Scalable APRIORI-Based Frequent Pattern Discovery

Sean Chester, Ian Sandler, Alex Thomo
University of Victoria
Victoria, Canada
schester@uvic.ca, iansandl@uvic.ca, thomo@cs.uvic.ca

***Abstract*—Frequent pattern discovery, 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 paper we take the classic algorithm for the problem, *A Priori*, and by adding a vertical sort drastically improve its performance characteristics when processing very large data sets. We use the benchmark large dataset webdocs from the FIMI 2004 conference to contrast our performance against several state-of-the-art implementations and demonstrate both equal efficiency with lower memory usage at all support thresholds and also the ability to mine support thresholds as yet unattempted in literature. We also indicate how this work can be extended to achieve yet more impressive results.

***Keywords*-frequent pattern discovery; apriori; data mining;**

I. Introduction

The frequent pattern discovery problem of [1] is by now well-known within the data mining community. 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 I. If one sets the threshold to be $s = 2$, then 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, $\mathcal{U}$ (which is $\{a, b, c, d, e, f\}$ in Table I). Then, a *dataset*, $\mathcal{D}$, is defined to be a multiset of transactions and a *transaction*, t, is defined to be a subset of $\mathcal{U}$.[1] An *itemset* is likewise defined to be a subset of $\mathcal{U}$. The *support* of an itemset i is

$$supp(i) = |\{t \in \mathcal{D} : i \subseteq t\}|.$$

With these definitions, the objective of frequent itemset mining is to determine, given a dataset $\mathcal{D}$ and a fixed *support threshold*, $0 < s \leq |\mathcal{D}|$, 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

[1]In the original definition of the problem, a transaction is defined to be a 2-tuple, (tid, set), but we do not use the tids in this paper and so drop them from the definition to simplify the ensuing discussion.

Table I
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**	e	f

quite difficult for two primary reasons: $|\mathcal{D}|$ is typically massive and the set of possible itemsets, $|\mathcal{P}(\mathcal{U})|$, 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 in [2] 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 emphasise an effective search space pruning strategy or other heuristics to address the NP-completeness of the problem.

Since the introduction of the frequent itemset mining task, numerous algorithms have been proposed for it. Two stand out in literature because of their positive experimental results. *A Priori*[3] is the oldest well-adopted algorithm, but has fallen out of favour for the newer, more popular, and more complex *FPGrowth* algorithm of [4].

In this paper, we revert back to *A Priori*. With the novel idea of introducing *vertical* sorting to the algorithm, we will be able to contribute several new innovations that allow us to mine support thresholds that are yet unattempted in literature. But before explaining this sorting or those innovations, first we review in Section II the work that has been done prior to now. The new ideas are explained in Section III and we give experimental results in Section IV. Finally, in Section V we show how this work can be extended.

II. Background

A. The A Priori Algorithm

Shortly after the problem was introduced, Agrawal et al. proposed the *A Priori* algorithm[3] to solve it efficiently. The cleverness of their algorithm comes from an aggressive search space pruning strategy. The property that permits this

has been coined the *A Priori Principle* and still forms the basis of many of the algorithms that have been published.

If some set t contains some subset s, then it also contains *all* subsets of s. Considering this on a grander scale, if s is known to occur in, say, p transactions, then all subsets of s occur in at least p transactions, since they must occur in those transactions in which s occurs, even if no others. Stating this alternatively gives the *A Priori Principle*:

$$supp(s_i) \geq supp(s_i \cup s_j),\ for\ all\ sets\ s_i, s_j$$

The algorithm proceeds in a step-wise fashion, considering first all itemsets with one item, then all itemsets with two items, and so on. On the $(k+1)^{th}$ step, three things happen. First, frequent itemsets of size k are merged together to produce candidates of size $k+1$. Second, the *A Priori Principle* is then applied to the candidates to determine which of them are quite obviously infrequent. Finally, the candidates which could not be pruned are compared against $\mathcal{D}$ to explicitly ascertain their support count. The process is then repeated for step $k+1$. The algorithm terminates as soon as all candidates of a particular size are pruned.

It is commonplace to refer to these three happenings as *Candidate Generation*, *Candidate Pruning*, and *Support Counting*, respectively. Because they occur in series, they are often considered in literature independently of each other. For the details of each, refer to [3]. We do provide a detailed description of the candidate generation procedure next because it is particularly relevant to our discussion in Section III-B.

