

A Distributed Approach To Frequent Itemset Mining At Low Support Levels

by

Neal Clark

BSeng, University of Victoria, 2009

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

Master of Science

in the Department of Computer Science

© Neal Clark, 2014

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

A Distributed Approach To Frequent Itemset Mining At Low Support Levels

by

Neal Clark

BSeng, University of Victoria, 2009

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

ABSTRACT

Frequent Itemset Mining, the process of finding frequently co-occurring sets of items in a dataset, has been at the core of the field of data mining for the past 25 years. During this time the datasets have grown much faster than the algorithms capacity to process them. Great progress was made at optimizing this task on a single computer however, despite years of research, very little progress has been made on parallelizing this task. FPGrowth based algorithms have proven notoriously difficult to parallelize and Apriori has largely fallen out of favor with the research community.

In this thesis we introduce a parallel, Apriori based, Frequent Itemset Mining algorithm capable of distributing computation across large commodity clusters. Our case study demonstrates that our algorithm can efficiently scale to hundreds of cores, on a standard Hadoop MapReduce cluster, and can improve executions times by at least an order of magnitude at the lowest support levels.

Contents

Supervisory Committee	ii
Abstract	iii
Table of Contents	iv
List of Tables	vi
List of Figures	vii
Dedication	viii
1 Introduction	1
1.1 Introduction	1
1.2 Organization	2
1.3 My Claims	3
2 Background	4
2.1 Problem Definition	4
2.1.1 Why this problem is hard	5
2.2 The Apriori Algorithm	6
2.2.1 The $(k - 1) \times (k - 1)$ Candidate Generation Method	7
2.2.2 Candidate Pruning	7
2.2.3 Support Counting	8
2.2.4 Direct Hashing and Pruning Park Chen Yu Algorithm	8
2.2.5 Apriori Example	9
2.3 Recent Advances	11
3 A Scalable Apriori Algorithm	13
3.1 Dataset Analysis	14

3.2	Dead Data Removal	15
3.3	Item Labeling	15
3.3.1	Support Relabeling	17
3.3.2	Randomized Relabeling	18
3.4	Horizontally Sorted Transactions	18
3.5	Vertically Sorted Itemsets	19
3.6	Dataset Partitioning	19
3.6.1	K_{n-1} Itemset Partitioning	20
3.6.2	Partitioned Count Table Generation	22
3.7	Prefix Itemset Trie	23
3.8	Candidate Generation	24
3.9	Candidate Counting	25
3.10	Support Based Candidate Pruning	26
3.11	h-confidence Candidate Counting	27
3.12	Frequent Itemset Output	28
3.13	Transaction Partitioning	29
3.14	Chapter Summary	30
4	Implementation	31
4.1	Hadoop as a platform	31
4.2	Input Preprocessing	32
4.2.1	K_1 Itemset Relabeling	33
4.2.2	Transaction Preprocessing	33
4.3	Itemset Generation	34
4.4	Itemset Data Format	34
4.5	Chapter Summary	35
5	Evaluation, Analysis and Comparisons	36
5.1	Development	37
5.2	Hardware & Software	37
5.3	Results	38
5.4	Chapter Summary	42
6	Conclusions	43
	Bibliography	45

List of Tables

Table 2.1	Webdocs statistics	5
Table 2.2	Example set of transaction D	9
Table 2.3	Example K_1 candidates	9
Table 2.4	Example K_2 candidates	10
Table 2.5	Example K_3 candidates	10
Table 3.1	Effective datasets at various support levels	15
Table 3.2	Candidate itemsets given $U = \{0,1,2,3,4\}$	17
Table 3.3	Number of count table entries per item	21
Table 3.4	Partitioned count table	22
Table 3.5	Candidate suffix counts	23
Table 3.6	Example prefix tree layout	24
Table 3.7	Count table offsets for each candidate itemset	26
Table 3.8	Count table values after each transaction is processed	26
Table 3.9	Transaction partitions	29

List of Figures

Figure 3.1 Support Distribution Histogram	16
Figure 4.1 K_1 Itemset Generation	33
Figure 4.2 K_n Itemset Generation	33
Figure 4.3 K_n Itemset Generation	34
Figure 5.1 Running times on Webdocs	39
Figure 5.2 Frequent Itemsets by phase at various support levels	40
Figure 5.3 Running times by phase at various support and h-confidence levels	41
Figure 5.4 Frequent Itemsets by phase at various support and h-confidence levels	41

DEDICATION

This work is dedicated to the mountains which provide a constant source of inspiration.

Chapter 1

Introduction

Frequent Itemset Mining attempts to extract meaningful information from the vast quantities of transactional data collected. A common example of a transactional dataset is the list of purchases an individual makes at a supermarket. In this example an item is a product, such as bread or butter, and a transaction is analogous to the set of items placed in a shopper's basket. The dataset D is the collection of all such transactions.

1.1 Introduction

Our world generates an astounding amount of data and computer systems allow us to gather and store that data at unprecedented rates. In 2004 IBM estimated that we create 2.5 quintillion bytes of new data each day. It is also estimated that 90% of the data in the world has been created in the past two years [14]. Our ability to analyze the data have been far outpaced by our ability to created it. Hidden within this data undoubtedly lays some of the greatest discoveries of our time.

The human genome, and those of an ever increasing number of species, has now been sequenced. However our ability to analyze these genomes to detect genetic disease is primitive by comparison. Social networks are collecting an human behavior in unprecedented ways. They provide a unique insight into our intricate social structures,

political systems, and every changing ways of life. The large hadron collider records nearly 600 million collisions per second of which over 99.999% are discarded [15]. The remaining collisions represent more than 25 petabytes of data per year. In the face of this the world remains a mysterious place.

Data mining is the process of extracting meaningful information from these massive datasets. This thesis is about that process. *Frequent Itemset Mining* is a process that was proposed over 20 years ago but still remains at the forefront of that field. Its goal is to find sets of items that co-occur throughout the data.

1.2 Organization

In this thesis we take a two pronged approach to the problem. From one side we present a novel algorithm for computing frequent itemsets, and from the other we detail our experimental setup and provide insights into how to work with massive datasets. The challenge is that techniques that work well when the entire dataset can fit in memory can fail spectacularly on larger datasets. Knowing the underlying properties of the dataset as well as careful partitioning of load are just as important as the performance algorithm itself. We do our best to avoid over tuning our solution to a specific dataset and further provide rationale as to the strategy we applied to ensure this.

We begin in *Chapter 2* by formally defining the problem of Frequent Itemset Mining, giving background on recent progress, as well as challenges of working with massive datasets. We include an introduction to the *Apriori* algorithm as well as a review of other relevant work.

In *Chapter 3* we explain our modifications to the *Apriori* algorithm to achieve near linear scalability. We contrast our ideas against those of others and further identify the areas in which we were able to make our improvements. At the end of the chapter we describe some ways the problem can be narrowed, using additional heuristics, to cut out noise and quickly find highly correlated itemsets.

Our experimental results are detailed in *Chapter 4*. The goal of this chapter is two part: to provide an analysis of the performance of our algorithm on a well known

benchmark dataset; as well as attempt to detail our experimental setup in such a way that our results are easily reproducible.

In *Chapter 5* we include our conclusions and suggest ways in which this research can be extended.

1.3 My Claims

The *Apriori* algorithm naturally lends itself to parallelization and distribution; without any loss of efficiency every step can be designed to be parallelized. We started with the classic *Apriori* algorithm and introduced sorting, parallelization, and partitioning to all aspects of the computation. This allows for efficient distribution of computation across an arbitrary number of computers while improving every step of the original algorithm. The result of which is a horizontally scalable algorithm that outperforms all of the existing implementations.

The *Apriori* algorithm naturally lends itself to parallelization as it does not rely on complex in-memory data structures. The unique nature of the algorithm allows each partition to definitively output only frequent items without the need for intermediary communication. Alternative approaches, such as FP-Tree based algorithms [25, 6, 5] do not share this property and attempts to parallelize them have been unsuccessful at lowering support levels. Support is defined as the number of transactions an item must be present in to be considered frequent.

We introduce a number of techniques that allow Frequent Itemset Mining to be efficiently parallelized. We adapt a number of state-of-the-art techniques such as transaction sorting [7], prefix tries, and extend our work using in-memory count tables [20] to allow itemsets to be output in sorted order without any loss of efficiency. Lastly we demonstrate how additional heuristics for quickly finding highly correlated itemsets such as *h-confidence* [22] can be efficiently applied.

Chapter 2

Background

2.1 Problem Definition

Frequent Itemset Mining attempts to extract interesting information from the vast quantities of transactional data collected. A common application is in the field of bioinformatics where Frequent Itemset Mining is used in the interpretation of gene expression data, bimolecular localization prediction, annotations, and protein interaction networks [17].

The goal of data mining in general is to discover interesting or useful relationships from collected data. What makes a relationship interesting is largely dependent on intended use. Frequent Itemsets Mining tries to produce sets of items which can be used to generate association rules.

Let $I = I_1, I_2, I_3, \dots, I_n$ be a set of integer attributes representing the individual items. Let D be the set of all transactions. Each transaction t_i is represented by a array of integers with $t_i[k] = I_m$ where I_m is the item at position k within the transaction. A transaction contains an itemset X if, $X \subseteq t_i$. The support count $supp(X)$ of an itemset X is defined using the following formula:

$$supp(X) = |\{t_i | X \subseteq t_i, t_i \in D\}|$$

In Frequent Itemset Mining a candidate is considered frequent if $\text{supp}(X) \geq s$, where s is the minimum number of times a candidate must appear. We define K as the set of all X satisfied by a transaction in D where $\text{supp}(X) \geq s$.

