequent itemsets and the poisson binomial recurrence to compute the support probability distribution function (SPDF).

We provide a faster solution by proposing the first probabilistic frequent pattern growth approach (ProFP-Growth), thus avoiding expensive candidate generation and allowing us to perform PFIM in large databases. Furthermore, use a more intuitive generating function method to compute the SPDF.

Existing approaches in the field of uncertain data management and mining can be categorized into a number of research directions. Most related to our work are the two categories “probabilistic databases” and “probabilistic query processing”.

The uncertainty model used in the approach is very close to the model used for probabilistic databases. A probabilistic database denotes a database composed of relations with uncertain tuples, where each tuple is associated with a probability denoting the likelihood that it exists in the relation. This model, called “tuple uncertainty”, adopts the possible worlds semantics. A probabilistic database represents a set of possible “certain” database instances (worlds), where a database instance corresponds to a subset of uncertain tuples. Each instance (world) is associated with the probability that the world is “true”. The probabilities reflect the probability distribution of all possible database instances.

In the general model description, the possible worlds are constrained by rules that are defined on the tuples in order to incorporate object (tuple) correlations.

The ULDB model proposed, which is used in Trio, supports uncertain tuples with alternative instances which are called x-tuples. Relations in ULDB are called x-relations containing a set of x-tuples. Each x-tuple corresponds to a set of tuple instances which are assumed to be mutually exclusive, i.e. no more than one instance of an x-tuple can appear in a possible world instance at the same time. Probabilistic top-k query approaches are usually associated with uncertain databases using the tuple uncertainty model. The approach proposed was the first approach able to solve probabilistic queries efficiently under tuple independency by means of dynamic programming techniques [1] and [4].

Recently, a novel approach was proposed to solve a wide class of queries in the same time complexity, but in a more elegant and also more powerful way using generating functions.

The efficient data structures and techniques used in frequent itemset mining such as TID-lists, FP-tree, which adopts a prefix tree structure as used in FP-growth, and the hyper-linked array based structure as used in H-mine can no longer be used as such directly on the uncertain data. Therefore, recent work on frequent itemset mining in uncertain data that inherits the breadth-first

and depth-first approaches from traditional frequent itemset mining adapts the data structures to the probabilistic model.

U-Apriori is based on a level wise algorithm and represents a baseline algorithm for mining frequent itemsets from uncertain datasets. Because of the generate and test strategy, level by level, the method does not scale well.

UCP-Apriori is based on the decremental pruning technique which consists in maintaining an upper bound of the support and decrementing it while scanning the dataset. The itemset is pruned as soon as its most optimistic value falls below the threshold. This approach represents the state of the art for mining frequent patterns from uncertain data with a generate-and-prune strategy.

UF-growth extends the FP-Growth algorithm. It is based on a UF-tree data structure (similar to FP-tree). The difference with the construction of a FPtree is that a transaction is merged with a child only if the same item and the same expected support exist in the transaction and in the child node, leading to a far lower compression ratio as in the original FP-tree. The improvements consist in discretization of the expected support to avoid the huge number of different values and in adapting the idea of co-occurrence frequent itemset tree (COFItree).

The UF-trees are built only for the first two levels. It then enumerates the frequent patterns by traversing the tree and decreasing the occurrence counts.

Aggarwal et al. extended several existing classical frequent itemset mining algorithms for deterministic data sets, and compared their relative performance in terms of efficiency and memory usage. The UH-mine algorithm, proposed in their paper, provides the best trade-offs. The algorithm is based on the pattern growth paradigm. The main difference with UF-growth is the data structure used which is a hyperlinked array.

The limitations of these existing methods are the ones inherited from the original methods. The size of the data for the level-wise generate-and-test techniques affects their scalability and the pattern-growth techniques require a lot of memory for accommodating the dataset in the data structures, such as the FP-tree, especially when the transactions do not share many items. In the case of uncertain data, not only the items have to be shared for a better compression but also the existence probabilities, which is often not the case [2] and [3].

Many itemset mining algorithms have been proposed since association rules were introduced.

