Scalable APRIORI-Based Frequent Pattern Discovery

by

Sean Chester
B.Sc. University of Victoria 2007

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

Scalable APRIORI-Based Frequent Pattern Discovery

by

Sean Chester
B.Sc., University of Victoria, 2007

Supervisory Committee

Dr. A. Thomo, Supervisor (Department of Computer Science)

Dr. V. Srinivasan, Member (Department of Computer Science)

Dr. M. Serra, Member (Department of Computer Science)

Supervisory Committee

Dr. A. Thomo, Supervisor (Department of Computer Science)

Dr. V. Srinivasan, Member (Department of Computer Science)

Dr. M. Serra, Member (Department of Computer Science)

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. Consequentely, 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 benchmark large dataset, webdocs, from the FIMI 2004 conference 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 unattempted in literature. We also indicate how we believe this work can be extended to achieve yet more impressive results.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Frequent Itemset Mining

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 thesis 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.

In particular, we have started with the classic algorithm for this problem and introduced a conceptually simple idea—sorting—the consequences of which have permitted us to outperform all of the available state-of-the-art implementations.

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.

1.2 Organization

We begin in Chapter 2 by defining the *frequent itemset mining* problem formally and giving background on the progress that has been made on it. Included in that is a full introduction to the algorithm that we upgrade, *A Priori*, and a review of other relevant work that has strived toward solving this problem on large datasets.

Then, in Chapter 3 we detail quite thoroughly how we modify the original algorithm in order to achieve the efficiency that we have. In so doing, we contrast to the ideas of other researchers when appropriate. The chapter is broken into several sections, one for each phase of the algorithm, and an additional section about general improvements.

These changes we implement and in Chapter 4 we detail the results of our experiments on a well-known benchmark test dataset. We compare against four state-of-the-art implementations that were all designed with the same dataset in mind.

Finally, in Chapter 5 we offer our conclusions and indicate in what manner we believe this research can be extended.

Chapter 2

Background

2.1 Problem Definition

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 basic flow of the *A Priori* algorithm.

The task of frequent itemset mining was first introduced in (AIS93). 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 2.1. If one sets the threshold to be $s = 2$, then

Table 2.1: 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

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 2.1). 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] (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 $\mathcal{U}$. 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 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 (PGG04) 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,

[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.

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. Among those, two stand out in literature because of their positive experimental results. *A Priori*(AS94) is the oldest well-adopted algorithm, but has fallen out of favour for the newer, more popular, and more complex *FPGrowth* algorithm of (HPY00).

2.2 The *A Priori* Algorithm

Shortly after the problem was introduced, Agrawal et al. proposed the *A Priori* algorithm(AS94) to more efficiently solve it. 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. The classical approach to each of these is detailed next.

First however, it is nice to visualise the execution of the algorithm. Frequent itemset mining originated in the analysis of supermarkets, where the surprisingly frequent co-occurrence of beer and diapers was discovered. We adopt this domain for our toy example that we illustrate in Figure 2.1.

To begin, the sets $\{beer, chocolate\}, \{beer, milk\}, \{chocolate, diapers\}, \{chocolate, milk\}$ are all frequent. Then, since $\{beer, chocolate\}$ and $\{beer, milk\}$ share their first element, they are merged into a new candidate: $\{beer, chocolate, milk\}$.

Similarly, $\{chocolate, diapers\}$ and $\{chocolate, milk\}$ create $\{chocolate, diapers, milk\}$.

Finally, we can prune $\{chocolate, diapers, milk\}$ because we see that one of its subsets, namely $\{diapers, milk\}$, is infrequent. So, we need only count the support of $\{beer, chocolate, milk\}$.