The Apriori algorithm splits computation into a series of iterations. For ease of notation we define $K_n := \{X : X \in K \wedge |X| = n\}$ or in other words K_n is equivalent to the set of frequent candidates generated in iteration n of the Apriori algorithm.

2.1.1 Why this problem is hard

To provide a better understanding of why this problem is hard we need to look at some of the properties of the computation. $|D|$ is generally extremely large and the set of possible itemset $\mathcal{P}(I)$ is exponential. As a result the search space is likewise exponential.

In fact it has been shown that determining if any sets occur in K_n is NP-complete [19]. Therefore finding the frequent itemsets of any size, K_n for $0 \leq n \leq |I|$, must be at least as hard. To address the NP-completeness any algorithms in this space implement aggressive search space pruning strategies as well as other heuristics.

The *Webdocs dataset* [16] is the largest commonly used dataset in Frequent Itemset Mining. This dataset is derived from real-world data and is reasonably representative of data sets commonly mined. This dataset was chosen because many published results in Frequent Itemset Mining use this dataset and it allows for easier comparison between competing approaches.

Transaction	1,692,082
Longest Transaction	71,472
Unique Items	5,267,656
Dataset Size	1.48 G

Table 2.1: Webdocs statistics

There are two widely adopted approaches to Frequent Itemset Mining; FPGrowth [12] and Apriori [13]. Apriori gained initial success but fell out of favor to the much more complicated FPGrowth algorithm was shown [24] to perform better when the

application fits in memory. Attempts to parallelize FPGrowth [26] have shown some success in reducing execution time however support thresholds are often raised.

Though it is difficult to compare our results to recent publications that have not used Webdocs [26, 6, 4, 9], we have tried to produce a refined experiment by specifying support levels clearly. This is not always done, as in [5]. Since Frequent Itemset Mining is exponential support levels have a dramatic effect on the difficulty of the computation. We believe the quirks of the Webdocs dataset that make it difficult to process are exactly what make it a valuable test piece.

2.2 The Apriori Algorithm

The Apriori algorithm [13] employs a clever heuristic to dramatically reduce the search space. The principle the algorithm exploits is the subsets of an item can not have greater support than its parent. More formally the Apriori principle is defined as:

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

What this principal means is that for any frequent item all subsets of any size must also be frequent. The Apriori algorithm exploits this principal by first finding the frequent K_1 itemsets. Using only these items it then attempts to find the K_2 itemsets and so on. Each iteration of the algorithm must make a complete pass over the transaction dataset.

Each iteration in the classical Apriori algorithm is split into three distinct phases *Candidate Generation*, *Candidate Pruning*, and *Support Counting*. Since these phases occur in series they are often considered independently of one another in the literature.

2.2.1 The $(k - 1) \times (k - 1)$ Candidate Generation Method

The classical approach to candidate generation involves combining candidates from previous iterations to produce new sets. Taking the union of arbitrary sets, of size n , is not guaranteed to generate sets of exactly $n + 1$ elements. There are a number of ways to choose sets to join. For example we could take the cross product, $K_{n-1} \times I$, and filter out any sets that do not contain n elements. However this could include obviously infrequent items into the candidate itemsets.

The standard approach is the $(n - 1) \times (n - 1)$ method. Consider the itemsets of size n this union will contain exactly $n + 1$ items, if and only if, they both share a common subset of size $n - 1$. That is to that that of the n items in each set they share $n - 1$ items. As a result the union of these two sets will contains exactly $n + 1$ items. Sorting the candidate itemsets ensures that the common subset is at the beginning of the itemset and guarantees candidates are generated only once.

2.2.2 Candidate Pruning

The combinatorial nature of the candidate generation step results in many itemsets that are infrequent. Since each candidate must be matched against every transaction in the support counting phase eliminating any obviously infrequent candidates is greatly beneficial.

In the previous step candidates are generated by combining two frequent subsets. This is a weak application of the *Apriori Principal* the itemset may be additional subsets which are infrequent. The next step is to apply the *Apriori Principal* more vigorously and verify every possible subset of size k .

Based on the *Apriori Principal* it follows that if any size k subsets is infrequent the candidate must also be infrequent, the candidate can safely be pruned. However even if all size k subsets are frequent it does not guarantee candidate will also be frequent which brings us to the support counting phase.

2.2.3 Support Counting

The final phase of the *Apriori Algorithm* is to verify which of the remaining $(k + 1)$ candidates are in fact frequent. To do so Agrawal et al. employ a hashing scheme to index each $(k + 1)$ itemset into an array. For each transaction in D they match the transaction against each itemset of size $(k + 1)$. If the transaction contains the itemset; the corresponding value is incremented in the array using the hash function.

At the end of the phase a single pass is made over the array and any candidates which meet the minimum support threshold are considered frequent. The algorithm repeats until no new $(k + 1)$ are found.

2.2.4 Direct Hashing and Pruning Park Chen Yu Algorithm

Direct Hashing and Pruning Park Chen Yu (PCY) [18] algorithm employs a hashing algorithm to approximate support counts of K_2 items. The algorithm works by first hashing the items to counting buckets. The hash function is not 1-to-1 and a number of itemsets can share the same bucket. Before the counting phase each counter is compressed to a single bit. The bit is set to 1 if the counter was greater than the minimum support threshold.

During the candidate generation phases the K_1 and K_2 bitvectors are used to aggressively prune the candidates. The algorithm assumes that the number of false positives is small and that the K_1 and K_2 itemsets are sufficient to prune enough of the infrequent pairs.

PCY works well when most of the counters are below the minimum support threshold. As the support level is reduced too many pairs can map to the same bucket increasing the number of false positives. The algorithm assumes that the system will have sufficient memory and number of false positives will be sufficiently low. These assumptions have been shown to be false when support levels are sufficiently low [21].

2.2.5 Apriori Example

Consider an example of the Apriori algorithm where $I = \{0, 1, 2, 3\}$ and $minsupp = 3$. For this example we employ an in memory count table to merge the *Candidate Generation*, *Candidate Pruning*, and *Support Counting* into a single phase. Instead of generating and pruning candidates prior to counting (k) against each $t \in D$ and increment the counters for any items in $(t - k)$.

ID	Transaction
t_1	0,1,2,3
t_2	0,1,3
t_3	0,1
t_4	1,2,3
t_5	1,2
t_6	2,3
t_7	1,3

Table 2.2: Example set of transaction D

In this example we will create a count table with the K_k itemsets as rows and I representing columns. Traditionally counts have been stored in a hash table however this makes partitioning the counts significantly more difficult. In chapter 3 we outline a number of techniques that dramatically reduce wasted memory and speed computation. Any inefficiencies introduced by this technique are easily offset by adding additional compute resources.

During the first iteration of the algorithm we compute the K_1 candidate itemsets. Using a count table we can efficiently count all pairs in memory. To make the algorithm more general we define $K_0 = \{\{\}\}$. Since every transaction matches against the empty set we count the occurrences of each individual item.

Item	0	1	2	3
$\{\}$	3	6	4	5

Table 2.3: Example K_1 candidates

The frequent K_1 itemsets can be output directly by making a single pass over the count table and outputting any itemsets that meet the minimum support threshold.

Each entry in the table contains the final support count for the candidate itemset. The results of the first are as follows:

$$K_1 = \{\{0\}, \{1\}, \{2\}, \{3\}\}$$

During the second phase we find all itemsets of cardinality 2. Consider the first candidate, $K_n[0] = \{0\}$, with first transaction $t_1 = 0, 1, 2, 3$, in this case $\{0\}$ is the first element and $(t - k) = \{1, 2, 3\}$, so we increment the counters for $\{1\}, \{2\}, \{3\}$ by 1. This process continues until each candidate is matched against every transaction in the database.

The closed itemsets $\{3\}$ have been omitted from the count table as they can not appear as the prefix for any of the K_{n+1} itemsets.

Item	0	1	2	3
$\{0\}$	-	3	1	2
$\{1\}$	-	-	3	4
$\{2\}$	-	-	-	3

Table 2.4: Example K_2 candidates

$$K_2 = \{\{0, 1\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\}$$

Item	0	1	2	3
$\{0,1\}$	-	-	0	0
$\{1,2\}$	-	-	-	2

Table 2.5: Example K_3 candidates

In the third phase the candidate itemset $\{1, 2, 3\}$ appears only twice. As a result, $K_3 = \{\}$, is empty. Since there are no surviving candidates the algorithm terminates.

2.3 Recent Advances

Recent work in the field of Frequent Itemset Mining has largely been directed at distributing computation to leverage additional compute resources. As with all distributed algorithms there are two main problems that need to be addressed, effective partitioning and minimization of communication costs.

As multi-core CPUs started to become more prevalent work began to attempt to leverage these resources [5, 3]. As commodity clusters became available work began to distribute the computation across increasingly larger numbers of machines [4, 6, 25]. The best of these approaches achieved a 1.5x speedup when running on a cluster of 10 PCs. All of these approaches used variations of the FPGrowth algorithm. Inherent in this approach is the need to mine a large tree structure in memory. None of these approaches were able to effectively partition the tree and resorted to passing large amounts of data between nodes. As the number of nodes increased so did the need for increased communication.