Most algorithms can be broadly classified into two groups, the item enumeration which operate with a general-to-specific search strategy, and the row enumeration techniques which find specific itemsets first. Broadly speaking, item enumeration algorithms are most effective for data sets where |T| >> |I|, while row enumeration algorithms are effective for data sets where |T| << |I|, such as for micro-array data. Furthermore, a specific to general strategy can be useful for finding maximal frequent itemsets in dense databases. Item enumeration algorithms mine subsets of an itemset I0 before mining the more specific itemset I0. Only those itemsets for which all (or some) subsets are frequent are generated making use of the anti-monotonic property of support. This property is also known as the Apriori property. Apriori-like algorithms search for frequent itemsets in a breadth first generate-and-test manner, where all itemsets of length k are generated before those of length k+1. In the candidate generation step, possibly frequent k + 1 itemsets are generated from the already mined k-itemsets prior to counting items, making use of the Apriori property. This creates a high memory requirement, as all candidates must remain in main memory. For support counting (the test step), a pass is made over the database while checking whether the candidate itemsets occur in at least minSup transactions. This requires subset checking a computationally expensive task especially when the transaction width is high. While methods such as hash-trees or bitmaps have been used to accelerate this step, candidate checking remains expensive. Various improvements have been proposed to speed up aspects of Apriori. For example, proposed a hash based method for candidate generation that reduces the number of candidates generated, thus saving considerable space and time. Argues for the use of the Trie data structure for the subset checking step, improving on the hash tree approach. Apriori style algorithms make multiple passes over the database, at least equal to the length of the longest frequent itemset, which incurs considerable I/O cost. GLIMIT does not perform candidate generation or subset enumeration, and generates itemsets in a depth first fashion using a single pass over the transposed data set.

Frequent pattern growth (FP-Growth) type algorithms are often regarded as the fastest item enumeration algorithms. FP-Growth generates a compressed summary of the data set using two passes in a cross referenced tree, the FP-tree, before mining itemsets by traversing the tree and recursively projecting the database into sub-databases using conditional FP-Trees. In the first database pass, infrequent items are discarded, allowing some reduction in the database size. In the the second pass, the complete FP-Tree is built by collecting common prefixes of transactions and incrementing support counts at the nodes. at the same time, a header table and node links are maintained, which allow easy access to all nodes given an item. The mining step operates by following these node links and creating (projected) FPTrees that are conditional on the presence of an item (set), from which the support can be obtained by a traversal of the leaves. Like GLIMIT, it does not perform any candidate generation and mines the itemsets in a depth first manner while still mining all subsets of an itemset I0 before mining I0. FP-Growth is very fast at reading from the FP-tree, but the downside is that the FP-tree can become very large and is expensive to generate, so this investment does not always pay off.

Further, the FP-Tree may not fit into memory. In contrast, GLIMIT uses only as much space as is required and is based on vector operations, rather than tree projections. It also uses a purely depth first search [5] and [6].

Frequent Pattern Mining Useful in Building Predictive Models

The recent studies of pattern mining have given more attention to discovering patterns that are interesting, significant, discriminative and so forth, than simply frequent. Does this imply that the frequent patterns are not useful anymore?

The amount of available literature related to frequent pattern mining (FPM) reflects the reason for calling it a focused theme in data mining. Due to the evolution of the modern information technology, collecting, combining, transferring and storing huge amounts of data could be done at very low costs. As a consequence more sophisticated methods of mining associations from such data have been introduced, expanding the typical problem of analyzing the associations of commonly found items in a shopping list [3] into discovering interesting and useful patterns from large, complex and dense databases. A frequent pattern in a pattern database may be a set of items, or a subsequence, a sub-tree or a sub-graph that appears across the database with a frequency that is not less than a pre-defined threshold value [2]. A *frequent item set* is a set of items appearing together more frequently, such as bread and butter, for example, in a transaction database [2]. A frequent sequential pattern may be a frequently seen subsequence from a database of an ordered list of items [4], while frequent sub-trees of a database of trees may represent the frequently linked web pages in a weblog database. Frequent sub-graphs in a graph database may be, for example, a more frequently appeared motif in a chemoinformatics database. Such patterns discovered by FPM algorithms could be used for classification, clustering, finding association rules, data indexing, and other data mining tasks.

Ever since the introduction of the first FPM algorithm by Agrawal et.al., plenty of approaches related to pattern mining problem are added, including new algorithms as well as improvements to existing algorithms, making the FPM a scalable and a solvable problem]. We can find studies dealing with complexity issues, database issues, search space issues, scalability issues of FPM in abundance. After about two decades since the concept emerged, with thousands of research publications accumulated in the literature, one may wonder whether the problem of FPM is solved at an acceptable level on most of the data mining tasks. On the other hand there is a common criticism about the FPM approaches for (1) producing unacceptably large set of frequent patterns, which limits the usage of the patterns, (2) contains large amount of redundant patters, (3) most of the frequent patterns are not significant, i.e, they do not discriminate between the classes [6, 14]. The arguments of these studies return a fair question; are frequent patterns not useful anymore? In this paper we investigate, with the support of an empirical evaluation using a representative set of pattern mining algorithms, how far the frequent patterns are useful in building predictive models.