The $(k-1) \times (k-1)$ Candidate Generation Method: How does one construct candidates of a particular size from a set of frequent itemsets? Just taking the union of arbitrary sets is not going to produce new sets with exactly $k+1$ elements. Although there are a number of ways to choose sets to join, only one is used prominently and with much success: the $(k-1) \times (k-1)$ method that we adopt. Consider two itemsets of size k. Their union will contain precisely $k+1$ items exactly when they share $k-1$ items. By imposing a lexicographical sort, one guarantees that each candidate will be generated only once.

B. Recent Advances

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 issue of frequent itemset mining: scalability. 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.

Tries: As demonstrated in [5], the primary bottleneck of the classical *A Priori* algorithm is in incrementing counters for those candidates that are active in a particular transaction. Storing candidates in a trie structure helps immensely because the process of matching a candidate to a transaction simultaneously accomplishes that of loading the appropriate counter because it is stored in the leaf of the trie. But even this approach is not fast enough. When comparing nearly 1.7 million transactions to 30 million candidates as is done on the webdocs data set[6], the cost of everything is significantly magnified. Every node in the trie requires two pointer dereferences and a candidate may require a traversal of as many nodes as items it contains. Having a data structure that permits faster access is invaluable.

This approach breaks down on large datasets once the data structure no longer fits in main memory. The depth of the trie is equal to the length of the candidates. To fit all nodes into main memory requires those candidate to overlap quite substantially. When they do not, the effect of the trie's heavily pointer-based makeup is very poor localisation and cache utilisation. So, traversing the structure causes thrashing on cache and disk. A typical random disk I/O takes about 10 milliseconds; a typical memory access takes a small fraction of a microsecond, a ratio of at least 100,000:1. Consequently the efficiency of this structure is quickly consumed by these costs.

FPGrowth: In [4], 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 $\mathcal{D}$ with the hope that it will fit entirely in main memory.

The data structure appears to eliminate the construction of candidates entirely. Experimental results have demonstrated consistently that it significantly outperforms *A Priori*. However, once the trie no longer fits in memory it suffers exactly the same consequences as in [5]. Even building the trie becomes extremely costly, to the point that in [7] 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 initialise.

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, one must be careful here. *FPGrowth* has a preprocessing step that prunes out all infrequent 1-itemsets prior to building the trie. Consequently, it does not scale as claimed because as the support threshold is lowered, the number of items pruned from the dataset decreases—and each of these newly unpruned items needs appear in the trie. So the trie needs to be reconstructed and it grows. How much it grows is dependent on the distribution of the dataset

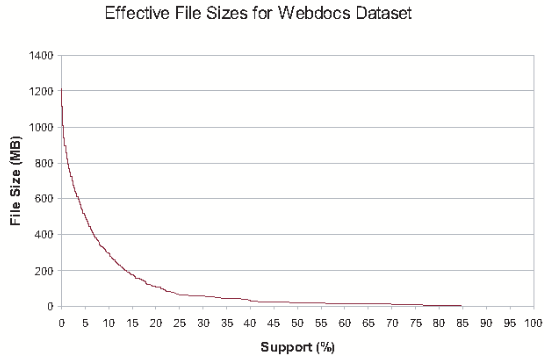

Figure 1. Size of webdocs dataset with noise (infrequent 1-itemsets) removed, graphed under the number of frequent itemsets (output size)

and the amount by which the support threshold is reduced. This growth can be several orders of magnitude for relatively small decreases in support threshold.

To illustrate the significance of this growth, we show in Figure 1 the size of the popular benchmark webdocs dataset[6] after applying the preprocessing step. The size of the trie is much closer related to this "effective file size" than the size of the original dataset. This gives a better representation of the true input size of the dataset for a given support threshold. For contrast, the higher curve graphed on the right axis is the number of frequent itemsets in the output.

Furthermore, despite the claim that *FPGrowth* does not produce any candidates, Goethals demonstrates in [8] that it can, in fact, be considered a candidate-based algorithm and Dexters et al. later show that the probability of any particular candidate being generated is actually higher in *FPGrowth* than in the classical *A Priori* algorithm[9].

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 machine running *A Priori* fail or shut down 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.

Consequently, despite its profound success to date, we choose not to use the *FPGrowth* algorithm on very large datasets with low support thresholds. Instead we have developed a variant of the *A Priori* algorithm that that uses much less memory, preventing the problems with disk thrashing and allowing us to efficiently generate results when other algorithms fail.