The Apriori algorithm has seen little activity. One notable exception is the Chester algorithm [7]. It currently has the lowest published support level on Webdocs at 5% and running times comparable with all of the state-of-the-art algorithms of the time.

Despite the fact Apriori is embarrassingly parallel there are few papers describing attempts to distribute it. IMRApriori [23] implements the Apriori algorithm using the popular MapReduce programming model. Their approach works by partitioning the transaction set across the cluster to compute the partial candidate itemset counts from each partition. The partial counts are then collected, sorted, combined, pruned, and the survivors output. The problem with this approach is that the number of candidates grows exponentially and the communication costs quickly negate any advantages of parallelism.

The IMRApriori algorithm [9] attempts to introduce additional heuristics to reduce the high communication overhead of MRApriori. They introduce the concept of a local support threshold. It is computed by taking the desired support percent and multiplying it by the number of transactions in the partition. This heuristic relies on a number of critical assumptions about the distribution of items in the transaction dataset. Items falling near the global support threshold are likely to fall below the

local support threshold on some partitions resulting in the counts for entire partition to be dropped. The impact of this becomes an exercise in probabilities with variations in dataset, the number of partitions, and the support threshold, all contributing to the accuracy of results.

Chapter 3

A Scalable Apriori Algorithm

In this chapter we dive into detailing out modifications to the standard Apriori algorithm as well as outline the decision making process that lead to them. We show that the effective size of the Webdocs dataset is directly proportional to support. At the most common support levels in scientific literature over 90% of the dataset is never used in the computation.

From here we describe how this unused data can be safely removed from the transaction dataset and then sorted. With much of the data removed there are now large gaps in the label key space where there are no frequent items. We outline and analyze a number of relabeling strategies to remove any unused labels and facilitate better partitioning.

With the unused labels removed we introduce a number of sorting strategies to optimize candidate generation and counting. We now partition the sorted candidate items in order to distribute the computation across the cluster. Each partition is then assigned to a worker node. The candidate itemsets are loaded into a prefix trie to optimize the candidate counting generation steps.

We introduce a novel count table to efficiently store the counts for each candidate while optimizing memory access patterns. The count table also allows for easy candidate pruning and in order output of the frequent itemsets. We detail how to apply the standard support based pruning as well as make an argument for using h-confidence [22] to output only highly correlated itemsets.

With the frequent itemsets identified they must now be output, repartitioned, and used as input for the next phase of the algorithm. Initially we used Hadoop’s built in sorting functionality but this was both unnecessary and costly. Our experiments show that the sort phase represented around half of the total execution time detailed in our results. Although not reflected in our experimental results we detail a method for efficiently outputting the frequent itemsets in sorted order while also allowing for easy partitioning of data.

Lastly, in order to handle truly massive datasets we describe how our algorithm can be adapted to partition the transaction dataset. While this technique introduces considerable overhead it and is unnecessary for processing the Webdocs dataset it makes our approach capable of processing virtually any dataset.

3.1 Dataset Analysis

Before delving into the algorithms it is important to gain understanding of the datasets we are working with. The dataset of choice in papers on Frequent Itemset Mining is the Webdocs dataset [16].

The first iteration of the Apriori algorithm outputs the set of single items which meet the minimum support threshold. By the Apriori principle we know that frequent itemsets can only be comprised of smaller frequent itemsets. It follows that every subset with cardinality of one must also be frequent. Conversely any infrequent subsets of size one can never appear in a frequent itemset.

From this we can conclude that any infrequent single item can be safely removed from the transaction dataset. We define the effective dataset size to be set of transactions remaining after all infrequent single items have been removed. Removing infrequent items from the dataset dramatically reduces the overall dataset size and eliminates unnecessary computation.

The lowest published support level for the Webdocs dataset is 7.5%. At this level only 396 of the original 5,267,656 distinct items survive. The longest transaction drops from 71,472 to 396 and the overall dataset size from 1.48G to 321M.

The Table 3.1 shows statistics on the effective datasets at various support levels.

Support	Support %	Distinct Items	Size (GB)
253,812	15.00%	118	133 M
169,208	10.00%	262	246 M
126,906	7.50%	396	321 M
84,604	5.00%	683	434 M
16,921	1.00%	2,709	777 M
8,460	0.50%	4,413	874 M
1,692	0.10%	12,439	1.0 G
169	0.01%	59,914	1.1 G
17	0.001%	303,506	1.2 G
1	0.00%	5,267,656	1.48 G

Table 3.1: Effective datasets at various support levels

3.2 Dead Data Removal

The initial iteration of the Apriori algorithm computes the set frequent itemsets with a cardinality of one. Before proceeding with the next iteration of the algorithm we can take advantage of this set to remove any infrequent items from the transaction dataset.

3.3 Item Labeling

In the Webdocs dataset each item is represented by a positive integer between zero and the total number of items. However once the infrequent items have been removed there can be large gaps in value between surviving labels. For efficient indexing we want to eliminate any gaps between labels as they would result in unused memory if the labels are used to index directly into an array.

The naive relabeling strategy is to assign a monotonically increasing integer to each item in the order they appear in the dataset. Since the Webdocs dataset is already labeled with integers it is tempting to use this ordering when relabeling. The problems with this approach are not immediately apparent until we attempt to partition the computation. One of the properties of the Webdocs dataset is that a small number of the items have support counts several order of magnitude greater than average.

The following histogram was constructed using the surviving K1 frequent items with a support threshold of 17. The 303,506 pairs were mapped sequentially into 100 buckets and the support counts for each bucket were summed. The sum of supports is used as it approximates the processing and I/O workload of each partition.

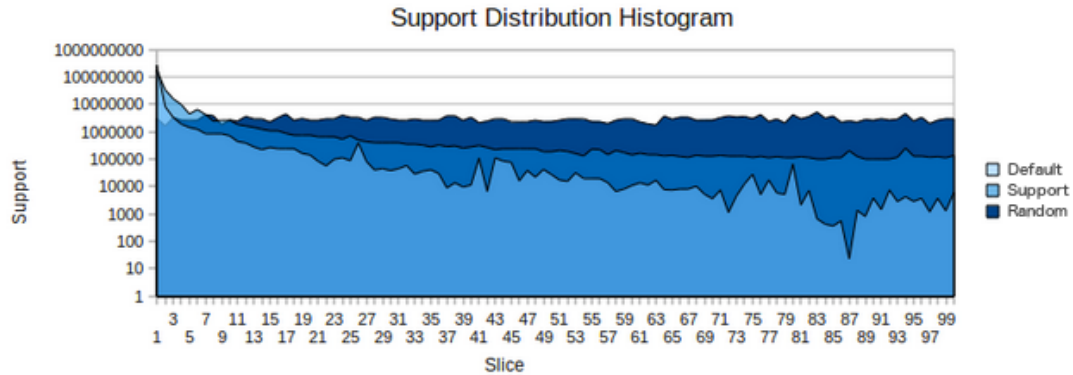


Figure 3.1: Support Distribution Histogram

There is a considerable variability between workloads depending on which items are assigned to a partition. With the naive and support relabeling approaches some partitions have several orders of magnitude larger support sums. This results in partitions which take significantly longer to process. In a distributed environment where all partitions must complete before the next phase can begin this can result in significant slowdowns and wasted resources.

The workload of each partition is difficult to predict as it relies on the relationships between items in the underlying dataset. Without actually computing which items appear frequently together it is difficult to predict how many items will be generated. Using the sum of supports as a metric assumes that more frequent items have a greater probability of appearing together than less frequent ones. While not guaranteed to result in more even partitions in practice we found it proved effective. If necessary the next phase of the algorithm can be run on a random sampling of the transaction set to generate more accurate results.

The memory required for the counting phase is not evenly distributed across all itemsets. Since itemsets are generated in increasing order it means that the prefix ending in the smallest labels can appear in substantially more candidate itemsets than one ending in a large label.

Itemset prefix	Candidate itemset
{0, 1}	{0,1,2}, {0,1,3}, {0,1,4}
{0, 2}	{0,2,3}, {0,2,4}
{1, 2}	{1,2,3}, {1,2,4}
{0, 3}	{0,3,4}
{1, 3}	{1,3,4}
{2, 3}	{2,3,4}
{0, 4}	{}
{1, 4}	{}
{2, 4}	{}
{3, 4}	{}

Table 3.2: Candidate itemsets given $U = \{0,1,2,3,4\}$.

Table 3.3 is sorted by the last item in the prefix itemset to illustrate how the memory requirements are dependent on the value of the last item. Itemsets ending with smaller labels can appear in substantially more candidate itemsets.

Another factor to be considered when relabeling is that high support itemsets have a greater probability of appearing in other high support itemsets. While this is highly dependent on the properties of the underlying dataset experimental results have shown this to be true. Consider the case where a high support itemset also ends with a small label. This means that not only are there more potential candidates there is also greater probability of more of the candidates being frequent.

An effective relabeling strategy needs to allow for easy partitioning, even distribution of frequent itemsets across labels, as well as balancing each partitions computational and I/O requirements.

3.3.1 Support Relabeling

Support based relabeling first sorts the existing labels by support count. This groups the labels in increasing or decreasing order before the new labels are applied.