Discovery of frequent itemsets is a two step procedure, the search of frequent patterns and support count. The apriori principle, which is introduced and substantially improved later, limits the search space using the monotonicity property of the support of sets [5]. The methods using horizontal [3, 10] and vertical [12] layouts for organizing the transaction database support efficient scanning of the database. Brief descriptions of various studies related to improving efficiency, scalability, usage of computer memory and so forth, in the frequent itemset mining algorithm [2] and [14].

Despite the optimizations introduced to improve the performance of the frequent itemset mining algorithm, it often generates substantially large amount of frequent itemsets, which limit the possibility of using them meaningfully. To overcome this problem, the concepts of maximal frequent itemsets and closed frequent itemsets were proposed. The *maximal frequent itemset* is a frequent itemset *I*, which is included in no other frequent itemset. A frequent itemset is called a *closed frequent itemset* if there does not exist a super frequent itemset that has the same support. The extra computational burden related to determining whether a frequent itemset is closed or maximal are being solved by approaches such as keeping the track of the Transaction ID lists, using hash functions, or maintaining a frequent pattern tree similar to FP-tree. Maximal frequent itemset mining approaches use techniques such as vertical bit maps for the Transaction ID list, or apriori based pruning for search space.

Proposed Frequent Itemset Mining Techniques

The Apriori-based algorithms find frequent itemsets based upon an iterative bottom-up approach to generate candidate itemsets. Since the first proposal of association rules mining by R. Agrawal, many researchers have been done to make frequent itemsets mining scalable and efficient. But there are still some deficiencies that Apriori based algorithms suffered from, which include: too many scans of the transaction database when seeking frequent itemsets, large amount of candidate itemsets generated unnecessarily and so on.

The proposed of our method is the classical A Priori algorithm. Our contributions are in providing novel scalable approaches for each building block. We start by counting the support of every item in the dataset and sort them in decreasing order of their frequencies. Next, we sort each transaction with respect to the frequency order of their items. We call this a horizontal sort. We also keep the generated candidate itemsets in horizontal sort. Furthermore, we are careful to generate the candidate itemsets in sorted order with respect to each other. We call this a vertical sort. When itemsets are both horizontally and vertically sorted, we call them fully sorted. As we show, generating sorted candidate itemsets (for any size k), both horizontally and vertically, is computationally free and maintaining that sort order for all subsequent candidate and frequent itemsets requires careful implementation, but no cost in execution time. This conceptually simple sorting idea has implications for every subsequent part of the algorithm. In particular, as we show, having transactions, candidates, and frequent itemsets all adhering to the same sort order has the following advantages: Generating candidates can be done very efficiently. Indices on lists of candidates can be efficiently generated at the same time as are the candidates. Groups of similar candidates can be compressed together and counted simultaneously. Candidates can be compared to transactions in linear time. Better locality of data and cache-consciousness is achieved. Our particular choice of sort order (that is, sorting the items least frequent first) allows us to with minimal cost entirely skip the candidate pruning phase.

Effective Candidate Generation

Candidate generation is the important first step in each iteration of A Priori. Typically it has not been considered a bottleneck in the algorithm and so most of the literature focuses on the support counting. However, it is worth pausing on that for a moment. Modern processors usually manage about thirty million elementary instructions per second. We devote considerable attention to improving the efficiency of candidate generation, too.

Let us consider generating candidates of an arbitrarily chosen size, k + 1. We will assume that the frequent k-itemsets are sorted both horizontally and vertically. The (k − 1) × (k − 1) technique generates candidate (k+1) itemsets by taking the union of frequent k-itemsets. If the first k−1 elements are identical for two distinct frequent k-itemsets, fi and fj , we call them near-equal and denote their near-equality by fi = fj . Then, classically, every frequent itemset fi is compared to every fj and the candidate fi ∪ fj is generated whenever fi = fj. However, our method needs only ever compare one frequent itemset, fi, to the one immediately following it, f_{i+1}.

A crucial observation is that near-equality is transitive because the equality of individual items is transitive. So, if $f_i = f_{i+1}, \ldots, f_{i+m-2} = f_{i+m-1}$ then we know that $(\forall j, k) < m$, $f_{i+j} = f_{i+k}$.