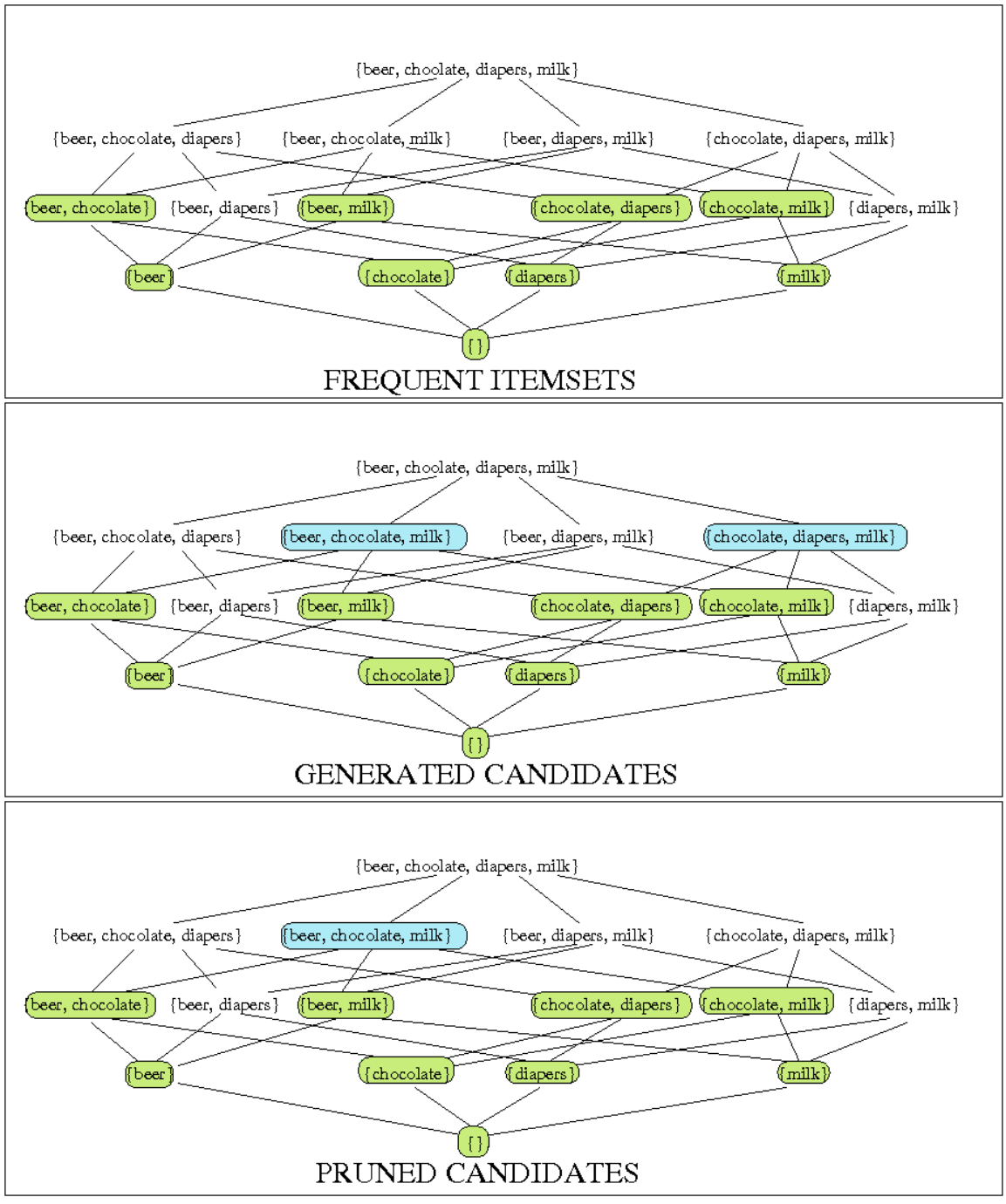

Figure 2.1: Candidate Generation and Pruning in *A Priori*

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

$$k + k - (k+1) = k - 1$$

items. So, all candidates that are generated are done so from frequent itemsets of size k that share in common $k-1$ items. But consider the itemsets of Table 2.2.

Table 2.2: Some 3-itemsets

f_0	a	b	c
f_1	a	b	d
f_2	b	c	d

As we desire, f_0 will be joined to f_1 to form a candidate, $\{a, b, c, d\}$ because they share in common the items 'a' and 'b'. But, so will be f_2 because it shares in common with f_0 the items 'b' and 'd'. Also, f_1 will be joined to f_2. So, the candidate will be produced three times. To remedy this, we can sort all the itemsets lexicographically and impose the condition that some f_i and a corresponding f_j must match on the *first* $k-1$ elements, rather than any $k-1$ elements.

In so doing, one guarantees that a candidate will be generated only once.

Pruning the Search Space

By only generating candidates from frequent $(k-1)$-itemsets as above, one makes use of the *A Priori Principle*. Candidates can only be generated if two particular subsets are frequent. If either happens to be infrequent, then the support of their union is necessarily infrequent, too—and as such the candidate is not generated.

However, this is a weak application of the *A Priori Principle*, because it only necessitates that two particular subsets are frequent, when in fact *all* subsets must be. So the next step is to apply the principle more rigorously by verifying the frequency of all size k subsets. If they are all frequent, then (again using the same principle) all their subsets are frequent and consequently every subset of the candidate is frequent. On the other hand, if any size k subset is infrequent, then the candidate is necessarily infrequent, too, and can be pruned from the search space.

This pruning phase is typically implemented quite naively, probably because no better approach has been proposed.

Support Counting

Finally, one must count the support of those candidate $(k+1)$-itemsets that survived the pruning phase in order to determine whether they truly are frequent. Agrawal et al. accomplished this with a hashing scheme. They construct an array in which to store the counts for each candidate and a hash function that maps the candidates onto the array. They then scan the entire dataset and, for each transaction t in it, extract every size $k+1$ subset of t and apply the hash function to the subsets. If a subset of t hashes to a candidate, they increment that candidate's support count.

After proceeding through the entire dataset, they make one pass through the array and collect those candidates with support counts above the threshold. These are the frequent $(k+1)$-itemsets. It is important to note that the purpose of the hashing is for indexing, not for compression. Each candidate still retains its own support count.

Now, the entire procedure starting with the generation of candidate $(k+2)$-itemsets will be repeated using the new set of frequent itemsets. Algorithm 1 outlines these steps taken together.

Algorithm 1 The classic *A Priori* algorithm

INPUT: A dataset $\mathcal{D}$ and a support threshold s
OUTPUT: All sets that appear in at least s transactions of $\mathcal{D}$

F is set of frequent itemsets
C is set of candidates
C $\leftarrow \mathcal{U}$
while $|C| > 0$ **do**
  {Count support}
  for all $t \in \mathcal{D}$ **do**
    for all $u \in \mathcal{P}(t)$ **do**
      Hash u and increment support count if $u \in C$
    end for
  end for
  for all $c \in C$ **do**
    Hash c and add c to F if $supp(c) > s$
  end for
  Output F
  Clear C

  {Generate candidates}
  Use $(k-1) \times (k-1)$ method to produce C from F

  {Candidate pruning}
  for all $c \in C$ **do**
    for all $i \in c$ **do**
      if $c \setminus \{i\} \notin F$ **then**
        Remove c from C
        break
      end if
    end for
  end for
  Clear F
  Reset hash
end while

2.3 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 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.

Tries

The first of these advances on which we focus is the idea, as introduced in (BMUT97), of storing candidates in a trie. A trie (alternatively known as a prefix tree) is a data structure developed by Fredkin(Fre60) which takes advantage of redundancies in the keys that are placed in the tree. It is easiest to illustrate the data structure with an example. Consider the candidates $\{\{beer, chocolate, diapers\}, \{beer, chocolate, milk\}, \{chocolate, diapers, milk\}\}$. These are, respectively, three different keys that should be inserted into the trie. Because each has three items, they will each contribute to a path of length three. Each item of a candidate corresponds to one node along that path. Then to retrieve a candidate from the structure, one reads the sequence

of items encountered while traversing to the leaf.