In the case of decreasing support values the smallest label has the greatest support, the second label the second smallest, and so on. This results in the prefixes ending in the highest support items also having the greatest memory, computational, and I/O

requirements. However this approach does ensure that items with similar support levels appear next to each other; a property we exploit when using h-confidence [22] based pruning.

3.3.2 Randomized Relabeling

From Figure 3.1 we can see that support counts are not evenly distributed across label key space. Underlying patterns in the data lead to uneven partitioning of workload. In the Webdocs dataset, when the labels are sorted value and split into 100 evenly sized partitions, the sum of the supports for each partitions range from a few hundreds to 500 million.

Randomizing item labels allows us to more evenly distribute support values across the label key space. Figure 3.1 illustrates how effective this strategy is on decreasing the variability of support sums across partitions. Our experimental results have shown that this is an effective strategy for balancing load across partitions.

3.4 Horizontally Sorted Transactions

To generate the candidate itemsets of size n our algorithm first finds the existing itemset of size $n-1$ in the transaction. Given an unordered transaction we would have to perform a search to find each item in the k_{n-1} itemset. Additionally, during the counting phase, we must increment the counts for the remaining items. This procedure is repeated for every k_{n-1} itemset on every transaction.

One approach to optimize both lookup time and traversal time is to sort each transaction. Assuming that the itemsets are also generated in sorted order it means that we can match the k_{n-1} itemset against the transaction from left to right. Additionally the counting phase only needs to consider items larger than the rightmost item in the k_{n-1} itemset. This means that once last item in the k_{n-1} itemset has been found the candidate can be counted by counting from the next position in the transaction array to the end.

Longer transactions can easily be searched using a binary search implementation. Subsequent searches only need to search from the last item found to the end of the array. This also results in linear traversals of memory. This improves CPU cache performance and dramatically increases the chances that subsequent memory access will occur on the same page and can reduce thrashing on a memory constrained system.

3.5 Vertically Sorted Itemsets

Maintaining the itemsets in sorted order has a number of benefits including easier partitioning, sequential memory access, and improved performance during the candidate counting phase. One useful property of our algorithm is that it can maintain a total ordering of generated itemsets without the need to sort.

By only matching common prefixes against a transaction once we can dramatically reduce the number of comparisons necessary. When a new prefix is loaded it can be matched against the previous prefix. The transaction only needs to be searched from the point where the prefixes differ.

3.6 Dataset Partitioning

At the core of any distributed algorithm is the ability to effectively partition the workload across computational resources. When support levels drop the size of the frequent itemsets generated during each phase of the algorithm can become orders of magnitude larger than the input dataset. With a 1% support level the K_5 itemsets contain over 8.5 billion itemsets and represents several terabytes of data.

As the dataset size grows the computational and memory requirements of Frequent Itemset Mining can easily exceed the resources available on a single computer. In order to scale the algorithm data must be partitioned across multiple machines. There are two main data partitioning strategies which can be applied to this problem. The first is to partition the transaction dataset by dividing the data into smaller pieces.

This has high costs associated with it as described below but may be necessary in some circumstances. The second is to partition the frequent itemsets output by each phase of computation. In the next section we show that this can be done with little to no loss of efficiency.

3.6.1 K_{n-1} Itemset Partitioning

Due to the nature of the algorithm the size generated itemsets at during each phase often far exceeds the size of the transaction dataset. Carefully partitioning the K_{n-1} itemsets is actually more important than partitioning the transactions.

Partitioning the K_{n-1} itemsets poses a number of technical challenges. Recall in the Support distribution histogram Figure 3.1 we showed that a small number of items have magnitudes larger support levels. As a result itemsets containing one or more of these high frequency items will appear in a disproportionate number of itemsets. This property can result a highly skewed workload. Techniques such as randomized relabeling help to significantly reduce this effect but can not eliminate it completely.

The partitioner must trade off effectively balancing the workload while minimizing the effects of making additional passes over the transaction dataset. Remember that in order to output the K_n itemsets each partition of the K_{n-1} itemsets will need a complete pass over the transactions. In practice this is less of a concern since Frequent Itemset Mining is heavily CPU bound and the size of the K_{n-1} itemsets is often orders of magnitude larger than the transaction dataset.

Consider the Webdocs dataset, with a support level of 1%, the K_3 phase generates 19,395,124 frequent itemsets, by the K_5 phase this has grown to 8,558,613,302, while there are only 1,692,082 transactions in the input dataset.

$$\# \text{ of partitions} = \frac{|K_{n-1}|}{\# \text{ of workers} \times \text{partition factor}}$$

There is one last factor that must be considered and that is the memory requirements of each partition. Each partition will need to build the K_{n-1} itemset trie for its partition as well as a count table. The size of the trie is relatively easy to estimate as

it grows approximately linearly to the input size. Consider the worst case example where no item shares a common prefix. In this case each node will have exactly 1 child so the size of the structure will be $|K_{n-1}[m]| \times n \times c$ where m is the partition number and c is approximate memory requirements of each node.

The size of the count table is a bit harder to estimate. Consider the example illustrated in Table 3.3:

$$I = 0, 1, 2, \dots, 89, 99$$

K_2	# of candidates
{0,1}	98
{0,2}	97
...	...
{0,98}	1
{0,98}	0
{97,98}	1
{97,99}	0
{98,99}	0

Table 3.3: Number of count table entries per item

The memory requirements for each row is determined by the last item in the input itemset. The result is a series of triangular matrices or decreasing size requiring roughly $|K_{n-1}| \times \frac{n}{2}$ count table cells. The problem arises from the fact that the memory requirements are skewed towards the top of the dataset. If we were to simply split the dataset into even sized partitions some partitions would have substantially greater computational and memory requirements.

The solution we propose is to make a pass over the dataset maintaining a sum of the memory requirements for each item output. When the size meets a threshold a new output file is created. This results in partitions of roughly equal size but to further balance the partitions we record the exact memory requirements of each partition. This step can be combined with the output phase to eliminate the need to make a separate pass and allows us to precisely partition the results at the beginning of the next phase.

We can not predict exactly how many output pairs will be generated from any given partition which can result in some partition not being completely filled. To combat this we over partition the dataset and then merge the partitions at the beginning of the next round. If, for example, we over partition by a factor of ten the merged partitions will vary by at most 10%. Table 3.4 illustrates how the count table is partitioned.

Partition	K_0	K_1	0	2	3	4
0	0	1			1	1
0		2				1
1	1	2				1

Table 3.4: Partitioned count table

3.6.2 Partitioned Count Table Generation

The memory needed for the count table is computed as the prefix itemsets are loaded into memory. Each offset into the count table is simply the sum of count table cells up to that point. This offset is then assigned to the corresponding prefix. The range of cells or a prefix is extended over the range of possible candidates.

This approach ensures memory is densely packed and accessed in order as transactions are processed. Sequential memory access has been shown to be up to an order or magnitude faster than random access [15]. Additionally this method minimizes page faults in the event a partitions memory requirements are underestimated.

For each transaction start at the root of the candidate trie. If that item is found continue traversing down the tree until a leaf is reached. The offset stored at the leaf is used to index into the count table.

$$Transaction = \{0, 1, 2, 3\}$$

$$K_{n-1}Item = \{0, 1\}, \{0, 2\}$$

$$Candidateitems = \{0, 1, 2\}, \{0, 1, 3\}$$

Once the K_{n-1} items are located in the transaction the suffix of the transaction contains the items to count. We know that the candidate items can range between $K[1] + 1$ and $N = 3$ where N is the largest label.

Transaction suffix = $\{2, 3\}$

$K[0]$	$K[1]$	2	3
0	1	1	1

Table 3.5: Candidate suffix counts

This item is repeated for the next item in the trie $\{0,2\}$. In this case the transaction suffix is $\{3\}$. Each item in this set is iterated over and the corresponding entry in the count table incremented by 1. Since the transactions are sorted a binary search algorithm can be used to efficiently search within the transaction.

3.7 Prefix Itemset Trie

As additional suffixes are added to the itemsets from the previous round many itemsets will share common prefixes. Since the itemsets are maintained in sorted order the common prefixes will be grouped together. We exploit this property by loading the prefix itemsets into a trie [10]. The trie data structure inherently introduces additional pointers and results in increased memory fragmentation however there are techniques [11] that can be applied to optimize access patterns.

The benefits to storing the K_{n-1} in a trie are realized during the counting phase. Since every candidate itemset must be backed against every transaction a considerable amount of execution time is spent on this task. To match a candidate against a transaction each item in the candidate itemset must be found in the transaction. Placing the the itemsets in a prefix trie allows us to perform only one search per common prefix. Our experimental results show that this leads to a 60% reduction in the number of searches performed.

The prefix trie represented in Table 3.7 is generated by making a single pass over a partition of sorted itemsets. Each internal node contains a list of items and child pointers; the leaf nodes just contain a list items and offsets. As each itemset is loaded it is matched against the previous one. Instead of traversing the trie from the root node each time we start at the previous leaf and work backwards up the tree until a matching prefix is found. From here the algorithm descends creating new nodes for each item.

$$I = \{0, 1, 2, 3, 4, 5\}$$

k_0	k_1	k_2	Offset
0	1	2	0
		3	3
		4	5
	2	3	6
1	2	3	8
		4	10

Table 3.6: Example prefix tree layout

To calculate the offsets a count of the number of cells required is maintained. The offset is simply the current value of this variable. Before moving to the next itemset the counter is updated with the additional memory requirements for the current itemset. Each node is only visited once during during tree construction when the node is inserted.

