ISSN 2319-2720

Volume 1, No.1, July – August 2012
International Journal of Computing, Communications and Networking
Available Online at http://warse.org/pdfs/ijccn04112012.pdf

Apriori Based Novel Frequent Itemset Mining Mechanism

Sanjaydeep Singh Lodhi[1]
Department of Computer Application (Software Systems), S.A.T.I , Vidisha, (M.P), India
sanjayeng.mt@rediffmail.com

Sandhya Rawat
Department of C.S.E, Truba Engineering College,Bhopal, M.P, India
sandhya.turba@gmail.com

Premnarayan Arya
Asst. Prof. Dept. of CA (Software Systems), S.A.T.I, Vidisha, (M.P), India
premnarayan.arya@rediffmail.com

ABSTRACT

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 computer science and data mining, Apriori is a classic algorithm for learning association rules. Apriori is designed to operate on databases containing transactions. Other algorithms are designed for finding association rules in data having no transactions, or having no timestamps. Frequent itemset mining is not a real-time system, so the precise speed of execution is not especially important. What is important is the ability to process datasets that are otherwise simply too large from which to extract meaningful patterns. In this paper proposed an efficient frequent itemset mining using Priori algorithm which naturally lends it to sorting because, without any loss in efficiency, every step of it can be designed to either create or preserve sort order. This allows us to improve every step of the original algorithm. 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.

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

1. INTRODUCTION

Frequent patterns are itemsets, subsequences, or substructures that appear in a data set with frequency no less than a user-specified threshold. For example, a set of items, such as milk and bread that appear frequently together in a transaction data set is a frequent itemset. A substructure can refer to different structural forms, such as subgraphs, subtrees, or sublattices, which may be combined with itemsets or subsequences. If a substructure occurs frequently in a graph database, it is called a (frequent) structural pattern. Finding frequent patterns plays an essential role inmining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data indexing, classification, clustering, and other data mining tasks as well. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research. Frequent pattern mining was first proposed by Agrawal et al. for market basket analysis in the form of association rule mining [1] and [3] and [7].

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 thesis, we take the classic algorithm for the problem, A Priori, 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 this paper proposed an efficient frequent itemset mining using Priori algorithm which naturally lends it to sorting because, without any loss in efficiency, every step of it can be designed to either create or preserve sort order. This allows us to improve every step of the original algorithm.

2. BACKGROUND TECHNIQUES

2.1 Sequence Mining

Sequence mining is concerned with finding statistically relevant patterns between data examples where the values are delivered in a sequence. It is usually presumed that the values are discrete, and thus Time series mining is closely related, but usually considered a different activity. Sequence mining is a special case of structured data mining. There are two different kinds of sequence mining: string mining and itemset mining. String mining is widely used in biology, to examine gene and protein sequences, and is primarily concerned with sequences with a single member at each position. There exist a variety of prominent algorithms to perform alignment of a query sequence with those existing in databases. The kind of alignment could either involve matching a query with one subject e.g. BLAST or matching multiple query sets with each other. Itemset mining is used more often in marketing and CRM applications, and is concerned with multiple-symbols at each position. Itemset mining is also a popular approach to text mining. There are several key problems within this field. These include building efficient databases and indexes for sequence information, extracting the frequently occurring patterns, comparing sequences for similarity, and recovering missing sequence members. Two common techniques that are applied to sequence databases for frequent itemset mining are the influential apriori algorithm and the more-recent FP-Growth technique. However, there is nothing in these techniques that restricts them to sequences [6] and [7].

2.2 Maximal Frequent Itemset

The concept of MFI (Maximal Frequent Item sets) and the Max-Miner algorithm for mining only MFI. Max-Miner looks only the MFIs, because of that, the search space can be reduced. Max-Miner uses a bottom up traversal of a database. Max-Miner employs a purely breadth-first search of the set-enumeration tree in order to limit the number of passes made over the data.