The compression achieved by the structure occurs when two keys share a common prefix (such as $\{beer, chocolate, diapers\}$ and $\{beer, chocolate, milk\}$). In this case, the keys can share a common sequence of ancestors the same length as the key (namely nodes corresponding to *beer* and *chocolate*). So, rather than needing six nodes to store two paths of length three, one needs only four nodes. The structure built off these three candidates is shown in Figure 2.2. Today still, this trie idea is a ubiquitous approach and adopted in the state-of-the art *A Priori* implementations(Bor04, Bod03).

As Brin et al. suggest in their paper, the primary bottleneck of the classical *A Priori* algorithm is in incrementing counters for those candidates that are active in a particular transaction. The trie structure helps immensely in this regard 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, we believe. When comparing nearly 1.7 million transactions to 30 million candidates as is done on the webdocs data set, 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 would be invaluable.

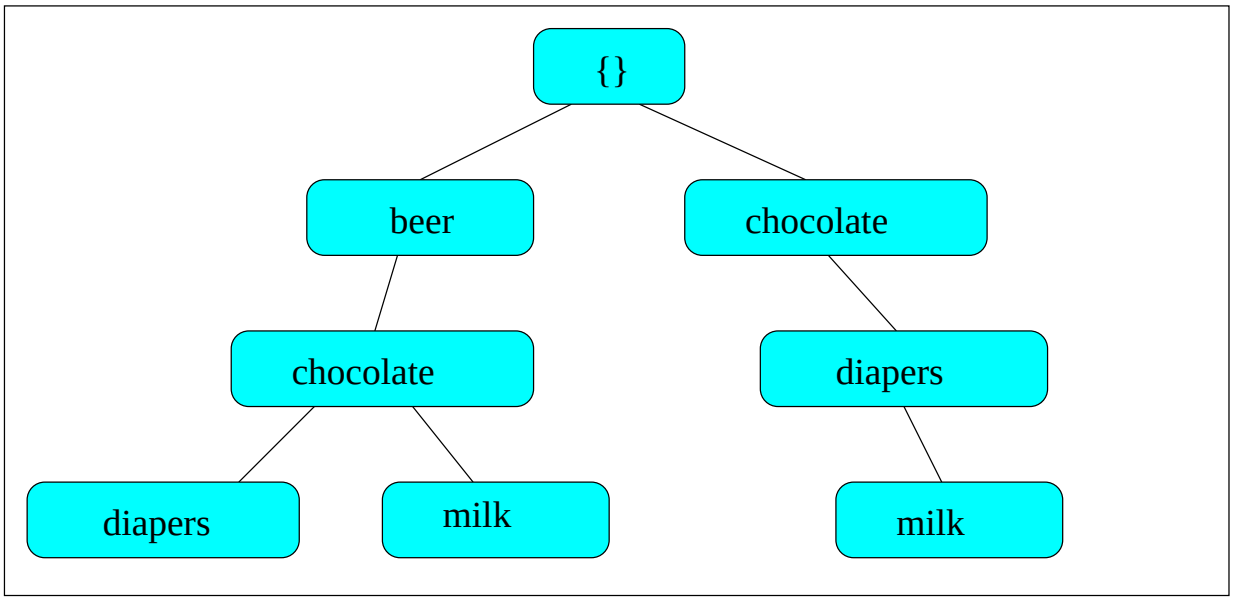

Figure 2.2: Example of Candidates Stored in a Trie Structure

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 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. Consequentely, traversing it causes one to thrash on disk and the efficiency of the structure is quickly consumed by I/O costs.

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 in (Cal02), where Calders demonstrates the sub-optimality of the *A Priori Principle*. He 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*.

FPGrowth

In (HPY00), 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. 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. Taking the example of Table 2.3, the trie constructed in the *FPGrowth* algorithm would appear as in Figure 2.3.

Next, the trie is mined recursively to extract the frequent itemsets. By following the linked-list of nodes labelled 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 labelled 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.

Table 2.3: Frequencies of Itemsets for an *FPGrowth* Example

Itemset Transformed Itemset *LFF*-Sorted Itemset	Frequency
{beer, chocolate, diapers, milk}	5
{beer, chocolate, diapers}	2
{beer, chocolate, milk}	2
{chocolate, diapers, milk}	1
{beer, diapers}	1
{chocolate, milk}	2
{chocolate}	6
{beer}	1
{milk}	10