3.8 Candidate Generation

The candidate generation phase of the algorithm takes the set of candidate itemsets and the transaction set as input. Each candidate itemset is matched against the transaction. If every item in the candidate set is found in the transaction the counts for the remaining unmatched items are incremented. The count table can be thought of as a two dimensional matrix with the candidate itemsets along the left hand side and the k_1 frequent items along the top. The k_{n-1} item creates the prefix of the

candidate item and the k_1 item along the top represents the suffix. To generate a candidate the first step is to match the prefix against the transaction. If each item in the prefix is found the counts for the remaining items in the transaction are incremented.

Leveraging the fact that the transactions are sorted we can search the transaction from left to right. Once the prefix has been matched we can increment the counts for any remaining items remaining at the end of the transaction.

3.9 Candidate Counting

Candidate counting is the core operation of our algorithm. Most of the steps described previously are designed to optimize this operation. There are two main operations, prefix matching and candidate counting, both of which we describe here.

For a candidate itemset to be counted each item it is composed of must be found in the transaction. We have broken the candidate down into a prefix, stored in the prefix trie, and a suffix. The first step is to match the prefix against the transaction. This is done by traversing the prefix trie. At each node a binary search used to find the position of the item in the transaction. If the item is not found we know the entire branch of the trie can be safely ignored. If found, execution progresses down the tree until a leaf is found.

Remember that we sorted the transactions in ascending order. As a result any items from the last prefix item to the end of the transaction can be treated as candidates. To update the counts we simply iterate from the position of the last time to the end of the transaction incrementing the counts by 1.

The offsets on the leaf nodes are used to find the correct position in the count table. The exact position can be computed using the following formula:

$$I = \{0, 1, 2, 3, 4\}, t = \{0, 1, 2, 3, 4\}$$

$$pos(j, t, i) = \text{offset} + (t[j] - i[-1] - 1)$$

i	offset	m	$i[-1]$	$j = 2$	$j = 3$	$j = 4$
{0,1}	0	1	1	0	1	2
{0,2}	3	2	2		3	4
{1,3}	5	3	3			5

Table 3.7: Count table offsets for each candidate itemset

As each transaction is processed the candidate counts incremented in the count table. The algorithm has been carefully designed to make a series of linear passes over the count table. This was done to optimize memory access patterns.

Transaction	Count Table					
	0	1	2	3	4	5
{}	0	0	0	0	0	0
{0,1,2,3,4}	1	1	1	1	1	1
{0,1,2,4}	1	1	1	2	1	3

Table 3.8: Count table values after each transaction is processed

3.10 Support Based Candidate Pruning

Once the candidate itemsets have been counted the next step is to output any frequent itemsets. To do so we make one final pass over the prefix tree. As we traverse down the tree we build up the candidate itemset. Once at a leaf we iterate over the counts stored in the count table. Any entries with a value greater than the minimum support threshold represent frequent itemsets. The candidate and count is then output to file. This process is repeated for every entry in the prefix tree until all candidates have been evaluated.

The candidates consist of a prefix and suffix. The prefix is generated as the prefix tree is walked and the suffix corresponds to the position of the count in the count table. This is a highly efficient method of storing candidates and counts. Only as the candidates are output are they generated. The frequent itemsets are output as $\langle itemset, support \rangle$ pairs. This is not the optimal format for storing the itemsets but allows for easy partitioning of the data for the next round.

3.11 h-confidence Candidate Counting

Noise in the transaction dataset can result in the generation of many itemsets that are not necessarily strongly correlated with one another. Work by Sandler et al [20] has shown that with a support level of 10% on the Webdocs dataset none of the pairs generation are even moderately correlated with one another.

The h-confidence metric was proposed as an additional candidate pruning strategy to find highly correlated itemsets. We apply the Apriori principal the h-confidence values to allow us to aggressively prune the search space.

Traditionally the h-confidence metric is computed as follows:

$$\text{h-confidence} = \frac{\min(\text{support}(k))}{\max(\text{support}(k))}$$

However, we can leverage a few of the unique properties of our algorithm to dramatically reduce the number of candidates generated. By applying the support based relabeling strategy we ensure that smaller item labels will always have larger support values. It follows that for any generated itemset the smallest label has the maximum support value and the largest label the minimum.

Both the prefix trie and transactions are horizontally sorted we then know exactly which items have the minimum and maximum support values.

This allows us to simplify the h-confidence metric as follows:

$$\begin{aligned} \max(\text{support}(k)) &= \text{support}(k[0]) \\ \min(\text{support}(k)) &= \text{support}(k[n]) \end{aligned}$$

Given a candidate item $K = \{0,1,2\}$

Therefore the h-confidence value for a candidate itemset is always:

$$\text{h-confidence} = \frac{\text{support}(k2)}{\text{support}(k0)}$$

Since we already know the support values for each item and we also know the minimum confidence threshold we can compute the maximum item label that falls above the h-confidence threshold. Any item with a label larger than this value can safely be excluded from the computation. We use this information when generating the count table to remove candidate cells that have no possibility of surviving the pruning stage.

We also exploit this property during the candidate counting phase. Each transaction is also sorted in ascending order this means that we can apply this same principle to the candidate counting step. As we iterate over a transaction as soon as an item falls below our precomputed minimum we can stop.

In this section we described how h-confidence pruning can be efficiently implemented. Through the clever application of the Apriori principle we showed how only the count table generation and candidate counting stages can be modified to only generate pairs that meet the h-confidence threshold. The only additional work required is the computation of a lookup table to determine the range of labels that meet a particular h-confidence threshold.

3.12 Frequent Itemset Output

Once the candidates have been pruned the next step of the algorithm is to output the surviving itemsets to disk. As the output from one phase becomes the input to the next we can take this opportunity to prepare the data for partitioning. At this point there are a few properties we can exploit. Firstly, the surviving itemsets are stored in sorted order in the candidate trie. We also know the relative position of this dataset in relation to the total ordering of all partitions.

3.13 Transaction Partitioning

To handle massive transaction sets it may be necessary to partition the transaction dataset. In order to confidently prune candidate itemsets all counts for a particular candidate item must be counted. When the transaction set is partitioned this is not possible as each partition has no knowledge of the data in the other partitions. Without combining the partial sums from each partition we can not be absolutely sure if a candidate itemset is frequent or not.

This can be accomplished by adding a processing step to our algorithm in between the candidate counting and pruning steps. After the counting step the partial sums are output and then combined. Table 3.13 illustrates the output generated by each partition. In this example the counts in files counts_0_0 and counts_1_0 are summed together.

Itemset Partition	Transaction Partition	Count Table Partition
0	0	counts_0_0
	1	counts_0_1
	...	
	k	counts_0_k
1	0	counts_1_0
	1	counts_1_0
	...	
	k	counts_1_k

Table 3.9: Transaction partitions

The partial count tables are summed together to calculate the total number of occurrences for each item. Since the countable is a linear array references by offsets the count tables can be output by outputting the count table directly to disk. To sum the count tables can be treated as integer arrays and summed together. From here we can reload the prefix trie and progress as usual with candidate pruning and frequent itemset output.

3.14 Chapter Summary

We began this chapter with a detailed analysis of the Webdocs dataset to make the argument that the existing algorithms have barely been able to cut through the statistical noise. Even with a 5% support level only 683 of 5,267,656 distinct items are considered by the state-of-the-art approaches.

We introduce a number of techniques beginning with sorting and preprocessing to minimize wasted computation, reduce communication costs, and efficiently partition the data. We demonstrate how the output from each stage of the algorithm can maintain the necessary sorts with little overhead. Combined with a novel count table structure, computation is efficiently partitioned across any number of computers. Our distribution strategy permits each partition to authoritatively output frequent itemsets without the need for intermediary communication. This provides substantial performance improvements over existing techniques and allows our approach to scale linearly with respect to the number of nodes available.

We use a candidate prefix trie compress and optimize the candidate counting and generation phases. This structure supports direct indexing into the count table allowing for a dense packing of count table cells and permits in-order memory traversal dramatically improving CPU cache performance.

The candidate pruning and frequent itemset output phases are combined into a single pass over the count table. We detail how the traditional support based pruning can be implemented as well as show how the algorithm can easily be extended to support additional pruning strategies such as h-confidence.

We believe our approach can easily be extended to support larger datasets and detail how the transaction dataset can be partitioned. The count table can easily be passed between nodes holding the same partition and summed without the need to reconstruct any of the in memory data structures. We illustrated our changes using the Webdocs dataset but believe our techniques are applicable to any transactional dataset.

Chapter 4

Implementation

To test our ideas we developed a core implementation in Java executed on a Hadoop cluster. We concerned ourselves mainly with performance on large datasets in a distributed environment. It is the ability to process massive datasets which makes this study interesting. To test our ideas we chose the Webdocs [16] dataset as it is the largest commonly used dataset in publications for this problem. At all but the lowest support levels Webdocs is of quite modest size when run in a distributed environment.

We considered generating our own synthetic dataset however it would provide little benefit or comparing our results with existing research. Additionally the data in Webdocs comes from a real world problem and therefore provides a more meaningful analysis. We felt the risk that synthetically generated data would not accurately approximate real world data outweighed the benefits of having arbitrarily large datasets to test our algorithms against. Instead we will demonstrate how our approach scales linearly with respect to the number of available compute cores and therefore can be applied to almost any dataset.