The key to an efficient set-enumeration search is the pruning strategies that are applied to remove entire branches from consideration. Without pruning, a set-enumeration tree search for frequent item sets will consider every item set over the set of all items. Max-Miner uses pruning based on subset infrequency, as does Apriori, but it also uses pruning based on superset frequency. In Max-Miner each node represent in the set enumeration tree let us call it a candidate group. A candidate group g consists of two item sets. The first, called the head and denoted h (g), represents the item set enumerated by the node. The second item set, called the tail and denoted t (g), is an ordered set and contains all items not in h (g) that can potentially appear in any sub-node. The ordering of tail items reflect how the sub-nodes are to be expanded. In the case of a static lexical ordering without pruning, the tail of any candidate group is trivially the set of all items and the greatest item in the head according to the item ordering [3] and [5].

- ***Apriori algorithm***

Apriori is the best-known algorithm to mine association rules. It uses a breadth-first search strategy to counting the support of itemsets and uses a candidate generation function which exploits the downward closure property of support.

- ***Eclat algorithm***

Eclat is a depth-first search algorithm using set intersection.

- ***FP-growth algorithm***

FP-growth (frequent pattern growth uses an extended prefix-tree (FP-tree) structure to store the database in a compressed form. FP-growth adopts a divide-and-conquer approach to decompose both the mining tasks and the databases. It uses a pattern fragment growth method to avoid the costly process of candidate generation and testing used by Apriori.

- ***GUHA procedure ASSOC***

GUHA is a general method for exploratory data analysis that has theoretical foundations in observational calculi.[18] The ASSOC procedure is a GUHA method which mines for generalized association rules using fast bitstrings operations. The association rules mined by this method are more general than those output by apriori, for example "items" can be connected both with conjunction and disjunctions and the relation between antecedent and consequent of the rule is not restricted to setting minimum support and confidence as in apriori: an arbitrary combination of supported interest measures can be used.

- ***OPUS search***

OPUS is an efficient algorithm for rule discovery that, in contrast to most alternatives, does not require either monotone or anti-monotone constraints such as minimum support. Initially used to find rules for a fixed consequent it has subsequently been extended to find rules with any item as a consequent. OPUS search is the core technology in the popular Magnum Opus association discovery system [4] and [9].

2.3 Uncertain Data Model

The uncertain data model applied in this paper is based on the possible worlds semantic with existential uncertain items.

Definition 1: An uncertain item is an item x ∈ I whose presence in a transaction t ∈ T is defined by an existential probability P(x ∈ t) ∈ (0; 1). A certain item is an item where P(x ∈ t) ∈ f0; 1g. I is the set of all possible items.

Definition 2: An uncertain transaction t is a transaction that contains uncertain items. A transaction database T containing uncertain transactions are called an uncertain transaction database. An uncertain transaction t is represented in an uncertain transaction database by the items x ∈ I associated with an existential probability value P(x ∈ t) 2 (0; 1].

Example uncertain transaction databases are depicted. To interpret an uncertain transaction database we apply the possible world model. An uncertain transaction database generates possible worlds, where each world is defined by a fixed set of (certain) transactions. A possible world is instantiated by generating each transaction ti ∈ T according to the occurrence probabilities P(x ∈ ti). Consequently, each probability 0 < P(x ∈ ti) < 1 derives two possible worlds per transaction: One possible world in which x exists in ti, and one possible world where x does not exist in ti. Thus, the number of possible worlds of a database increases exponentially in both the number of transactions and the number of uncertain items contained in it [3] and [6].

2.4 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 to 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 [10] and [11].

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 Probabilistic Frequent Itemset Mining (PFIM) problem is to find itemsets in an uncertain transaction database that are (highly) likely to be frequent. This problem has two components; efficiently computing the support probability distribution and frequentness probability, and efficiently mining all probabilistic frequent itemsets.

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 [8] and [12].

3. PROPOSED SCHEME ANALYSIS

The publication of A Priori, many subsequent ideas have been proposed. However, the majority of these interest us very little because they do not address the real trouble of frequent itemset mining: scalability. There are lots of cute ideas that use various novel data structures or some tricks to try to reduce the scope of the problem, but if they merely improve the execution time on a dataset that already fits in memory, their value is questionable. Frequent itemset mining is not a real-time system, so the precise speed of execution is not especially important. What is important is the ability to process datasets that are otherwise simply too large from which to extract meaningful patterns. As such, we focus our discussion on those proposals that are designed to address the issue of scalability.