III. VERTICALLY SORTED *A Priori*

A. *Overview of Improvements*

We start as in classical *A Priori* by counting the support of every item in the dataset and sort them in decreasing order of frequency. Next, we sort the dataset *horizontally*; that is, we sort the items of each transaction least-frequent-first.[2] We generate the candidate itemsets such that they are also horizontally sorted. But in addition to this, we generate them such that entire candidates are sorted 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

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.

Each of these advantages is detailed more thoroughly in the next sections.

B. *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 useful instructions per second. In the example of the webdocs dataset on a 10% support threshold, comparing each frequent 6-itemset to each other involves 6*(55881*55880)/2 comparisons. Even if each comparison can be done with just two operations, one still requires about 10.5 minutes to generate these candidates prior even to pruning. In comparison, we count the support of these

[2] Strictly speaking, an ordered collection is not a set, so we are slightly abusing the set notation. When we indicate the union operation, for example, we mean the union of the ordered collections such that the result maintains the ordering.

Table II
EXAMPLE SET OF FREQUENT 4-ITEMSETS

f_0	6	5	3	2
f_1	6	5	3	1
f_2	6	5	3	0
f_3	6	5	2	0
f_4	6	5	1	0
f_5	5	4	3	2
f_6	5	4	3	0

candidates in roughly 36 minutes. So, it is worthwhile to devote considerable attention to improving the efficiency of candidate generation, too. Here we explain how.

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. A small example if k were four is given in Table II.

As described in Section II-A, the $(k-1) \times (k-1)$ 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, f_i and f_j, we call them *near-equal* and denote their near-equality by $f_i \doteq f_j$. Then, classically, every frequent itemset f_i is compared to every f_j and the candidate $f_i \cup f_j$ is generated whenever $f_i \doteq f_j$. However, even in our small example, we must verify this relationship for

$$\binom{7}{2} = 7 * 8/2 = 28$$

pairs of frequent k-itemsets. Given the size of datasets that we are interested in mining, this step is too slow because the number of frequent k-itemsets is so large.

However, our method needs only ever compare one frequent itemset, f_i, to the one immediately following it, f_{i+1}. In the example of Table II, we improve from comparing twenty-eight itemsets for near-equality to only comparing seven. The ability to do this is entirely dependent on having the frequent itemsets vertically sorted.

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

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. Consider the given example.

To begin, we set a pointer to the first frequent itemset, $f_0 = \{6, 5, 3, 2\}$. Then we check if $f_0 \doteq f_1$, $f_1 \doteq f_2$, $f_2 \doteq f_3$ and so on until the near-equality is no longer satisfied. This occurs between f_2 and f_3 because they differ on their 3rd items. Let m denote the number of itemsets we determined to be near-equal, 3 in this case. Then, because near-equality is transitive, we can take the union of every possible pair of the $m = 3$ itemsets to produce our candidates. In this case, we create the three candidates $\{\{6, 5, 3, 2, 1\}, \{6, 5, 3, 2, 0\}, \{6, 5, 3, 1, 0\}\}$ and in general $\binom{m}{2}$ candidates will be produced.

Then, to continue, we set the pointer to f_3 and proceed as before. We see that f_3 is not near-equal to f_4, so we have no pairs to merge. The pointer is next set to f_4 for which the same can be said. We then set the pointer to f_5 and verify that $f_5 \doteq f_6$.

Since there are no more frequent itemsets, we pair f_5 and f_6 and the candidate generation is complete. The full set of candidates that we generated is $\{\{6, 5, 3, 2, 1\}, \{6, 5, 3, 2, 0\}, \{6, 5, 3, 1, 0\}, \{5, 4, 3, 2, 0\}\}$.

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. At any rate, we circumvent this as described in the next section.

Candidate compression: 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*.

We illustrate this by reusing the example of Table II on page 4. Of those frequent 4-itemsets, we discover that f_0, f_1, and f_2 are near-equal. From them, $c_0 = \{6, 5, 3, 2, 1\}$, $c_1 = \{6, 5, 3, 2, 0\}$, $c_2 = \{6, 5, 3, 1, 0\}$ would be generated as candidates. But instead consider $c = f_0 \cup f_1 \cup f_2$.

Then, the 2-tuple $(k+m-1, c) = (6, \{6, 5, 3, 2, 1, 0\})$ encodes all the information we need to know about all the candidates generated from f_0, f_1, and f_2. The first $k-1$ items in the set c are common to all $\binom{m}{2}$ candidates. We call this 2-tuple 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. This is covered in detail in section III-D.