4.1 Hadoop as a platform

When distributing computation over hundreds or thousands of nodes faults are the norm not the exception. Reliable execution demands the ability to gracefully recover from faults. Hadoop provides a distributed execution framework with facilities managing the reliable execution of distributed jobs across a cluster of computers.

The Hadoop [2] project was started at Yahoo to provide an open source implementation of Google's MapReduce [8] programming model. In the traditional MapReduce model there are two functions *map()* and *reduce()*. Before executing the map function the input dataset is partitioned between the available workers. Each worker then applies the map function to its partition of the input data mapping them into $\langle key, value \rangle$ pairs. Between the map and reduce phases similar keys are grouped together before being passed to a single reduce function. The reduce function takes in a key and a list of values, performs some computation, and outputs a set of $\langle key, value \rangle$ pairs.

In order to efficiently implement this programming model a robust distributed execution environment is needed. Hadoop supplies advanced scheduling functionality gracefully handling node failure, task re-execution, as well as efficient distributed file system called The Hadoop Distributed File System (HDFS). Additional functionality such as backup tasks help to mitigate the straggler problem and improve execution times. With a little creativity Hadoop provides an ideal execution environment for a number of computational problems that do not necessarily fit the MapReduce model.

We exploit the fact that map and reduce tasks are in essence fully functional Java programs. Side effects are discouraged in MapReduce, there is no practical reason they can not be used. In this chapter we will describe how we leveraged Hadoop to implement an efficient fault tolerant Frequent Itemset Mining algorithm.

4.2 Input Preprocessing

As we outlined above the first step is to pre-process the input dataset to compute the K_1 itemsets and to remove any infrequent items from the transaction dataset. A simple method of computing the K_1 itemsets is to use the standard MapReduce word count example. For each item in every transaction a $\langle item, 1 \rangle$ pair is generated. A combining function is run on each map task to locally sum the local pairs with the same key. The reduce phase then sums and outputs the total frequency counts for each item. During this step we can also prune any infrequent items.

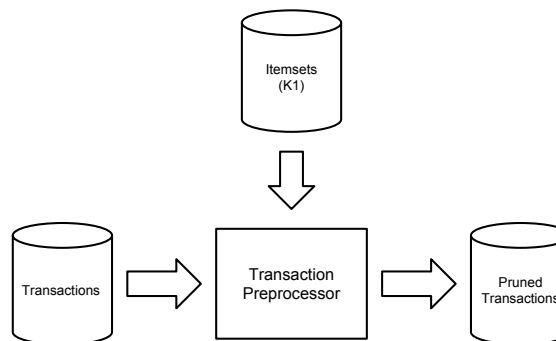
Figure 4.1: K_1 Itemset Generation

4.2.1 K_1 Itemset Relabeling

Once the K_1 have been generated the next step is relabeling; this task can easily be accomplished in memory. We reorder the itemsets and assign them an integer identifier as they appear in the file. The next step is to relabel and prune the transaction dataset. When the transaction set is small, like the Webdocs dataset, we can easily preform this operation on a single node.

4.2.2 Transaction Preprocessing

Once we have the relabeled K_1 itemsets we can remove infrequent items and prune empty transactions from the transaction dataset. To accomplish this we create a map only MapReduce job. Each task loads the K_1 itemsets into memory and receives a partition of the transaction dataset. For each transaction infrequent items are removed and the remaining items sorted in ascending order while any empty transactions are dropped.

Figure 4.2: K_n Itemset Generation

4.3 Itemset Generation

The main processing steps of our algorithm run within the Map states of a single MapReduce job. We created custom Hadoop partitioner class to read the partition file and generate even partitions from the inputs of the previous step. The partitioner sorts the entries in the partition file to create a total ordering of all K_{n-1} itemsets. From here the over-partitioned itemset files are split into roughly even input splits. These splits are run into the Mapper of a standard MapReduce job. We use Hadoop's distributed cache to distribute the preprocessed Webdocs dataset. This approach has the additional benefit of allowing the Mapper to decide how much of the input split to process at a time. Ideally the Mapper will be able to consume the entire split but in the event that the split is too large a second pass can be used to handle the excess data without risking swapping memory.

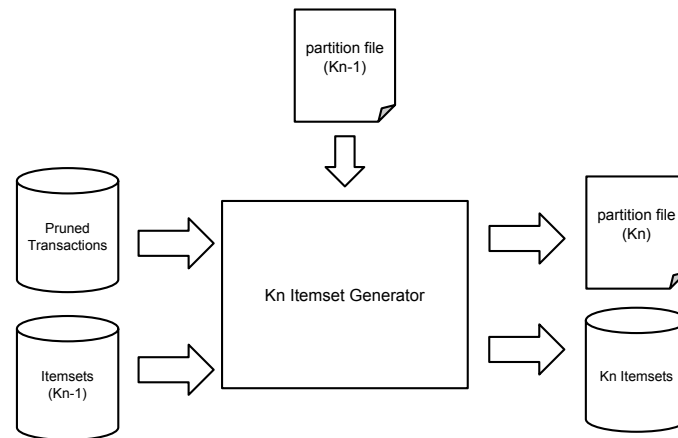


Figure 4.3: K_n Itemset Generation

4.4 Itemset Data Format

Before we get into generating itemsets we will briefly touch on the topic of data formats. To output the frequent itemsets we created a custom Hadoop OutputFormat implementation. Traditionally Hadoop MapReduce has used text files however this substantially increases the volume of data needed to represent the itemsets. We chose a flat binary structure where each entry is represented a $\langle \text{itemset}, \text{support}, \text{h-confidence} \rangle$ tuple. The tuple is serialized as a tuple of integer values with a double

representing the h-confidence value. This data format is highly compressible and we pass this data through the speedy compression algorithm to minimize disk space used.

There are a number of methods that can be applied to more efficiently encode this dataset. For example instead of outputting each item we could write the prefix trie directly to disk. While this would dramatically decrease the disk space requirements it complicated the output process and makes partitioning more difficult.

The second goal of our `OutputFormat` is to split the output data into smaller files that can easily be recombined into evenly sized input splits for the next phase of the algorithm. To do so we write a specified number of entries to each file and then log the first itemset and number of itemsets to the partition file. This information can be used to efficiently generate a total ordering of all output itemsets without the need to sort the contents of the files. Our experimental results showed that more than half the running time of our algorithm occurred during the reduce phase were initially used to perform this sort.

4.5 Chapter Summary

In this chapter we detailed the infrastructure and implementation details used in our implementation. Hadoop is a powerful distributed computing environment but the MapReduce programming model is not well suited for many computational problems. We describe how to leverage the Hadoop platform as a robust execution environment while avoiding the performance pitfalls other approaches experienced.

Our Map only MapReduce job allows each Mapper to run an entire phase of the algorithm without the need for intermediary communication. Our approach eliminates the bottleneck that has prevented other approaches from scaling effectively. At the end of each phase the frequent itemsets are serialized to disk using a simple but highly compressible flat binary format.

Chapter 5

Evaluation, Analysis and Comparisons

In this chapter we set out to experimentally prove the main thesis of our research. Our goal is to demonstrate that parallelization of Frequent Itemset Mining is not only possible but can dramatically decrease execution time and enable mining at lower support levels than any of the existing state-of-the-art implementations.

To facilitate easy comparison of our results with the existing approaches we choose to run our experiments against the Webdocs dataset[16]. Comparing distributed algorithms against serial ones can be difficult. There is inherent overhead in distributing an application so the benefits of parallelization may only become apparent when the workloads are sufficiently large. To complicate matters very few algorithms are able to handle a 5% support level and there are no results available below that threshold.

Other benchmark datasets are significantly smaller and therefore are of little interest for testing our ideas. The overhead of provisioning and managing the cluster will significantly overshadow any performance gains provided by parallelization. We considered generating larger synthetic datasets but feel that they would not be representative of real world datasets and may fail to provide meaningful analysis of the performance characteristics.

5.1 Development

Choosing an appropriate programming language is a difficult problem. Most of the existing implementations used C++ or C however these languages are not well supported in distributed environments. Hadoop has made Java the defacto standard for distributed processing. While it does support calling external applications written in C/C++ the lack of native access to the underlying Hadoop APIs would make their use impractical.

We did some early experimentation with Python and the Disco MapReduce framework [1] however it performed orders of magnitude worse than Java. The key operation in our approach are increment operations on static linear arrays. Python only supports associative arrays which are a much more memory and computationally expensive than your standard C or Java arrays. We tried a number of numerical and scientific computing libraries but the overhead of accessing native code also proved to be prohibitively expensive.

In the end we settled on Java. It performs very nearly as well as C/C++ for the types of operations we require and is natively supported by Hadoop. Since virtually all of the memory is allocated during the initial phases the need for garbage collection was minimal so did not significantly impact performance.

To verify the correctness of our implementation we compared our results against those from the FIMI conference of 2004. All of the implementations presented at this conference produced identical results when run against the published datasets. Careful unit testing was used extensively during development to ensure the correctness of our implementation.

5.2 Hardware & Software

Our experiments were run on a standard Hadoop cluster. Hadoop is the defacto standard in distributed data processing. It provides a distributed execution environment, advanced job scheduling, and the Hadoop Distributed File System (HDFS). While

Hadoop introduces some overhead we felt that the ubiquity of Hadoop for data processing would make our results more reproducible.