Recall also that the frequent k-itemsets are fully sorted (that is, both horizontally and vertically), so all those that are near-equal appear contiguously. This sorting taken together with the transitivity of near-equality is what our method exploits.

In this way, we successfully generate all the candidates with a single pass over the list of frequent k-itemsets as opposed to the classical nested-loop approach. Strictly speaking, it might seem that our processing of $\binom{m}{2}$ candidates effectively causes extra passes, but it can be shown using the A Priori Principle that m is typically much less than the number of frequent itemsets. First, it remains to be shown that our one pass does not miss any potential candidates. Consider some candidate $c = \{i_a, \ldots, i_k\}$. If it is a valid candidate, then by the A Priori Principle, $fi = \{i_1, \ldots, i_{k-2}, i_{k-1}\}$ and $f_j = \{i_1, \ldots, i_{k-2}, i_k\}$ are frequent. Then, because of the sort order that is required as a precondition, the only frequent itemsets that would appear between fi and fj are those that share the same (k − 2)-prefix as they do. The method described above merges together all pairs of frequent itemsets that appear contiguously with the same (k − 2)-prefix. Since this includes both f_i and f_j , $c = f_i \cup f_j$ must have been discovered.

Candidate compression

Let us return to the concern of generating $\binom{m}{2}$ candidates from each group of m near-equal frequent k-itemsets. Since each group of $\binom{m}{2}$ candidates share in common their first k−1 items, we need not repeat the information. As such, we can compress the candidates into a super-candidate. This new super-candidate still represents all $\binom{m}{2}$ candidates, but takes up much less space in memory and on disk. More importantly, however, we can now count these candidates simultaneously. Suppose we wanted to extract the individual candidates from a super-candidate.

Ideally this will not be done at all, but it is necessary after support counting if at least one of the candidates is frequent because the frequent candidates need to form a list of uncompressed frequent itemsets. Fortunately, this can be done quite easily. The candidates in a super-candidate c = (c_w, c_s) all share the same prefix: the first k − 1 items of cs. They all have a suffix of size

$(k + 1) − (k − 1) = 2$

By iterating in a nested loop over the last c_{w-k+1} item of cs, we produce all possible suffices in sorted order.

Consequence Candidate Generation

There is another nice consequence of generating sorted candidates in a single pass: we can efficiently build an index for retrieving them. In our implementation and in the following example, we build this index on the least frequent item of each candidate (k + 1)-itemset.

The structure is a simple two-dimensional array. Candidates of a particular size k+1 are stored in a sequential file, and this array provides information about offsetting that file. Because of the sort on the candidates, all those that begin with each item I appear contiguously. The exact location in the file of the first such candidate is given by the i^{th} element in the first row of the array. The ith element in the second row of the array indicates how many bytes are consumed by all (k + 1)-candidates that begin with item i.

On the precondition of sorting

This is a very feasible requirement. The first candidates that are produced contain only two items. If one considers the list of frequent items, call it F1, then the candidate 2-itemsets are the entire cross-product F1 × F1. If we sort F1 first, then a standard nested loop will induce the order we want. That is to say, we can join the first item to the second, then the third, then the fourth, and so on until the end of the list. Then, we can join the second item to the third, the fourth, and so on as well. Continuing this pattern, one will produce the entire cross-product in fully sorted order. This initial sort is a cost we readily incur for the improvements it permits.

After this stage, there are only two things we ever do: generate candidates and detect frequent itemsets by counting the support of the candidates. Because in the latter we only ever delete never add nor will change itemsets from the sorted list of candidates, the list of frequent itemsets retain the original sort order. Regarding the former, there is a nice consequence of generating candidates in our linear one pass fashion: the set of candidates is itself sorted in the same order as the frequent itemsets from which they were derived. Recall that candidates are generated in groups of near-equal frequent k-itemsets. Because the frequent k-itemsets are already sorted, these groups, relative to each other, are too. As such, if the candidates are generated from a sequential scan of the frequent itemsets, they will inherit the sort order with respect to at least the first k−1 items. Then, only the ordering on the kth and $(k+1)^{th}$ items (those not shared among the members of the group) need be ensured. That two itemsets are near-equal can be equivalently stated as that the itemsets differ on only the k^{th} item. So, by ignoring the shared items we can consider a group of near-equal itemsets as just a list of single items. Since the itemsets were sorted and this new list is made of only those items which differentiated the itemsets, the new list inherits the sort order. Thus, we use exactly the same method as with F1 to ensure that each group of candidates is sorted on the last two items. Consequently, the entire list of candidate (k + 1)-itemsets is fully sorted.