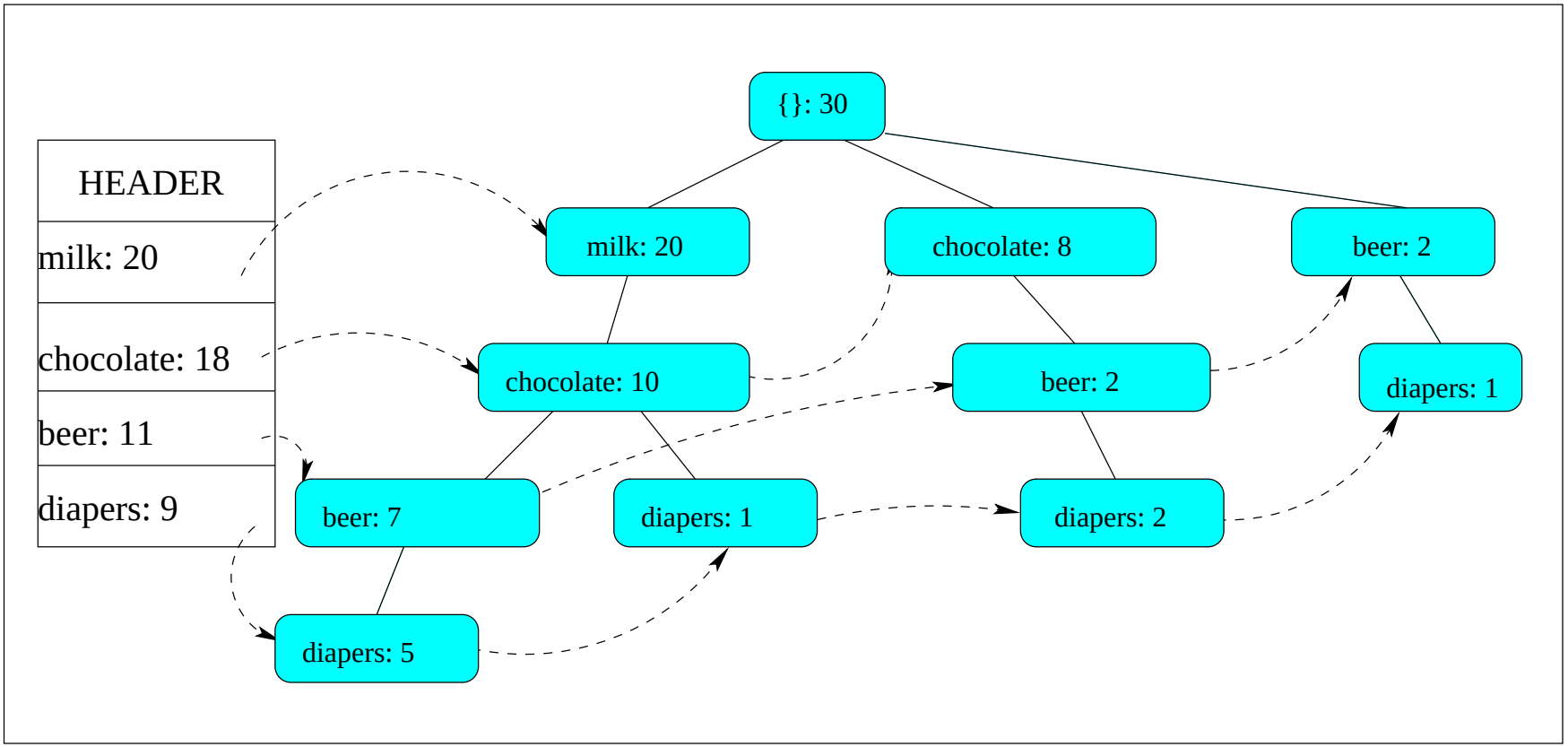

Figure 2.3: Example Trie Structure of *FPGrowth* Algorithm

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 in (BPG06) 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.

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(BPG06); cache-

consciousness(GBP+05); and auxilliary data structures to speed-up the bottleneck of the algorithm(GZ03). 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 in (Goe02) 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(DPVG06).

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.

Attempts at Scaling *A Priori*

In 1995, Savasere et al.(SON95) 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 in (BPG06) 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.

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*(PBTL99, Zak00a). 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.

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

Chapter 3

Vertically Sorted *A Priori*

3.1 Overview of Improvements

The skeleton 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 frequence. Next, we sort each transaction with respect to the frequency order of the their items.[1] 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

[1] 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.

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.

3.2 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 instruc-

tions 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 elementary operations, one still requires about 10.5 minutes to generate these candidates prior even to pruning. In comparison, we count the support of these candidates in roughly 36 minutes. So, we 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 3.1.

As described in Section 2.2, 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.

Table 3.1: 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

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 3.1, 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 3^{rd} 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.

But first, it remains to be shown that our one pass does not miss any potential candidates. Consider some candidate $c = \{i_1, \ldots, i_k\}$. If it is a valid candidate, then by the *A Priori Principle*, $f_i = \{i_1, \ldots, i_{k-2}, i_{k-1}\}$ 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 f_i and f_j are those that share the same $(k-2)$-prefix as they do. The method described above merges together all pairs of frequent itemsets that appear contiguously with the same $(k-2)$-prefix. Since this includes both f_i and f_j, $c = f_i \cup f_j$ must have been discovered.

Candidate compression

Let us return to the concern of generating $\binom{m}{2}$ candidates from each group of m near-equal frequent k-itemsets. Since each group of $\binom{m}{2}$ candidates share in common their first $k-1$ items, we need not repeat the information. As such, we can compress

the candidates into a *super-candidate*.

We illustrate this by reusing the example of Table 3.1 on page 27. 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 3.4.

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 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 i^{th} element in the first row of the array. The i^{th} 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 3.1. The candidates we generated, when stored sequentially as super candidates, appear as below:

6653210565310554320

Table 3.2: 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

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 3.2 indicates how the structure would look if each of these numbers consumed four bytes. (We use -1 in an i^{th} 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 structure is more precise (returns fewer candidates that could not match the transaction) but consumes more memory.

We note here that the idea of building an index on the candidates is not novel. In fact, this is quite apparent in (SA96, BMUT97, BK02). However, the nature of our indexing structure is very different. In (SA96, BMUT97, BK02), the candidates are compressed into a prefix tree in exactly the same way as transactions are compressed into an *FPTree* in *FPGrowth*. Consequently, this indexing structure can suffer the same fate as does an *FPTree* when the number of candidates causes the index to grow beyond the limits of memory.

Our structure does not suffer from the troubles of (SA96, BMUT97, BK02), as is evident in three immediate ways. First, it is more likely to fit into memory, because it only requires storing three numbers for each item, not the entire set of candidates. 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.

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 F_1, then the candidate 2-itemsets are the entire cross-product $F_1 \times F_1$. If we sort F_1 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 change—itemsets from the sorted list of candidates, the list of frequent itemsets will 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 k^{th} 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 F_1 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.

3.3 Candidate Pruning

When *A Priori* was first proposed in (AS94), 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 (DPVG06), 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 frequent items of all those in c. We observed that usually either f_0 or f_1 is the best testset.