Hadoop was configured to run on 40 nodes of an 42 node IBM Blade cluster. Each node in the has 4 Xeon cores running at 2.33 gigahertz and 2GB of system memory. Hadoop was configured to assign one worker process to each available core. Each worker process was allocated 300MB of memory and to ensure our results were not affected by memory swapping we configured the OS to place a hard limit of 350MB. If the memory limits were exceeded the process would be immediately terminated.

We tried to stick to the default Hadoop configuration as much as possible. We did enable speculative execution, in the Hadoop configuration files, to better take advantage of the available cores and help to reduce the impact of stragglers on program execution.

5.3 Results

We ran our implementation at support levels of 15%, 10%, 5% and 1%. At low support levels our algorithm matched or outperformed most of the state of the art algorithms. However the overhead of initializing the cluster represents the majority of the execution time. At lower support levels the algorithm outperformed all of the algorithms. At 5% our algorithm completed an order of magnitude faster than than the best known algorithm.

At 1% the time per frequent pair continued to improve however we ran out of disk space while computing the k_6 frequent itemsets. The k_5 phase generated over 8.5 billion frequent itemsets. There was no indication that the performance of the algorithm was degrading. It is reasonable to assume that if more disk space was made available the algorithm would complete successfully.

Figure 5.1 shows a comparison of the current state-of-the-art FPGrowth and Apriori implementations. We gathered the statistics on the running times of each algorithm from their respective publications. As expected the two FPGrowth algorithms, Lucchese and Zhu, perform best at support levels of 20% and 15% however become

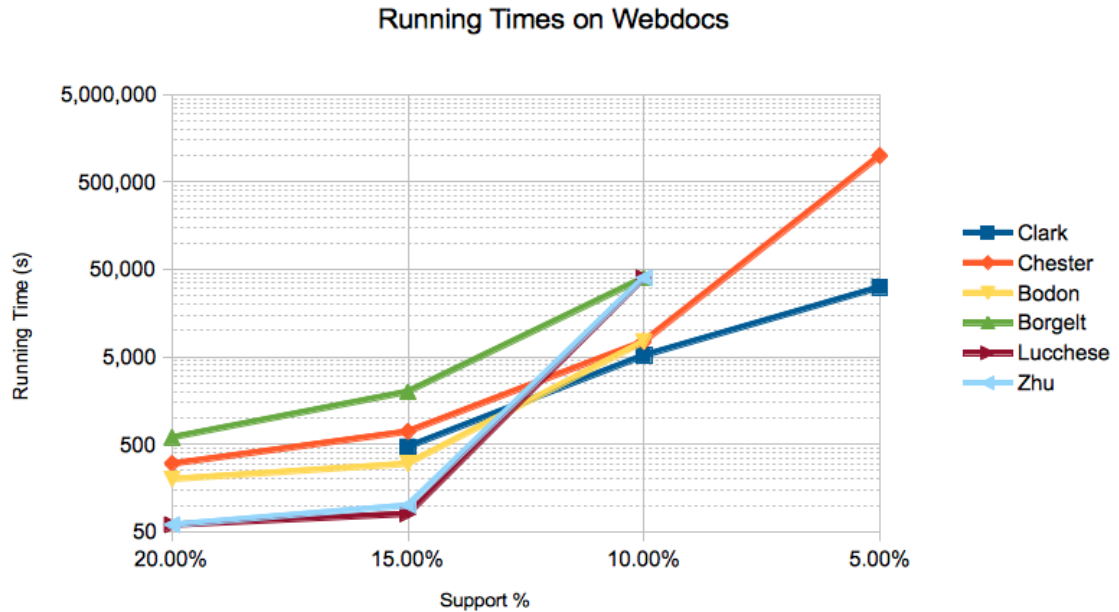


Figure 5.1: Running times on Webdocs

significantly slower at 10%. The in core Apriori algorithms, Borgelt and Bodon, both manage to attain a support level of 10% but their performance starts to degrade as memory becomes scarce. To the best of our knowledge the Chester algorithm currently has the lowest support level in academic literature at 5% and performs well at higher support levels.

At lower support levels our algorithm had comparable execution times but used substantially more resources. Most of the time was spent initializing and managing the cluster. It is only at lower support levels when the overhead of parallelization begins to outweigh the costs. However at a 5% support level the benefits become clear. Our algorithm achieved a 10x reduction in running time over the Chester algorithm and was showing similar scalability at 1% support before the frequent itemsets exhausted all available disk space on the cluster.

Figure 5.2 provides a breakdown of the number of frequent itemsets output after each phase of the algorithm. At each support level we show the number of frequent itemsets generated. It is quite striking how quickly the number of itemsets increases as support drops to 5% and 1%. Perhaps it is unsurprising given the exponential nature of the algorithm however it outlines just how important mining at lower support levels is.

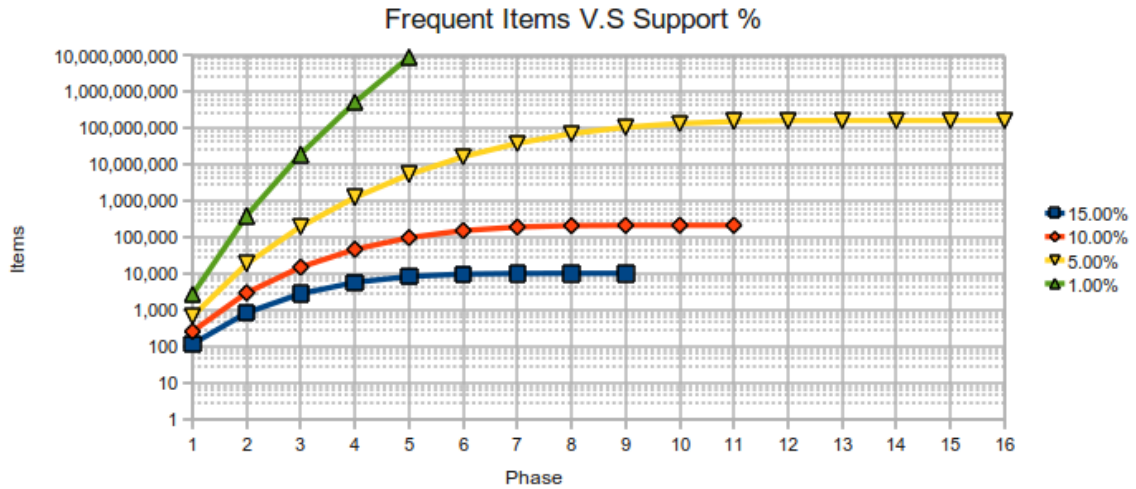


Figure 5.2: Frequent Itemsets by phase at various support levels

With the job overwhelming our cluster with a 1% support value clearly a more aggressive pruning strategy is necessary. To explore even lower support levels we apply the h-confidence variant of our algorithm as described in Chapter 3. Consider the itemsets generated after round 5 with a support level of 5%. Without h-confidence filtering there are over 8.5 billion frequent itemsets however only 500,000 survive a 25% h-confidence value and 100,000 a 50% h-confidence value. We believe it is safe to conclude that most of the itemsets generated are likely just random noise.

An interesting result is that the running times and number of itemsets generated are primarily limited by the h-confidence value and not the support level. In Figure 5.3 the running times for the 50% h-confidence trials at 1%, 0.5% and 0.1% are almost identical and so are the number of items sets generated as shown in Figure 5.4. However at 25% h-confidence this property is still present but no longer as distinct. This is likely due to the fact that as items become less correlated the amount of noise increases. Lower support levels allow more of this noise to slip into the results.

Our experiments have demonstrated that our algorithm can outperform the best state-of-the-art implementations at low support levels. When other implementations start to exhaust resources of a single machine ours can efficiently scale to virtually any size cluster. With sufficient resources there is no theoretical limit how low of a support level can be achieved. With more aggressive heuristics like h-confidence, substantially lower support levels can be investigated.