Candidate Pruning

When A Priori was first proposed, its performance was explained by its effective candidate generation. What makes the candidate generation so effective is its aggressive candidate pruning. We believe that this can be omitted entirely while still producing nearly the same set of candidates. Stated alternatively, after our particular method of candidate generation, there is little value in running a candidate pruning step.

In recent, the probability that a candidate is generated is shown to be largely dependent on its best testset that is, the least frequent of its subsets. Classical A Priori has a very effective candidate generation technique because if any itemset $c \setminus \{c_i\}$ for $0 \le i \le k$ is infrequent the candidate $c = \{c_0, \ldots, c_k\}$ is pruned from the search space. By the A Priori Principle, the best testset is guaranteed to be included among these. However, if one routinely picks the best testset when first generating the candidate, then the pruning phase is redundant.

In our method, on the other hand, we generate a candidate from two particular subsets, $fk = c \setminus \{c_k\}$ and $f_{k-1} = c \setminus \{c_{k-1}\}$. If either of these happens to be the best testset, then there is little added value in a candidate pruning phase that checks the other k−2 size k subsets of c. Because of our least-frequent-first sort order, f_0 and f_1 correspond exactly to the subsets missing the most frequent items of all those in c. We observed that usually either f_0 or f_1 is the best test set.

We are also not especially concerned about generating a few extra candidates, because they will be indexed and compressed and counted simultaneously with others, so if we do not retain a considerable number of prunable candidates by not pruning, then we do not do especially much extra work in counting them, anyway.

Support Counting using Indexing

It was recognized quite early that A Priori would suffer a bottleneck in comparing the entire set of transactions to the entire set of candidates for every iteration of the algorithm. Consequently, most A Priori -based research has focused on trying to address this bottleneck. Certainly, we need to address this bottleneck as well. The standard approach is to build a prefix trie on all the candidates and then, for each transaction, check the trie for each of the k-itemsets present in the transaction. But this suffers two traumatic consequences on large datasets. First, if the set of candidates is large and not heavily overlapping, the trie will not fit in memory and then the algorithm will thrash about exactly as do the other tree-based algorithms. Second, generating every possible itemset of size k from a transaction t = {t_0, . . . , t_{w-1}} produces $\binom{w}{k}$ possibilities. Even after pruning infrequent items with a support threshold of 10%, w still ranges so high.

Instead, we again exploit the vertical sort of our candidates using the index we built when we generated them. To process that same transaction t above, we consider each of the w − k first items in t. For each such item ti we use the index to retrieve the contiguous block of candidates whose first element is ti. Then, we compare the suffix of t that is {t_i, t_{i+1}, . . . , t_{w-1}} to each of those candidates.

Proposed Vertically-Sorted A Priori algorithm

Our method naturally does this because it operates in a sequential manner on prefaces of sorted lists. Work that is to be done on a particular contiguous block of the data structure is entirely done before the next block is used, because the algorithm proceeds in sorted order and the blocks are sorted. Consequently, we fully process blocks of data before we swap them out. Our method probably also performs decently well in terms of cache utilization because contiguous blocks of itemsets will be highly similar given that they are fully sorted. Perhaps of even more importance is the independence of itemsets. The candidates of a particular size, so long as their order is ultimately maintained in the output to the next iteration, can be processed together in blocks in whatever order desired. The lists of frequent itemsets can be similarly grouped into blocks, so long as care is taken to ensure that a block boundary occurs between two itemsets fi and fi+1 only when they are not near-equal. The indices can also be grouped into blocks with the additional advantage that this can be done in a manner corresponding exactly to how the candidates were grouped. As such, all of the data structures can be partitioned quite easily, which lends itself quite nicely to the prospects of parallelization and fault tolerance.

Frequent itemsets play an essential role in many data mining tasks that try to find interesting patterns from databases, such as association rules, correlations, sequences, episodes, classifiers, clusters. A Priori, and improve it quite significantly by introducing what we call a vertical sort.