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.

3.1 Generating Candidates

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 $f_i = f_{i+1}$, . . . , $f_{i+m-2} = f_{i+m-1}$ 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.

3.2 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 $0 \leq i \leq k$ 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 testset.

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.

3.3 Support Counting

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 $t = \{t_0, \ldots, t_{w-1}\}$ produces $\binom{w}{k}$ possibilities. Even after pruning infrequent items with a support threshold of 10%, w still ranges so high.

3.4 On Locality and Data Independence

It is fair to assume that any efficient and complete solution to the frequent itemset mining problem on a general, very large dataset is going to require data structures that do not fit entirely in memory. Recent work on FP-Growth accepts this inevitability for very large datasets and focusses on restructuring the trie and reordering the input such that it anticipates relying heavily on a virtual memory based solution. In particular, they aim to reuse a block of data so much as possible before swapping it out again. 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 utilisation 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.

3.5 Proposed Apriori Algorithm

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. 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.

Apriori uses breadth-first search and a tree structure to count candidate item sets efficiently. It generates candidate item sets of length k from item sets of length k-1. Then it prunes the candidates which have an infrequent sub pattern. According to the downward closure lemma, the candidate set contains all frequent k-length item sets. After that, it scans the transaction database to determine frequent item sets among the candidates.

Apriori, while historically significant, suffers from a number of inefficiencies or trade-offs, which have spawned other algorithms. Candidate generation generates large numbers of subsets (the algorithm attempts to load up the candidate set with as many as possible before each scan). Bottom-up subset exploration (essentially a breadth-first traversal of the subset lattice) finds any maximal subset S only after all $2^{|s|}$ -1 of its proper subsets.

The following is a formal statement of the problem: Let $\tau=\{i1, i2, i3……..\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq \tau$. Associated with each transaction is a unique identifier, called its TID. We say that a transaction T contains X, a set of some items in τ, if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subset \tau$, $Y \subset \tau$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set D with confidence $\mathbb{C}$ if c% of transactions in D that contain X also contain Y. The rule $X \Rightarrow Y$ has support $\mathbb{S}$ in the transaction set D if S% of transactions in D contain $X \cup Y$. Given a set of transactions D, the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support (called minsup) and minimum confidence (called minconf) respectively.

The problem is usually decomposed into two sub problems. One is to find those itemsets whose occurrences exceed a predefined threshold in the database; those itemsets are called frequent or large itemsets. The second problem is to generate association rules from those large itemsets with the constraints of minimal confidence. Suppose one of the large itemsets is Lk, Lk = {I1, I2, … , Ik}, association rules with this itemsets are generated in the following way: the first rule is {I1, I2, … , Ik-1}⇒ {Ik}, by checking the confidence this rule can be determined as interesting or not. Then other rule are generated

by deleting the last items in the antecedent and inserting it to the consequent, further the confidences of the new rules are checked to determine the interestingness of them. Those processes iterated until the antecedent becomes empty. Since the second sub-problem is quite straight forward, most of the researches focus on the first sub-problem. The Apriori algorithm finds the frequent sets L In Database D.

Let X, Y ⊆ I be any two itemsets. Observe that if X ⊆ Y , then sup(X) ≥ sup(Y), which leads to the following two corollaries:

- If X is frequent, then any subset Y ⊆ X is also frequent.
- If X is not frequent, then any superset Y ⊇ X cannot be frequent.

Based on the above observations, we can significantly improve the itemset mining algorithm by reducing the number of candidates we generate, by limiting the candidates to be only those that will potentially be frequent. First we can stop generating supersets of a candidate once we determine that it is infrequent, since no superset of an infrequent itemset can be frequent. Second, we can avoid any candidate that has an infrequent subset. These two observations can result in significant pruning of the search space.

→Find frequent set L_{k-1}.

→Join Step.

→C_k is generated by joining L_{k-1} with itself

→Prune Step.