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 c_s. 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

Table III
SAMPLE INDEX FOR CANDIDATE 5-ITEMSETS

Item	Offset	NumBytes
6	0	52
5	52	24
4	-1	-1
3	-1	-1
2	-1	-1
1	-1	-1
0	-1	-1

c_s, 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.

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 ith 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.

Consider again the example of Table II. The candidates we generated, when stored sequentially as super candidates, appear as below:

6653210565310554320

The first two super candidates have 6 as their first item and the third, 5. This creates a boundary between the second 0 and the 5 that succeeds it. The purpose of the indexing structure is to keep track of where in the file that boundary is and offer information that is useful for block-reading along this boundary. Table III indicates how the structure would look if each of these numbers consumed four bytes. (We use -1 in an ith position as a sentinel to indicate that no candidates begin with item i.

Note that one could certainly index using the j least frequent items of each candidate, for any fixed $j < k+1$. As j is chosen larger, the index becomes more precise but consumes more memory.

We note here that the idea of building an index on the candidates is not novel. In fact, this is precisely the idea behind the tries of [4], [5], [10], [11]. However, the nature of our indexing structure is very different: it does well in three immediately evident ways. First, it is more likely to fit into memory, because it only requires storing three numbers for each item, rather than entire candidate sets. Second, it partitions nicely along the same boundaries as the candidates are sorted; so, if the structure is too large to fit in memory, it can be easily divided into components that do. Third, it is incredibly quick to build.

C. Candidate Pruning

When *A Priori* was first proposed in [3], 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 [9], 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, $f_k = c \setminus \{c_k\}$ and $f_{k-1} = c \setminus \{c_{k-1}\}$.

If either of these happen 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 independently frequent items of all those in c. We observed that usually either f_0 or f_1 is the best testset. Let us consider why within the classic "beer and diapers" context.

Using an *LFF* sort, we have some predictability about the testsets that we use. We always try to extend a set with the most frequent items, rather than with an arbitrary choice. As such, we do not take some set of independently frequent items, like {eggs, milk}, and append to it just about everything. Instead, we start with more interesting groups, like {beer, diapers}, and extend them with those more frequent items which have occurred in conjunction with the group. This is naturally going to be more successful, because we have already ascertained the presence of the least likely elements. So, rather than going about an expensive candidate pruning procedure in which we examine every possible testset in order to guarantee we find the best one, we instead accept that there are a few (although not especially many) extra candidates and move right along.

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

retain additional candidates, then we do not do very much extra work to count them.

D. Support Counting

It was recognised 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.

Index-Based Support Counting: Here, we exploit the vertical sort of our candidates in conjunction with the index we built when we generated them. To process a transaction $t = \{t_0, \ldots, t_{w-1}\}$, we consider each of the $w - k$ first items in t. For each such item t_i we use the index to retrieve the contiguous block of candidates whose first element is t_i. Then, we compare the suffix of t, that is $\{t_i, t_{i+1}, \ldots, t_{w-1}\}$, to each of those candidates.

Counting with Compressed Candidates: Recall from section III-B that candidates can be compressed. 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 c_s 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 (no more, no fewer)? We illustrate this by example. Let $t = \{6, 5, 4, 3, 2, 0\}$ be the transaction and, as before, $c = (c_w, c_s) = (6, \{6, 5, 3, 2, 1, 0\})$ be the super-candidate and $k + 1 = 5$ be the size of the candidates. We lay out a linear integer array, A, of size

$$\binom{c_w - k + 1}{2} = \binom{3}{2} = 3$$

in which we keep track of each candidate's support count.

Some items of c_s are also in t. Each has an index in c_s and we keep all such indices above $k - 1$. This gives us $c' = \{3, 5\}$ (corresponding to the items 3 and 0). We then subtract these indices from $c_w = 6$, producing $c'' = \{3, 1\}$.

Finally, we increment the support counts for each of the $\binom{|c''|}{2}$ candidates contained in t.

To do so for elements i and j in c'' (with $i > j$), we increment

$$A\left[\binom{c_w - k + 1}{2} - 1 - x\right]$$

where $x = \binom{i}{2} + j - i$.

In our example, the only choices for i and j are $i = 3$ and $j = 1$, so

$$x = \binom{3}{2} + 1 - 3 = 1$$

and we only increment

$$A\left[\binom{3}{2} - 1 - x\right] = A[3 - 1 - 1] = A[1].$$

Reflecting on our super-candidate, it represented the candidates $c_0 = \{6, 5, 3, 2, 1\}$, $c_1 = \{6, 5, 3, 2, 0\}$, $c_2 = \{6, 5, 3, 1, 0\}$. Of these three, only c_1 is contained in t. The only integer we incremented was $A[1]$. Our mapping would increment $A[0]$ for c_0 and $A[2]$ for c_2.