Algorithm: The revised Vertically-Sorted A Priori algorithm:

INPUT: *A dataset D and a support threshold s*
OUTPUT: *All sets that appear in at least s transactions of D F is set of frequent itemsets*
C is set of candidates
C ← U
Scan database to count support of each item in C
Add frequent items to F
Sort F least-frequent-first (LFF) by support (using quicksort)
Output F
for all f ∈ F, sorted LFF do
for all g ∈ F, supp(g) ≥ supp(f), sorted LFF do
Add {f, g} to C
end for
Update index for item f
end for
while |C| > 0 do
{Count support}
for all t ∈ D do
for all i ∈ t do
RelevantCans ← using index, compressed cans from file that start with i
for all CompressedCans ∈ RelevantCans do
if First k − 2 elements of CompressedCans are in t then
Use compressed candidate support counting technique to update appropriate
support counts
end if; end for; end for; end for
Add frequent candidates to F
Output F
Clear C
{Generate candidates}
Start ← 0
for 1 ≤ i ≤ |F| do
if i == |F| OR fi is not near-equal to fi−1 then
Create super candidate from fstart to fi−1 and update index as necessary
Start ← i
end if; end for
{Candidate pruning—not needed!}
Clear F; Reset hash; end while

We then use the large dataset, web documents to contrast our performance against several state-of-the-art implementations and demonstrate not only equal efficiency with lower memory usage at all support thresholds, but also the ability to mine support

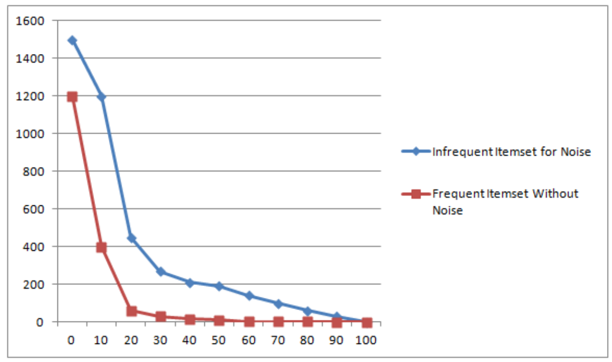
ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology
Volume 1, Issue 5, July 2012

thresholds as yet un-attempted in literature. We also indicate how we believe this work can be extended to achieve yet more impressive results. We have demonstrated that our implementation produces the same results with the same performance as the best of the state-of-the art implementations.

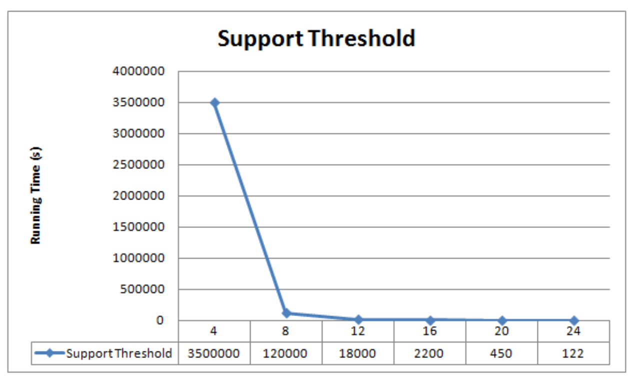
Experimental Results Analysis

The result of this paper is that the frequent itemset mining problem can now be extended too much lower support thresholds (or, equivalently, larger effective file sizes) than have even yet been considered. These improvements came at no performance cost, as evidenced by the fact that our implementation matched the state-of-the-art competitors while consuming much less memory. Prior to this work, it has been assumed that the performance of A Priori is prohibitively slow. Through these experiments we have demonstrated that at high support thresholds our implementation produces the same results with the same performance as the best of the state-of-the-art implementations. But as the support threshold decreases, the other implementations exceed memory and abort while the memory utilisation of our implementation remains relatively constant. As such, our performance continues to follow a predictable trend and our programmers can successfully mine support thresholds that are impossibly low for the other implementations.

We compare the performance of this implementation against a wide selection of the best available implementations of various frequent itemset mining algorithms. In previous works are state-of-the-art implementations of the A Priori algorithm which use a tire structure to store candidates. In order to maximally remove uncontrolled variability in the comparisons the choice of programming language is important. The correctness of our implementation's output is compared to the output of these other algorithms. Since they were all developed for the FIMI workshop and all agree on their output, it seems a fair assumption that they can serve correctly as an ”answer key”. But, nonetheless, boundary condition checking was a prominent component during development. We could generate our own large dataset against which to also run tests, but the value of doing so is minimal. The data in the web documents set comes from a real domain and so is meaningful. Constructing a random dataset will not necessarily portray the true performance characteristics of the algorithms. At any rate, the other implementations were designed with knowledge of web documents, so it is a fairer comparison. For these reasons, we used other datasets only for the purpose of verifying the correctness of our output. We test each implementation on webdocs with support thresholds of 21%, 14%, 11%, 7%, and 6%. Reducing the support threshold in this manner increases the size of the problem as observed in Figure 1 the number of candidate itemsets is implementation-dependent and in general will be less than the number in the figure.