Let us consider a classic "beer and diapers" example to illustrate why. Let there be one hundred transactions, ninety of which contain milk and seventy of which contain chocolate. Let the support threshold be 30 transactions and let there also be the frequencies of Table 3.3

Of course, {beer, chocolate, diapers, milk} should not be generated as a candidate because one of its subsets, namely {beer, chocolate, diapers}, is infrequent. Also, note that {beer, chocolate, diapers} is the best testset. Classic *A Priori* uses a lexicographical ordering, so generates this candidate from an (effectively) arbitrary choice of subsets. So, the best testset could not really be expected with a better than $2/(k+1)$ chance.

Table 3.3: Frequencies of Selected Itemsets in a Beer-Diapers Example

Itemset Transformed Itemset *LFF*-Sorted Itemset	Frequency
{beer, diapers} {beer, diapers} {diapers, beer}	35
{beer, diapers, milk} {apples, beer, diapers} {diapers, beer, milk}	34
{beer, chocolate, diapers} {beer, chocolate, diapers} {diapers, beer, chocolate}	25
{beer, chocolate, milk} {apples, beer, chocolate} {beer, chocolate, milk}	50
{chocolate, diapers, milk} {apples, chocolate, diapers} {diapers, chocolate, milk}	40

In the above example using a lexicographical ordering, the candidate would be generated from {beer, chocolate, diapers} and {beer, chocolate, milk} using the $(k-1) \times (k-1)$ method, which uses the best testset. However, if we change the label of "milk" to "apples" and resort, a change that should not really be reflected in the efficacy of the algorithm, then the candidate {apples, beer, chocolate, diapers} will be generated from {apples, beer, diapers} and {apples, beer, chocolate}—no longer using the best testset. So, now, candidate pruning is absolutely necessary.

But using an *LFF* sort, we have some predictability about the testsets that we use. We always try to extend the 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 {chocolate, milk}, and append to them 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 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.

3.4 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 focussed 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. On the webdocs dataset, the transactions sometimes contain over 70000 items. Even after pruning infrequent items with a support threshold of 10%, w still ranges so high as 262. Taken at $k = 4$, this produces

$$\binom{262}{4} = 191868495$$

itemsets. And there are nearly 1.7 million transactions.

So, we abandon the trie.

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 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.

Returning to the example of Table 3.1 on page 27, we had concluded that three super-candidates would be generated: $\{c_0 = (6, \{6, 5, 3, 2, 1, 0\}), c_1 = (5, \{6, 5, 3, 1, 0\}), c_2 = (5, \{5, 4, 3, 2, 0\})\}$. To compare to a transaction, say $t = \{t_0 = 6, t_1 = 4, t_2 = 3, t_3 = 2, t_4 = 1, t_5 = 0\}$ we first look up t_0 in the index and retrieve the first two super-candidates, c_0 and c_1. We then compare them each to t and update the support counts if they are contained in t. (In this case, they are not.)

Next, we proceed to t_1, looking it up in the index. We discover that there are no candidates that begin with 4, so move along to t_2. However, since

$$w - i = 6 - 2 = 4 < k$$

we know that there cannot possibly be any more candidates contained in t, so we are done.

Counting with Compressed Candidates

Recall from section 3.2 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 array, A, of

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

integers 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

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

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.

3.5 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 (BPG06) 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 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.

3.6 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. So, the revisions proposed in this chapter have been

summarised in Algorithm 2. It is perhaps useful to compare this to Algorithm 1, classic *A Priori*, on page 12 to illustrate from a high-level the changes that have been proposed in this research.

Algorithm 2 The revised Vertically-Sorted *A Priori* algorithm

INPUT: A dataset $\mathcal{D}$ and a support threshold s
OUTPUT: All sets that appear in at least s transactions of $\mathcal{D}$

F is set of frequent itemsets
C is set of candidates
C $\leftarrow \mathcal{U}$
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 $f \in F$, *sorted LFF* **do**
  for all $g \in F, supp(g) \geq supp(f)$, *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 $t \in \mathcal{D}$ **do**
    for all $i \in t$ **do**
      RelevantCans $\leftarrow$ using index, compressed cans from file that start with i
      for all CompressedCans $\in$ 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 $\leftarrow 0$
  for $1 \leq i \leq |F|$ **do**
    if $i == |F|$ OR f_i is not near-equal to f_{i-1} **then**
      Create super candidate from f_{start} to f_{i-1} and update index as necessary
      Start $\leftarrow i$
    end if
  end for

  {Candidate pruning—not needed!}
  Clear F
  Reset hash
end while

Chapter 4

Experimental Results

To test the ideas put forth here, we created an implementation in C.[1] 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 1.5GB of webdocs data(LOPS04) 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

[1] A repository has been setup as of May 2009 both at http://webhome.csc.uvic.ca/∼schester and at http://webhome.cs.uvic.ca/∼thomo in which the implementation can be found.

wide selection of the best available implementations of various frequent itemset mining algorithms. Those of Bodon(Bod03) and of Borgelt(Bor04) are state-of-the-art implementations of the *A Priori* algorithm which use a trie structure to store candidates. Lucchese et al.(LOP04) implement an alternative algorithm which exhibited the best performance on this benchmark dataset at the renowned FIMI conference of 2004.(JGZ04) That of Zhu and Grahne(GZ03) 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 similarlity 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.

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 assumptiont that they can serve correctly as an "answer key". But, nonetheless, boundary condition checking was a prominent component during development.