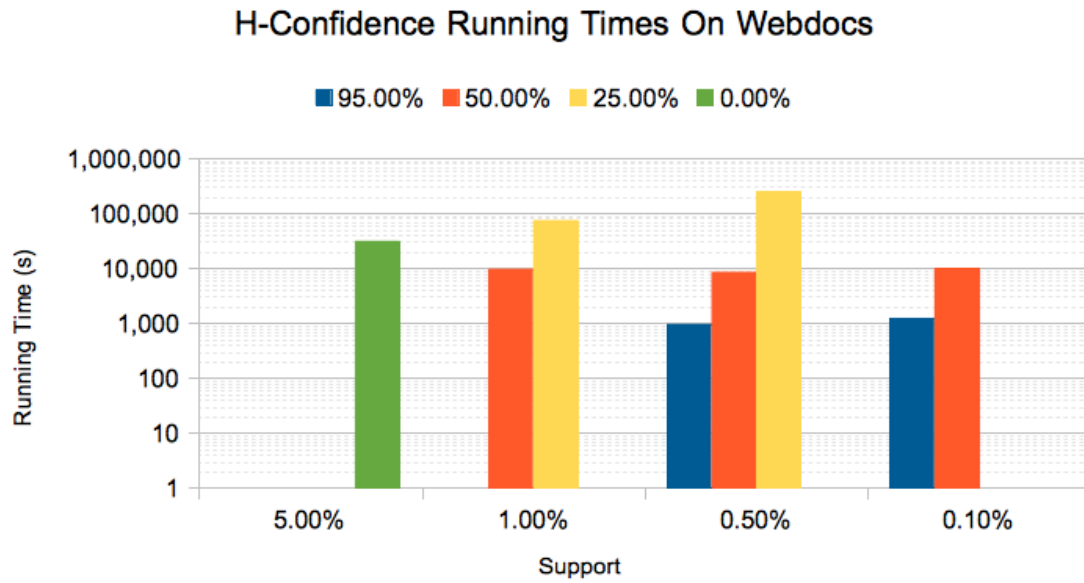


Figure 5.3: Running times by phase at various support and h-confidence levels

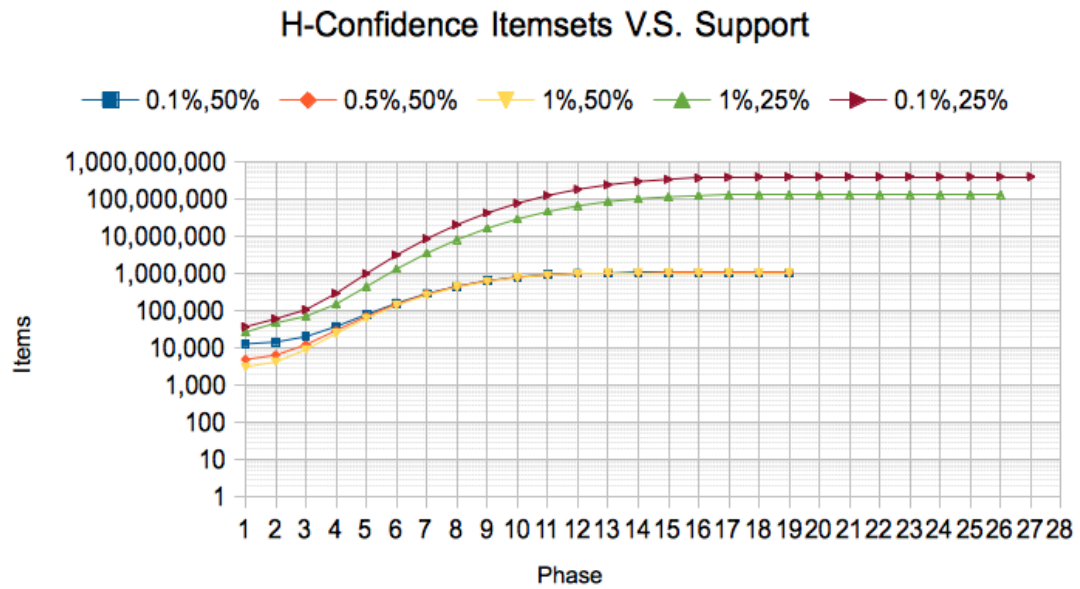


Figure 5.4: Frequent Itemsets by phase at various support and h-confidence levels

5.4 Chapter Summary

In this chapter we carefully detail our experimental setup in an attempt to make our results more easily reproducible. Comparing distributed algorithm with serial ones is difficult as different performance trade offs are made to enable parallelization. Additionally, Hadoop is a Java technology so does not lend itself well to direct comparison with existing C/C++ implementations.

We demonstrate how our approach can outperform the state-of-the-art algorithms with respect to running time. At 5% our algorithm beat, what we believe to be, the fastest known results by almost two orders of magnitude. Our attempt at a 1% support level exhausted the resources on our cluster during the 6th phase but there is no reason to believe it would not complete on a larger cluster.

To dive deeper into the dataset we apply a h-confidence based pruning strategy to compute highly correlated itemsets at support levels as low as 0.10%. We discovered that the number of frequent itemsets at each h-confidence level do not differ much with regard to changed in support. This may just be a interesting property of the webdocs dataset but we are excited to run more experiments to see if this result generalizes to other datasets. At low support levels most of the frequent itemsets generated are not even moderately correlated. We believe metrics such as h-confidence will become necessary for generating meaningful association rules at lower support levels.

Chapter 6

Conclusions

Frequent Itemset Mining has been at the forefront of the field of datamining for the past twenty years. Over this time many papers have been published describing various techniques for distributing this task. Despite all this research no one has produced an algorithm capable of achieving either of the two basic goals of distributed computing: reducing the overall running time of the algorithm or performing computations which are infeasible or fail altogether on a single machine.

We introduced a number of modifications to the classic Apriori algorithm to allow it to run efficiently in a distributed environment. By doing so we show that it can scale linearly with respect to the number of compute resources available as well as generate results at support levels much lower than any published results. Through clever partitioning strategies we ensure that frequent itemsets can be generated without the need for intermediary communication.

Our focus in this paper was to provide a robust framework for distributed Frequent Itemset Mining. We believe there is still considerable room for improvements in the areas of, data compression, candidate generation and counting, as well as job scheduling and data locality. We intend to build on our framework to incorporate some of the optimization tricks introduced in other implementations.

During development we applied our algorithm to many smaller datasets, in addition to Webdocs, and believe our techniques can be applied to any transactional dataset.

Our experimental results support our believe that support levels of 1% and lower are now possible and we are excited to run our experiments on a larger cluster to achieve these results.

The main result of this thesis is that Frequent Itemset Mining can now be distributed across the cheap commodity clusters that have become ubiquitous in modern computing. These abilities came with the additional benefits of minimal communication overhead and tight control over memory usage, CPU load, and partitioning. These techniques make it possible to mine at lower support levels than ever before considered and even apply additional heuristics to target just the highly correlated itemsets.

Bibliography

- [1] Disco. <http://discoproject.org/>.
- [2] Hadoop. <http://hadoop.apache.org/>.
- [3] Gregory Buehrer, Srinivasan Parthasarathy, and Amol Ghoting. Out-of-core frequent pattern mining on a commodity pc. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06*, pages 86–95, New York, NY, USA, 2006. ACM.
- [4] Gregory Buehrer, Srinivasan Parthasarathy, Shirish Tatikonda, Tahsin Kurc, and Joel Saltz. Toward terabyte pattern mining: an architecture-conscious solution. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–12. ACM, 2007.
- [5] Dehao Chen, Chunrong Lai, Wei Hu, Wenguang Chen, Yimin Zhang, and Weimin Zheng. Tree partition based parallel frequent pattern mining on shared memory systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, April 2006.
- [6] Min Chen, Xuedong Gao, and HuiFei Li. An efficient parallel fp-growth algorithm. In *Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC '09. International Conference on*, pages 283–286, Oct 2009.
- [7] Sean Chester, Ian Sandler, and Alex Thomo. Scalable apriori-based frequent pattern discovery. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 1, pages 48–55. IEEE, 2009.
- [8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

- [9] Zahra Farzanyar and Nick Cercone. Efficient mining of frequent itemsets in social network data based on mapreduce framework. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '13, pages 1183–1188, New York, NY, USA, 2013. ACM.
- [10] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, September 1960.
- [11] 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 *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 577–588, 2005.
- [12] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.
- [13] Liusheng Huang, Huaping Chen, Xun Wang, and Guoliang Chen. A fast algorithm for mining association rules. *Journal of Computer Science and Technology*, 15(6):619–624, 2000.
- [14] IBM. What is big data?, June 2004.
- [15] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.
- [16] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Webdocs: a real-life huge transactional dataset. In *FIMI*, 2004.
- [17] S. Naulaerts, P. Meysman, W. Bittremieux, T. N. Vu, W. Vanden Berghe, B. Goethals, and K. Laukens. A primer to frequent itemset mining for bioinformatics. *Brief. Bioinformatics*, Oct 2013.
- [18] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. *SIGMOD Rec.*, 24(2):175–186, May 1995.
- [19] P. Purdom, D. Van Gucht, and D. Groth. Average-case performance of the apriori algorithm. *SIAM Journal on Computing*, 33(5):1223–1260, 2004.
- [20] Ian Sandler and Alex Thomo. *Mining Frequent Highly-Correlated Item-Pairs at Very Low Support Levels*. PhD thesis, University of Victoria, 2011.

- [21] Ian Sandler and Alex Thomo. Large-scale mining of co-occurrences: Challenges and solutions. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012 Seventh International Conference on*, pages 66–73. IEEE, 2012.
- [22] Hui Xiong, Pang-Ning Tan, and Vipin Kumar. Hyperclique pattern discovery. *Data Mining and Knowledge Discovery*, 13(2):219–242, 2006.
- [23] Othman Yahya, Osman Hegazy, and Ehab Ezat. An efficient implementation of apriori algorithm based on hadoop-mapreduce model. 2012.
- [24] Zijian Zheng, Ron Kohavi, and Llew Mason. Real world performance of association rule algorithms. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 401–406, New York, NY, USA, 2001. ACM.
- [25] Le Zhou, Zhiyong Zhong, Jin Chang, Junjie Li, J.Z. Huang, and Shengzhong Feng. Balanced parallel fp-growth with mapreduce. In *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*, pages 243–246, Nov 2010.
- [26] Le Zhou, Zhiyong Zhong, Jin Chang, Junjie Li, J.Z. Huang, and Shengzhong Feng. Balanced parallel fp-growth with mapreduce. In *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*, pages 243–246, Nov 2010.