File Size (Mb) vs. Supports (%)
Figure 1: Size of web documents dataset with noise (infrequent 1-itemsets) removed, graphed against the number of frequent itemsets.



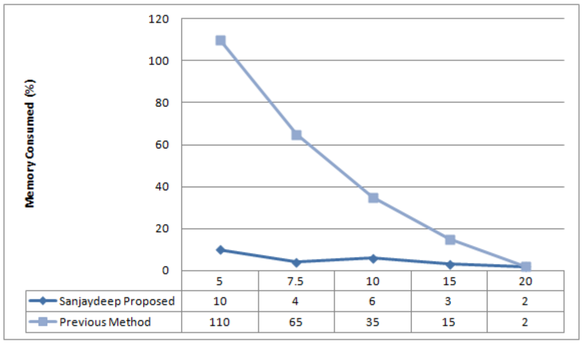
Running Time (s) vs. Support Threshold (%)
Figure 2: The Performance of Our Implementations on Webdocs Dataset

However, because our implementation uses explicit file-handling instead of relying on virtual memory, the memory requirements are effectively constant. However, those of all the other algorithms grow beyond the limits of memory and consequently cannot initialize. Without the data structures, the programs must obviously abort. Figure 2 described the Performance of our implementations on web-docs

ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology
Volume 1, Issue 5, July 2012

dataset with respect to Running Time (s) vs. Support Threshold (%).

Figure 2 described the Memory Usage of Our implementations on webdocs as Measured by final stages of execution with respect to Memory Consumed (%) vs. Supports Thresholds (%)

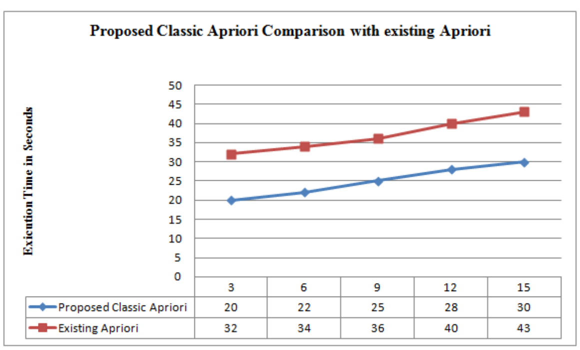


Memory Consumed (%) vs. Supports Thresholds (%)

Figure 3: Memory Usage of Our implementations on webdocs as Measured by final stages of execution

Proposed Classic Apriori Comparison with existing Apriori and FP-growth algorithm

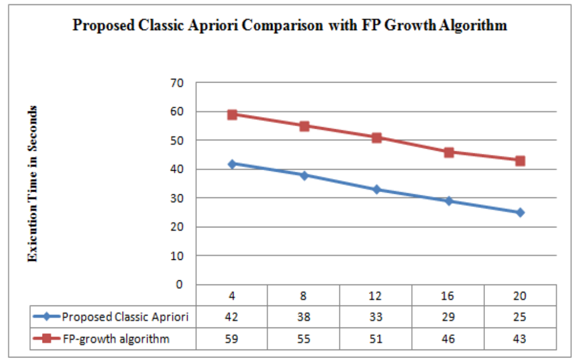
Figure 4 shows the running time of proposed classic Apriori frequent pattern mining approach with existing Apriori based multilevel frequent pattern mining algorithm on our created database with respect to the minimum support threshold at 1 with the minimum support.



Execution Time (seconds) vs. Minimum Support (%)

Figure 4 Proposed Classic Apriori Comparison with existing Apriori Graph

Fig.5 shows the running time of proposed classic Apriori frequent pattern mining approach with FP-growth algorithm on our created database with respect to the minimum support threshold at 1 with the minimum support. The result shows as minimum support decreases, the running time increases. The new approach runs faster than the existing algorithms Apriori and FP growth. It should be noted that in the previous works, their FP-Growth implementation on the same benchmark webdocs dataset as do we and they report impressive running times.



