Efficient Frequent Item set Discovery Technique in Uncertain Data

Deepak Chopra, Dilip Vishwakarma

Abstract—frequent itemset mining, the task of finding sets of items that frequently occur together in a dataset, has been at the core of the field of data mining for the past sixteen years. In that time, the size of datasets has grown much faster than has the ability of existing algorithms to handle those datasets. Consequently, improvements are needed. 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

Keywords- Uncertain Databases, Frequent Itemset Mining, Probabilistic Frequent Itemsets.

I. INTRODUCTION

The world around us is full of information and contemporary computer systems are allowing us to gather and store that information at an astounding rate. However, our ability to process that information lags far beyond our abilities as gatherers. Most of the truly amazing problems of our time are rooted in extracting the meaningful patterns from datasets so large as to have previously been unfathomable. The human genome has been sequenced but it is still uncertain which parts of it cause us to express particular phenotypes. Hundreds of weather stations around the world have been collecting information about local climate for decades, but it is still difficult to predict the effects of human activities. The universe beyond our stratosphere and the Internet within it are mysterious places, although we have terabytes of data collected from each. This paper is about data mining: the process of extracting the meaningful information from these massive datasets. Even quite defining what is a meaningful relationship among data is non-trivial; but, that said, determining sets of items that co-occur frequently throughout the data is a very good start. This task, frequent itemset mining, is a problem that was suggested sixteen years ago and is still at the heart of the field [1] and [2].

In particular, we have started with the classic algorithm for this problem and introduced a conceptually simple idea of sorting the consequences of which have permitted us to outperform all of the available state-of-the-art implementations. The A Priori algorithm naturally lends itself to sorting because, without any loss in efficiency, every step of it can be designed to either create or preserve sort order. We have exploited this by introducing a sort at the beginning and using it quite thoroughly throughout. This allows us to improve every step of the original algorithm.

II. BACKGROUND

Before explaining the implications of our sorting, let us first review the problem definition and the previous attempts at addressing the problem especially including the task of frequent itemset mining was first introduced. Informally speaking, the objective of it is to detect those items in a dataset that commonly co-occur, preferably indicating with what frequency. To achieve this, one fixes a threshold, s, and then strives to output all those sets of items that co-occur at least s times. Consider the rows of Table below. If one sets the threshold to be s = 2, then table: Example of a dataset in which {a, c} is frequent, designed to illustrate the frequent itemset mining problem

Transaction 0→	a	b	c		
Transaction 1→	a	d			
Transaction 2→	a	c	d	e	f

the sets {{}, {a}, {c}, {a, c}} are frequent because the sets can be found in at least two rows of the table. To make this more precise, we consider a universe, U (which is {a, b, c, d, e, f} in Table). Then, a dataset, D, is defined to be a multiset of transactions and a transaction, t, is defined to be a subset of U.1 (In the example, Transaction 1 = {a, d} is one such transaction and Transaction 0, Transaction 1, and Transaction 2 make up the dataset.) An itemset is likewise defined to be a subset of U. The support of an itemset i is supp (i) = $|\{t \in D : i \subseteq t\}|$

With these definitions, the objective of frequent itemset mining is to determine, given a dataset D and a fixed support threshold, $0 < s \leq |D|$, the set of frequent itemsets: $\{i : supp(i) \geq s\}$.

These frequent itemsets potentially imply new knowledge about the dataset. The task, although simple to describe, is quite difficult for two primary reasons: |D| is typically massive and the set of possible itemsets, |P (U)|, is exponential in size, so the problem search space is likewise exponential. In fact, given a fixed size k, even determining if there exists a set of k items that co-occur in the dataset s times is difficult: it was demonstrated to be NP-complete. Here, we are trying to discover all frequent sets, regardless of size, which is clearly at least as difficult (otherwise we could use the output to determine if a frequent set of size k exists). So, all algorithms for this problem need to emphasis an effective search space pruning strategy or other heuristics to address the NP-completeness of the problem.

Since 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.

Manuscript received on August 25, 2012

Deepak Chopra, Dept. of Computer Application SATI, Vidisha (M.P)
Dilip Vishwakarma, Dept. of Computer Application SATI, Vidisha (M.P) .

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 [2] and [5].

A. Tries:

The first of these advances on which we focus is the idea, as introduced of storing candidates in a trie. A trie (alternatively known as a prefix tree) is a data structure developed which takes advantage of redundancies in the keys that are placed in the tree.

This approach also has the potential to break down on large datasets if the data structure no longer fits in main memory. The depth of the trie is equal to the length of those candidates. To fit all nodes into main memory requires those candidates to overlap quite substantially. When they do not, the effect of the trie's heavily pointer-based makeup is very poor localization and cache utilization. Consequently, traversing it causes one to thrash on disk and the efficiency of the structure is quickly consumed by I/O costs [1] and [3].