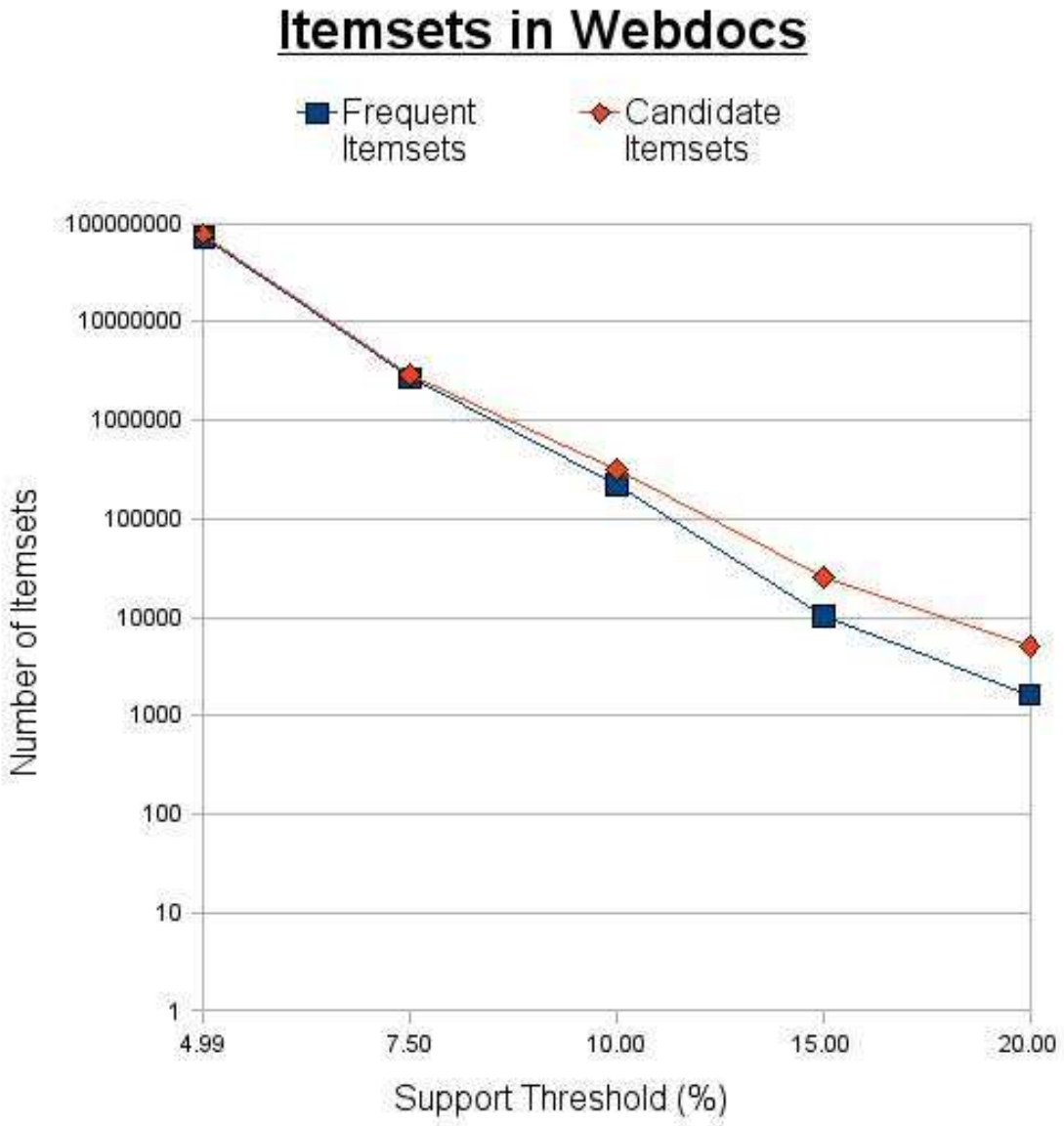

Figure 4.1: Number of Itemsets in Webdocs at Various Support Thresholds

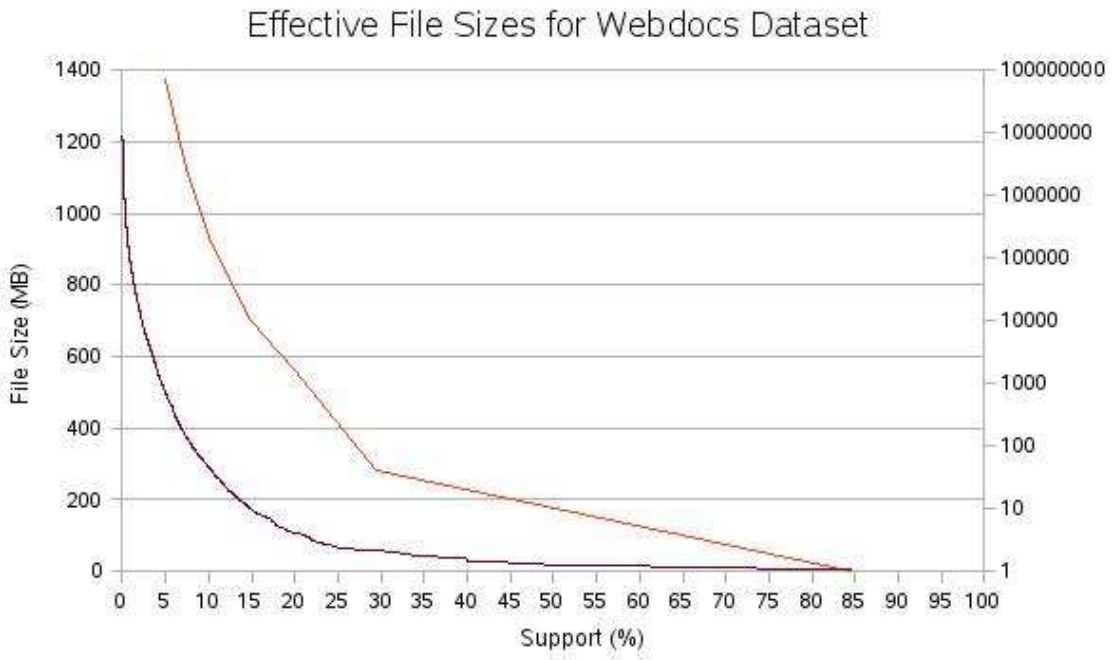

Figure 4.2: Size of webdocs dataset with noise (infrequent 1-itemsets) removed, graphed against the number of frequent itemsets

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 4.1 and Figure 4.2. (The number of candidate itemsets is implementation-dependent and in general will be less than the number in the figure. That value corresponds to the number of candidates that we generated.) 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 4.3, our implementation matches (to within about 10%) or outperforms *all* of the aforementioned state-of-the-art implementations at all support thresholds. Furthermore, no other implementation was able to process as low a support threshold as was ours. The implementations of Zhu and of Lucchese were unable to complete within a reasonable period of time at 10%. Those of Bodon and Borgelt could not compute the frequent itemsets at 5%. Our implementation has finished $k = 8$ at 5% and continues as well as ever.