→Any (k-1) -itemset that is not frequent cannot be a subset of a frequent k -itemset, hence should be removed.

where

→(C_k: Candidate itemset of size k)

→(L_k: frequent itemset of size k)

The changes that have come out of this sorting are far-reaching and have impacted every phase of the algorithm.

Algorithm: The revised Vertically-Sorted A Priori algorithm

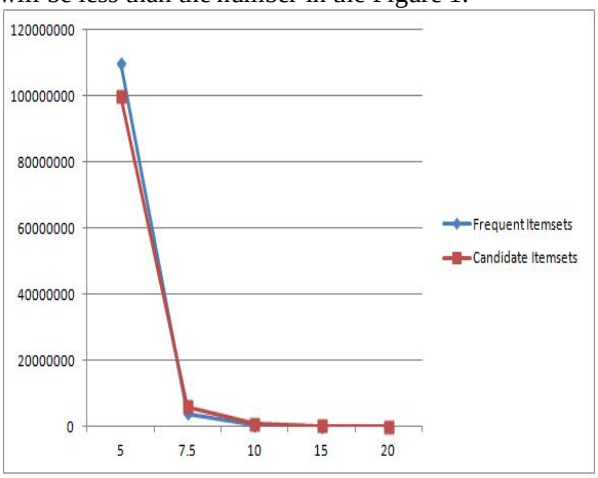
__INPUT:__ A dataset D and a support threshold s
__OUTPUT:__ All sets that appear in at least s transactions of D F is set of frequent itemsets
C is set of candidates
C ← U
Scan database to count support of each item in C
Add frequent items to F
Sort F least-frequent-first (LFF) by support (using quicksort)
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

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. Apriori algorithm is an influential algorithm for mining frequent itemsets for Boolean association rules.

4. EXPERIMENTAL RESULTS

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. 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.

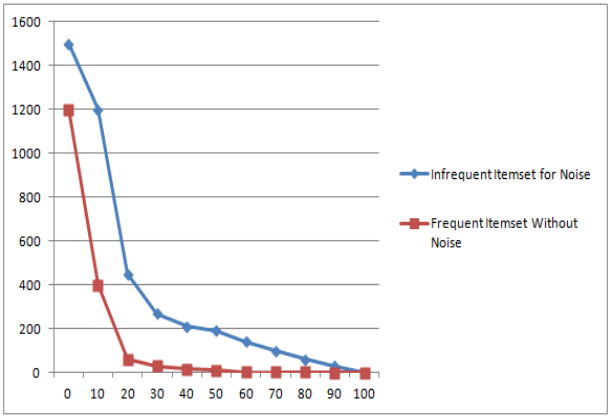
We test each implementation on webdocs with support thresholds of 21%, 14%, 11%, 7%, and 6%. Reducing the support threshold in this manner increases the size of the problem as observed in Figure 1 and Figure 2.The number of candidate itemsets is implementation-dependent and in general will be less than the number in the Figure 1.



Number of Itemsets vs. Supports Threshod (%)

Figure 1: Number of Itemsets in Web documents at Various Support Thresholds

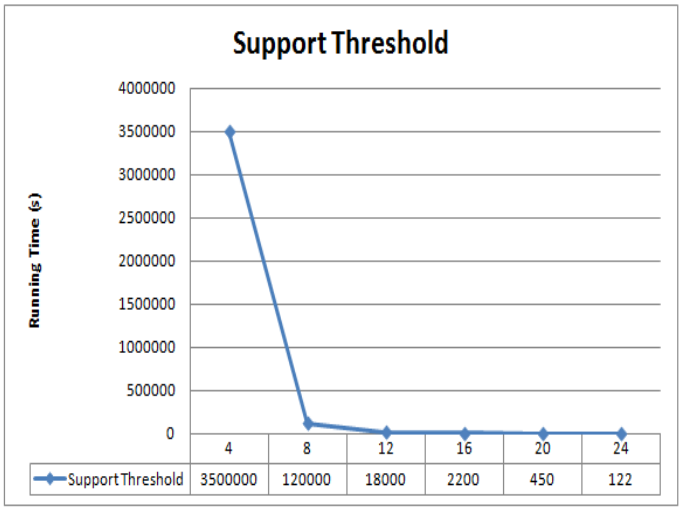
Our implementation uses explicit file-handling instead of relying on virtual memory, the memory requirements are effectively constant. However, those of all the other algorithms grow beyond the limits of memory and consequently cannot initialize. Without the data structures, the programs must obviously abort.