This is how we consistently index our arrays, but certainly any mapping from

$$\{(i, j) : 0 < j < i \le c_w - k + 1\}$$

onto the interval $\left[0, \binom{c_w-k+1}{2}\right)$ if applied consistently will work. In fact, one need not even map to such a tight interval if space is not a concern. We chose our mapping

$$\binom{c_w - k + 1}{2} - 1 - \left(\binom{i}{2} + j - i\right)$$

because it has the nice property that order is maintained.

E. 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 in [7] on *FP-Growth* accepts this inevitability and focuses on restructuring the trie and reordering the input to anticipate heavy reliance 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 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 also performs 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 f_i and f_{i+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 parallelisation and fault tolerance.

IV. Experimental Results

To test our ideas we created an implementation using C.[3] Only the performance on datasets that exceed memory size are interesting because otherwise the problem is trivial and most implementations perform well enough. The 1.5GB of webdocs data[6] 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 webdocs 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 webdocs, so it is a fairer comparison. For these reasons, we used other datasets only for the purpose of verifying the correctness of our output.

On this dataset, we compare the performance of this implementation against a wide selection of the best available implementations of various frequent itemset mining algorithms. Those of [12] and of [13] are state-of-the-art implementations of the *A Priori* algorithm which use a trie structure to store candidates. An alternative algorithm was implemented in [14] and exhibited the best performance on this benchmark dataset at the renowned FIMI conference of 2004.[15] That of [16] is the best available *FPGrowth* implementation. All of these implementations against which we compare are written in C++ by experienced coders. In order to maximally remove uncontrolled variability in the comparisons the choice of programming language is important. We chose C as a balance between programming experience and the similarity of the language to C++. All of the implementations were compiled on the same machine with the same class of gnu compilers (gcc 4.1.2 and g++ 4.1.2) set at the highest level of optimisation.

We test each implementation on webdocs with support thresholds of 20%, 15%, 10%, 7.5%, and 5%. Reducing the support threshold in this manner increases the size of the problem as observed in Figure 1. All tests were run on a Dual-Core Intel Xeon Processor 5140, 2.33 GHz/1333 MHz, 4MB L2 machine.

As can be seen in Figure 2, our implementation performs at least as well as *all* of the aforementioned state-of-the-art implementations on support thresholds below 15%. Furthermore, no other implementation was able to process as low a support threshold as was ours. The implementations of [16] and of [14] were unable to complete within a reasonable period of time at 10%. The trie-based *A Priori* implementations could not compute the frequent itemsets at 5%. On the other hand, our implementation easily finished $k < 9$ at 5% before the tests were concluded.

[3] A repository containing the implementation has been setup as of July 2009 at http://webhome.csc.uvic.ca/∼schester/.

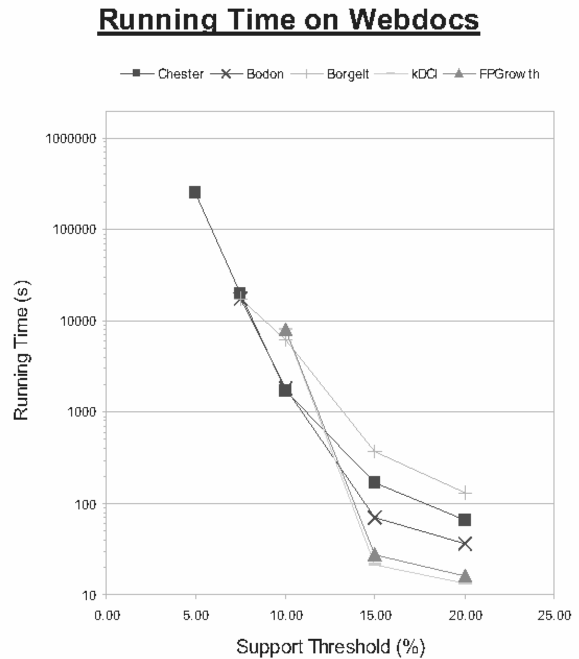

Figure 2. Relative Performance of Implementations on Webdocs Dataset

To explain the difference, Figure 3 displays the memory usage of our implementation and of [12] as measured by the Unix *top* command. As the size of the dataset grows (or, equivalently, the support threshold decreases), so too does the size of the memory structures required. However, because our implementation uses explicit filehandling instead of relying on virtual memory, the memory requirements are effectively constant. Those of all the other algorithms grow beyond the limits of memory and consequently cannot initialise. Without the data structures, the programs must obviously abort.