To explain the difference, Figure 4.4 displays the memory usage of our implementation and of Bodon's implementation 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. However, those of all the other algorithms grow beyond the limits of memory and consequently cannot initialise. Without the data structures, the programmes must obviously abort.

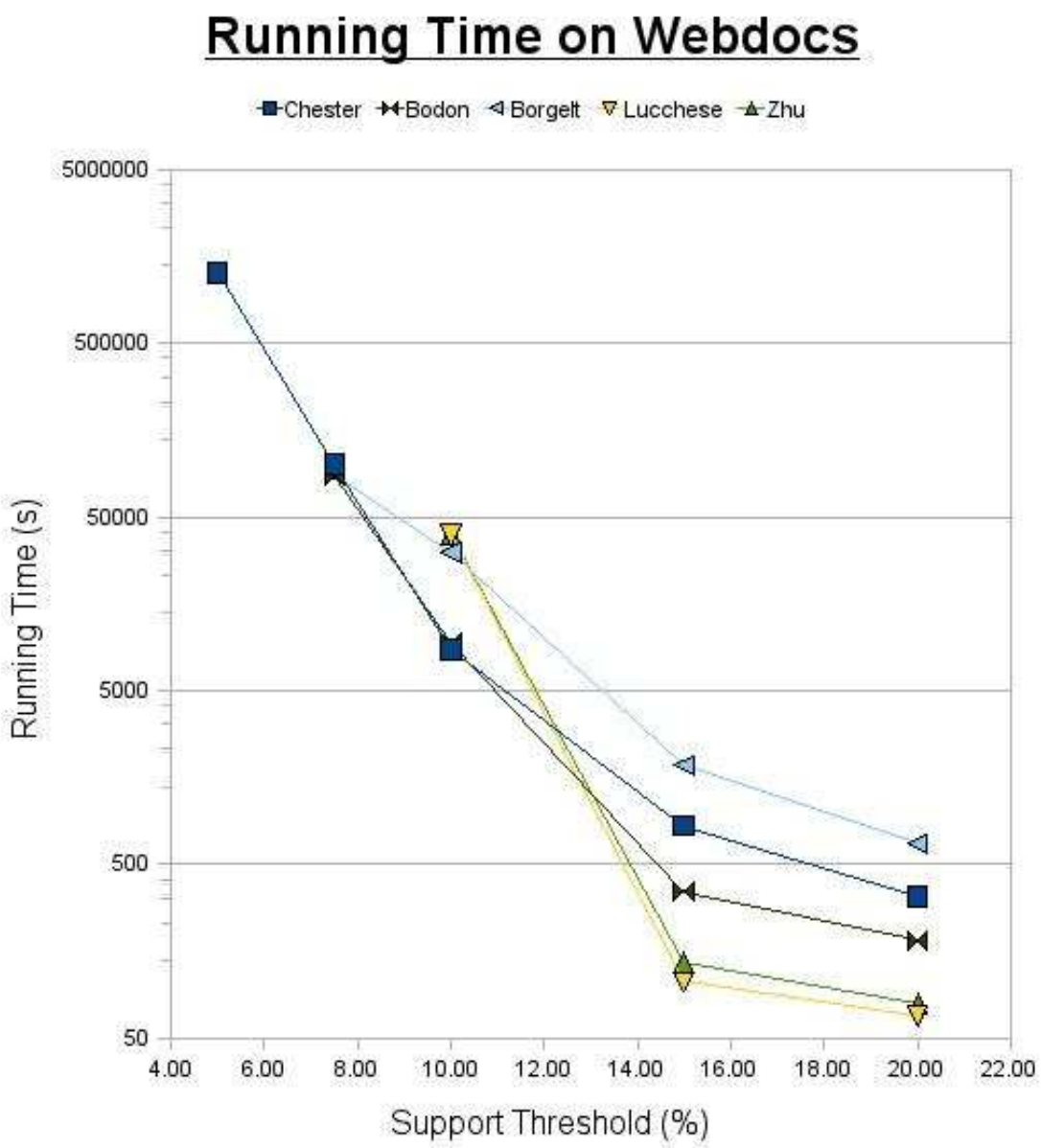

Figure 4.3: Relative Performance of Implementations on Webdocs Dataset

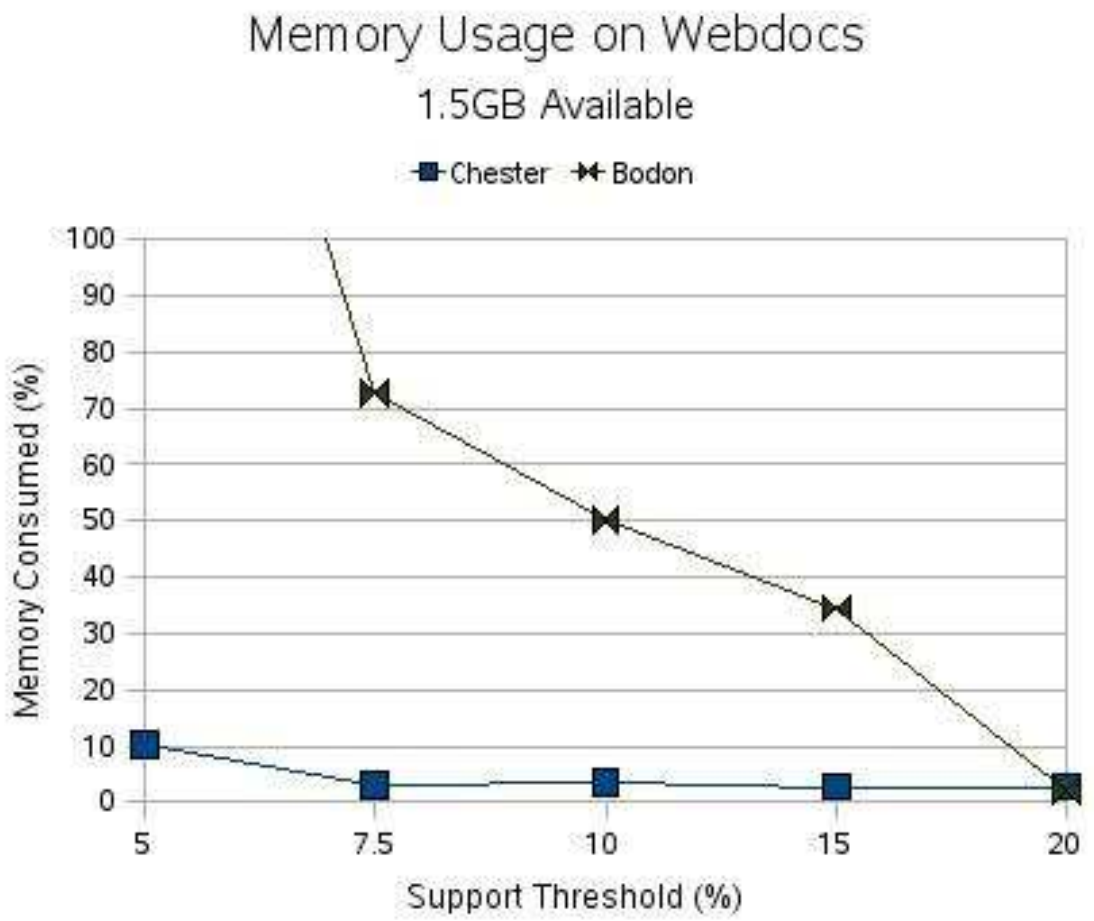

Figure 4.4: Memory Usage of Bodon and Chester implementations on webdocs as Measured by Unix *top* command during late stages of execution

It should be noted that in (BPG06) the authors test their *FPGrowth* implementation on the same benchmark webdocs 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 *FPGrowth* algorithm. As such, no fair comparison can truly be made. Yet still, they only publish results up to 10% which is insufficient as we demonstrated in Figure 4.2. 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 utilisation of our implementation remains relatively constant. As such, our performance continues to follow a predictable trend and our programme can successfully mine suppport thresholds that are impossibly low for the other implementations.

Chapter 5

Conclusion and Scope of Work

Frequent itemset mining is an important problem within the field of data mining, but sixteen years of algorithmic development has yet to produce an implementation that can 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, 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 optimised 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 parallelising 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 optimisation 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 to 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.

Bibliography

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. of VLDB Conference*, pages 487–499, 1994.

[BK02] Christian Borgelt and Rudolf Kruse. Induction of association rules: Apriori implementation. In *Proceedings of the fifteenth conference on computational statistics*, pages 395–400, 2002.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Rec.*, 26(2):255–264, 1997.

[Bod03] Ferenc Bodon. A fast apriori implementation. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of

CEUR Workshop Proceedings, Melbourne, Florida, USA, 19. November 2003.