File Size (Mb) vs. Supports (%)

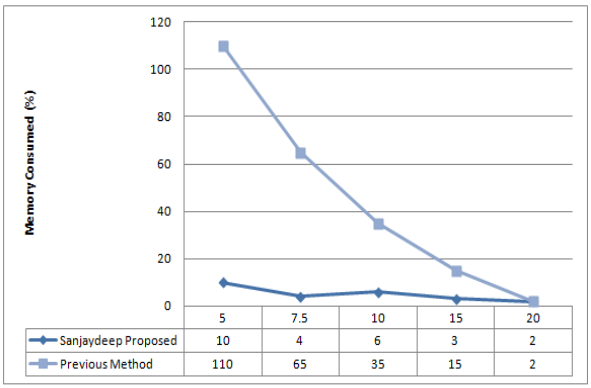
Figure 2: Size of web documents dataset with noise (infrequent 1-itemsets) removed, graphed against the number of frequent itemsets.

It should be noted that in the previous works, their FP-Growth implementation on the same benchmark web-docs dataset as do we and they report impressive running times. 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 2.



Running Time (s) vs. Support Threshold (%)

Figure 3: The Performance of Our Implementations on Webdocs Dataset



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

Figure 4: Memory Usage of Our implementations on webdocs as Measured by final stages of execution

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-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 programmed can successfully mine support thresholds that are impossibly low for the other implementations.

5. CONCLUSION AND FUTURE WORKS

Advantage of apriori is its easy implementation. Association rule mining has a wide range of applicability in many areas. By introducing a vertical sort at the onset of the classic A Priori algorithm, significant improvements can be made. Besides simply having better localized data storage, the candidate generation can be done more efficiently and an indexing structure can be built on the candidates at the same time. Candidates can be compressed to improve comparison times as well as data structure size, and support counting is thus speeded up. The cumulative effect of these improvements is observable in the implementation that we created. Through these experiments, 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.

REFERENCES

1. T. Bernecker, H.-P. Kriegel, M. Renz, F. Verhein, and A. Züfle. **Probabilistic frequent itemset mining in uncertain databases,** Proc. 15th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining, Paris, France, 2009.

2. C. K. Chui and B. Kao. **A decremental approach for mining frequent itemsets from uncertain data,** The 12th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), pp. 64–75, 2008.

3. C. K. Chui, B. Kao, and E. Hung. **Mining frequent itemsets from uncertain data**, 11th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, PAKDD 2007, Nanjing, China, pp. 47–58, 2007.

4. Toon Calders, Calin Garboni and Bart Goethals. **Approximation of Frequentness Probability of Itemsets in Uncertain Data**, IEEE International Conference on Data Mining, pp-749-754, 2010.

5. Bin Fu, Eugene Fink and Jaime G. Carbonell. **Analysis of Uncertain Data: Tools for Representation and Processing**, IEEE 2008.

6. N. Dalvi and D. Suciu. **Efficient query evaluation on probabilistic databases**. The VLDB Journal, 16(4):523–544, 2007.

7. C. K.-S. Leung, C. L. Carmichael, and B. Hao. **Efficient mining of frequent patterns from uncertain data**, ICDMW '07: Proceedings of the Seventh IEEE International Conference on Data Mining Workshops, pp. 489–494, 2007.

8. K. Leung, M. Mateo, and D. Brajczuk. **A tree-based approach for frequent pattern mining from uncertain data**, Advances in Knowledge Discovery and Data Mining, 2008.

9. C. C. Agarwal, Y. Li, J. Wang, and J. Wang. **Frequent pattern mining with uncertain data**, Proc. of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009.

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

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

12. L. Antova, T. Jansen, C. Koch, and D. Olteanu. **Fast and Simple Relational Processing of Uncertain Data**. Proc. 24th Int. Conf. on Data Engineering, Cancún, México, 2008.