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

V. Conclusion

Frequent itemset mining is an important problem within the field of data mining. But, sixteen years of algorithmic development has not produced an implementation that can

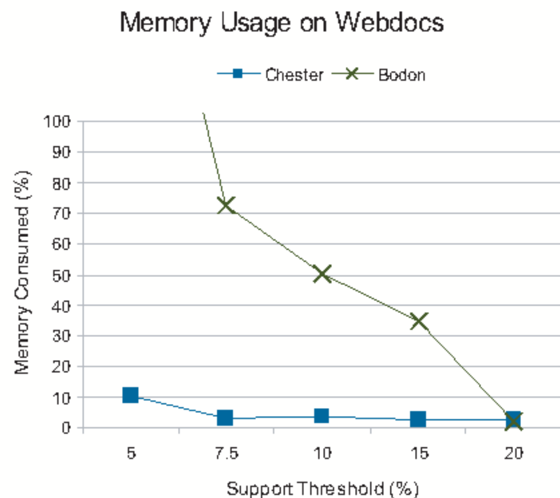

Figure 3. Memory Usage of Bodon and Chester implementations on webdocs as Measured by Unix *top* command during late stages of execution

mine sufficiently low support thresholds on even a modest-sized benchmark dataset—never mind 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 localised 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. The cumulative effect of these improvements is observable in the implementation that we created.

Whereas other algorithms have been heavily optimised, this work opens up many avenues for yet more pronounced improvement. Given their independence, the data structures used can be partitioned and parallelised quite easily with minimal need for inter-process communication. Because our algorithm's memory footprint is so small we have sufficient memory available to fully exploit multi-core architectures without disk thrashing. We expect future results to show a near-linear improvement with an increase in the number of cores. Extending the index to more than one item to improve its precision should also yield significant improvement.

The result of this research 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. 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. This reestablishes it as the frontier algorithm.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*. ACM Press, 1993, pp. 207–216.

[2] P. W. Purdom, D. V. Gucht, and D. P. Groth, "Average-case performance of the apriori algorithm," *SIAM Journal on Computing*, vol. 33, no. 5, pp. 1223–1260, 2004.

[3] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of VLDB*, 1994, pp. 487–499.

[4] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *SIGMOD Conference*. ACM, 2000, pp. 1–12.

[5] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," *SIGMOD Rec.*, vol. 26, no. 2, pp. 255–264, 1997.

[6] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri, "Webdocs: a real-life huge transactional dataset," in *FIMI*, ser. CEUR Workshop Proceedings, vol. 126, 2004.

[7] G. Buehrer, S. Parthasarathy, and A. Ghoting, "Out-of-core 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: ACM, 2006, pp. 86–95.

[8] B. Goethals, "Efficient frequent pattern mining," Ph.D. dissertation, transnationale Universiteit Limburg, 2002.

[9] N. Dexters, P. W. Purdom, and D. Van Gucht, "A probability analysis for candidate-based frequent itemset algorithms," in *SAC '06*. New York, NY, USA: ACM, 2006, pp. 541–545.

[10] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Proc. 5th Int. Conf. Extending Database Technology (EDBT96)*, 1996, pp. 3–17.

[11] C. Borgelt and R. Kruse, "Induction of association rules: Apriori implementation," in *Proceedings of the fifteenth conference on computational statistics*, 2002, pp. 395–400.

[12] F. Bodon, "A fast apriori implementation," in *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, ser. CEUR Workshop Proceedings, vol. 90, Melbourne, Florida, USA, 19. November 2003.

[13] C. Borgelt, "Recursion pruning for the apriori algorithm," in *FIMI*, ser. CEUR Workshop Proceedings, vol. 126, 2004.

[14] C. Lucchese, S. Orlando, and R. Perego, "kdci: on using direct count up to the third iteration," in *FIMI*, ser. CEUR Workshop Proceedings, vol. 126, 2004.

[15] *FIMI '04, Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004*, ser. CEUR Workshop Proceedings, vol. 126, 2004.

[16] G. Grahne and J. Zhu, "Efficiently using prefix-trees in mining frequent itemsets," in *FIMI*, ser. CEUR Workshop Proceedings, vol. 126, 2003.