[Bor04] Christian Borgelt. Recursion pruning for the apriori algorithm. In Jr. et al. (JGZ04).

[BPG06] Gregory Buehrer, Srinivasan Parthasarathy, and Amol 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*, pages 86–95, New York, NY, USA, 2006. ACM.

[Cal02] Toon Calders. Deducing bounds on the frequency of itemsets. In *EDBT Workshop DTDM Database Techniques in Data Mining*, 2002.

[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*, pages 541–545, New York, NY, USA, 2006. ACM.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[GBP+05] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-conscious frequent pattern mining on a modern processor. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 577–588. ACM, 2005.

[Goe02] B. Goethals. *Efficient Frequent Pattern Mining.* PhD thesis, transnationale Universiteit Limburg, 2002.

[GZ03] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Jr. et al. (JGZ04).

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 1–12. ACM, 2000.

[JGZ04] Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki, editors. *FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Brighton, UK, November 1, 2004*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[LOP04] Claudio Lucchese, Salvatore Orlando, and Raffaele Perego. kdci: on using direct count up to the third iteration. In Jr. et al. (JGZ04).

[LOPS04] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Webdocs: a real-life huge transactional dataset. In Jr. et al. (JGZ04).

[PBTL99] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT '99: Proceedings of the 7th International Conference on Database Theory*, pages 398–416, London, UK, 1999. Springer-Verlag.

[PGG04] Paul W. Purdom, Dirk Van Gucht, and Dennis P. Groth. Average-case performance of the apriori algorithm. *SIAM Journal on Computing*, 33(5):1223–1260, 2004.

[SA96] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. 5th Int. Conf. Extending Database Technology (EDBT96)*, pages 3–17, 1996.

[SON95] Ashok Savasere, Edward Omiecinski, and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st VLDB Conference*, 1995.

[Zak00a] Mohammed J. Zaki. Generating non-redundant association rules. In *KDD '00: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 34–43, New York, NY, USA, 2000. ACM.

[Zak00b] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowl. and Data Eng.*, 12(3):372–390, 2000.