B. Maximal Pruning:

As such, it became apparent that to make this trie idea scalable, one would need to reduce the number of candidates created. Precisely this was achieved where demonstrates the sub-optimality of the A Priori Principle and shows that a more rigorous study of the frequent itemsets can produce pruning that is superior to that based strictly on the A Priori Principle. However, perhaps because it is more costly or perhaps rather just because of the timing of his publication, the idea has not really taken off. So, it is still generally accepted that the A Priori algorithm produces quite excessively many candidates and the algorithm has mostly fallen out of the forefront of literature in favour for FPGrowth [3] and [6].

C. FPGrowth:

In recent, Han et al. introduce a quite novel algorithm to solve the frequent itemset mining problem. They adapt the idea of a trie to the set of transactions rather than candidates. In so doing, they effectively compress the dataset D with the hope that it will fit entirely in main memory. Each transaction is inserted into the trie in its most-frequent-first order and at each node of the trie is stored a support counter. When a new transaction t is inserted, a path of size |t| is traced; the count at each node along this path is incremented. Thus, inserting the transaction involves updating |t| support counts. Additionally, a linked list is maintained between all nodes sharing the same label. In this way, one can quickly find all paths that involve the same item. Next, the trie is mined recursively to extract the frequent itemsets. By following the linked-list of nodes labeled by the least-frequent item, one retrieves all paths involving that item. Then a new conditional prefix tree can be built by copying and then modifying the original tree. All paths whose leaves are not labeled with the least-frequent item are removed, this least-frequent item is itself removed, duplicate paths are merged, and the trie is resorted based on the new conditional frequencies. This creates a trie with the same structure as the original tree, but conditioned on the presence of the least frequent item. So, the procedure can be repeatedly recursively from here until the trie consists of nothing but a root node denoting the empty set. This yields all frequent itemsets involving the least-frequent item. The procedure is then repeated for the second-least-frequent item, third-least frequent item, and so on to extract from the trie all frequent itemsets.

The data structure is quite cute and appears to eliminate the construction of candidates entirely. Indeed, experimental results have demonstrated consistently that it significantly outperforms A Priori. However, the story changes when the dataset is quite large because it suffers the same consequences as did the trie of candidates.

Even building the trie becomes extremely costly, to the point that it is remarked that the dominant percentage of execution time is that of constructing the trie. Consequently, on truly large datasets, the FPGrowth algorithm fails even to initialize.

However, when first introduced, it was remarked that the algorithm scales quite elegantly. Indeed, if one has already constructed a trie, then the cost of mining it is roughly the same independent of the support threshold (except that the recursion produces more intermediate trees). However, FPGrowth has a preprocessing step that prunes out all infrequent 1-itemsets prior to building the trie. Consequently, it does not scale quite in the same way as described in literature because as the support threshold is dropped, the number of items pruned from the dataset is decreased and each of these newly unpruned items needs appear in the trie. So the trie needs be reconstructed and the size of it inflates. By what factor is dependent on the distribution of the dataset and the amount by which the support threshold is reduced. But since the algorithm has performed so admirably when it fits in main memory, it has been adopted for widespread study.

The result is that the algorithm has become highly optimised, with recent advances that include 64-bit processing and reconstructions of the data structure to improve its locality on disk; cache consciousness; and auxilliary data structures to speed-up the bottleneck of the algorithm. As such, FPGrowth runs quite well on some benchmarks datasets, but the potential for improvement is likely somewhat limited. Also, the heavy reliance on the trie's underlying pointers limits how efficient the I/O can become. Furthermore, despite the claim that FPGrowth does not produce any candidates, Goethals demonstrates that it can, in fact, be considered a candidate based algorithm and later show that the probability of any particular candidate being generated is actually higher in FPGrowth than in the classical A Priori algorithm.

Another general problem with the FPGrowth algorithm is that it lacks the incremental behaviour of A Priori, something that builds fault tolerance into the algorithm. Should A Priori crash after producing, say, its frequent 5-itemsets, the algorithm can be easily restarted from that point by beginning with the construction of candidate 6-itemsets, rather than starting from the beginning. However, because FPGrowth operates by means of recursion, there are very few points at which the program can save state in anticipation of failure.

Should one wish to begin analysis of the frequent itemsets as they are produced, it would be much more difficult with the FPGrowth algorithm because the preliminary results are all focussed on just the few particular items that happen to be least frequent in the

dataset. Contrasted with the preliminary results of A Priori, frequent itemsets up to a particular size but involving all items, this offers one little analytical power until the algorithm has entirely completed.

Consequently, despite its profound success on smaller benchmark datasets, we call into question the scalability and use on larger datasets of the FPGrowth algorithm [4] and [7].

Attempts at Scaling A Priori introduced a variant that partitions the dataset into components that can be mined within main memory. The idea is that if one partitions the dataset into m parts and an itemset appears in p% of all the transactions, then it must appear in at least (p/m)% of the transactions of at least one of the partitions. So, mining each partition of the dataset with a threshold of p/m will produce all the frequent itemsets. However, this approach incurs the cost of falsely proclaiming some infrequent itemsets as frequent.

Savasere et al. resolve this by, as a post-processing step, verifying all the frequent itemsets that they have produced. However, Buehrer et al. did a case study that demonstrated the number of these falsely proclaimed frequent itemsets grows exponentially as the support threshold is decreased. Consequently, for large datasets this is not an effective approach [8].