Execution Time (seconds) vs. Minimum Support (%)

Figure 5 Proposed Classic Apriori Comparison with FP Growth Graph

Unfortunately, the implementation is now unavailable. The details in the accompanying paper are not sufficiently precise that we could implement their modifications to the FP-Growth algorithm. As such, no fair comparison can truly be made. Yet still, they only publish results up to 8% which is insufficient as we demonstrated in Figure 1. It is a fair hypothesis that, were their implementation available, if would suffer the same consequences as do the other trie-based implementations when the support threshold is dropped further. So, through these experiments, we have demonstrated that our implementation produces the same results with the same performance as the best of the state-of-heart implementations.

Conclusion and Future Works

Algorithm APRIORI is one of the oldest and most versatile algorithms of Frequent Pattern Mining (FPM). With sound data structures and careful implementation it has been proven to be a competitive algorithm in the contest of Frequent Itemset Mining Implementations (FIMI). APRIORI is not only an appreciated member of the FIMI community and regarded as a baseline algorithm, but its variants to find frequent sequences of itemsets, episodes, Boolean formulas and labeled graphs have proven to be efficient algorithms as well.

We have demonstrated that our implementation produces the same results with the same performance as the best of the state-of-the art implementations. But, whereas they blow their memory in order to decrease the support threshold, the memory

utilization of our implementation remains relatively constant. As such, our performance continues to follow a predictable trend and our program can successfully mine support thresholds that are impossibly low for the other implementations.
Furthermore, whereas other algorithms in the literature are being fully optimized already, we believe that this work opens up many avenues for yet more pronounced improvement. Given the locality and independence of the data structures used, they can be partitioned quite easily. We intend to do precisely that in parallelizing the algorithm. Extending the index to more than one item to improve its precision on larger sets of candidates will likely also yield significant improvement. And, of course, all the optimization tricks used in other implementations can be incorporated here. The result of this paper is that the frequent itemset mining problem can now be extended to much lower support thresholds (or, equivalently, larger effective file sizes) than have even yet been considered.

References

[1] Toon Calders, Calin Garboni and Bart Goethals, “Approximation of Frequentness Probability of Itemsets in Uncertain Data”, 2010 IEEE International Conference on Data Mining, pp-749-754.

[2] Bin Fu, Eugene Fink and Jaime G. Carbonell, “Analysis of Uncertain Data: Tools for Representation and Processing”, IEEE 2008.

[3] Mohamed Anis Bach Tobji, Boutheina Ben Yaghlane, and Khaled Mellouli, “A New Algorithm for Mining Frequent Itemsets from Evidential Databases”, Proceedings of IPMU'08, pp. 1535{1542.

[4] Biao Qin, Yuni Xia, Sunil Prabhakar and Yicheng Tu, “A Rule-Based Classification Algorithm for Uncertain Data”, IEEE 2009 International Conference on Data Engineering, pp- 1633-1640.

[5] Thomas Bernecker, Hans-Peter Kriegel, Matthias Renz, Florian Verhein, Andreas Zuefle, “Probabilistic Frequent Itemset Mining in Uncertain Databases”, 15th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining (KDD'09), Paris, France, 2009.

[6] Gregory Buehrer, Srinivasan Parthasarathy, and Amol Ghoting. Out-ofcore frequent pattern mining on a commodity pc. In KDD ’06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, New York, NY, USA, 2006, pp 86–95.

[7] Toon Calders. Deducing bounds on the frequency of itemsets. In EDBT Workshop DTDM Database Techniques in Data Mining, 2002.

[8] DPVG06] Nele Dexters, Paul W. Purdom, and Dirk Van Gucht. A probability analysis for candidate-based frequent itemset algorithms. In SAC ’06: Proceedings of the 2006 ACM symposium on Applied computing, New York, NY, USA, 2006. ACM, pp541–545.

[9] Edward Fredkin. Trie memory. Commun. ACM, 3(9):490–499, 1960.

[10] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cacheconscious frequent pattern mining on a modern processor. In Klemens B¨ohm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Ake Larson, and Beng Chin Ooi, editors, VLDB ACM, 2005, pp 577–588.

[11] Mohammed J. Zaki. Scalable algorithms for association mining. IEEE Trans. on Knowl. and Data Eng., 12(3):pp 372–390, 2000.

[12] C. C. Aggarwal, Y. Li, J. Wang, and J. Wang. Frequent pattern mining with uncertain data. In Proc. of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009.

[13] P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. "Trio: A system for data, uncertainty, and lineage". In Proc. Int. Conf. on Very Large Databases, 2006.

[14] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proc. ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, MN, 1994.