Another widely adopted approach is to mine a subset of the frequent itemsets from which the entire set can be derived. The most notable of these subsets is the set of closed frequent itemsets. However, no implementation has demonstrated a scope-reduced approach to be especially effective on really large datasets. Therefore, we retain the original problem definition.

The majority of algorithms and implementations that do not use the above ideas including the demonstrably most efficient implementations yet developed attempt to scale by extending their data structures into virtual memory. However, there are drawbacks to this. In executing the implementations that rely on this strategy one sees that the processor remains largely idle as it waits for the data structure to be swapped in and out of main memory. As such, it does not matter much how efficient the algorithm is because the execution time is dominated by the cost of this swapping. In addition, the virtual memory strategy is not very robust in the scenario that there is another (user or operating system) process running because they then need compete for memory resources. As the competing processes require more resources, less is left available for the frequent itemset mining implementation and its performance is further degraded [9] and [10] and [11].

Thus, we instead use explicit file handling to manage our memory resources in our adaption of the A Priori algorithm.

III. PROPOSED TECHNIQUES

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:

i. Generating candidates can be done very efficiently
ii. Indices on lists of candidates can be efficiently generated at the same time as are the candidates
iii. Groups of similar candidates can be compressed together and counted simultaneously
iv. Candidates can be compared to transactions in linear time
v. Better locality of data and cache-consciousness is achieved

In addition to that, 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.

A. 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 focusses 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.

B. Efficiently 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 $f_i = f_{i+1}, \ldots, f_{i+m-2} = f_{i+m-1}$ 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.

C. 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} items of cs, we produce all possible suffices in sorted order. These, each appended to the prefix, form the $\binom{c_w - k + 1}{2}$ candidates in c.

D. Indexing:

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.

E. On the Precondition of Sorting:

Most of our method is dependent on maintaining the precondition that lists of frequent itemsets and lists of candidates remain sorted, both vertically and horizontally. 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.

F. 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, $f_k = c \setminus \{c_k\}$ 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.

G. 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.

H. Index-Based Support Counting:

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 $\{t_i, t_{i+1}, \ldots, t_{w-1}\}$ to each of those candidates.

I. Counting with Compressed Candidates:

This affords appreciable performance gains. All the candidates compressed into a super-candidate $c = (c_w, c_s)$ share their first $k-1$ elements. So, for a transaction t, if the first $k-1$ items of cs are not strictly a subset of t, then we can immediately jump over $\binom{c_w - k + 1}{2}$ candidates.

None could possibly be contained in t. Suppose instead that the first $k-1$ items of c_s are strictly a subset of a transaction t. How do we increment the support counts of exactly those candidates in c which are contained in t.

J. 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.

K. Full View of Vertically-Sorted A Priori:

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

IV. RESULTS

To test the ideas put forth here, we created an implementation of making frequent itemset in uncertain data. What is interesting in this study is really only the performance on large datasets because the size of the dataset is what makes this an interesting problem. The 2GB of web documents data fits nicely into this category, being the largest dataset commonly used throughout publications on this problem. All other benchmark datasets are quite a lot smaller and not relevant here. 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 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 trie 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.

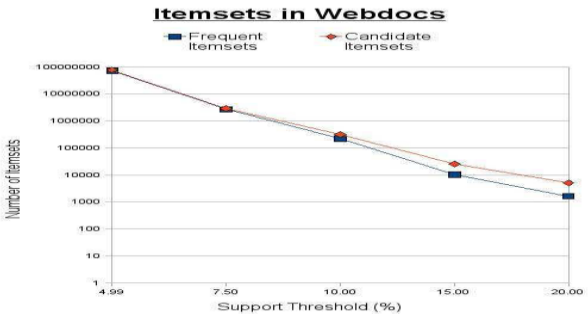

Figure 1: Number of Itemsets in Web documents at Various Support Thresholds

We test each implementation on webdocs with support thresholds of 22%, 16%, 11%, 8%, 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.

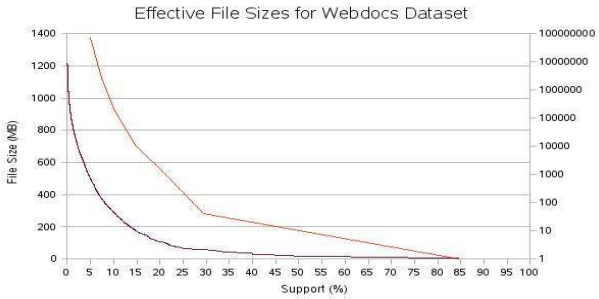

Figure 2: Size of web documents dataset with noise (infrequent 1-itemsets) removed, graphed against the number of frequent itemsets.

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.

V.CONCLUSION AND FUTURE WORKS

Frequent itemset mining is an important problem within the field of data mining, but in previous years of algorithmic development has yet to produce an implementation that can mine sufficiently low support thresholds on even a modest-sized the gigabytes of data in many real-world applications. 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.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 research 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 cost to performance, 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 inhibitively slow. But, in fact, this work reestablishes it as the frontier algorithm.

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.