Mining Frequent Highly-Correlated Item-Pairs at Very Low Support Levels

Ian Sandler, Alex Thomo,
University of Victoria, British Columbia, Canada
iansandl@cs.uvic.ca thomo@cs.uvic.ca

Abstract

The ability to extract frequent pairs from a set of transactions is one of the fundamental building blocks of data mining. When the number of items in a given transaction is relatively small the problem is trivial. Even when dealing with millions of transactions it is still trivial if the number of unique items in the transaction set is small. The problem becomes much more challenging when we deal with millions of transactions, each containing hundreds of items that are part of a set of millions of potential items. Especially when we are looking for highly correlated results at extremely low support levels.

For 25 years the Direct Hashing and Pruning Park Chen Yu (PCY) algorithm has been the principal technique used when there are billions of potential pairs that need to be counted. In this paper we propose a new approach that allows us to take full advantage of both multi-core and multi-CPU availability which works in cases where PCY fails, with excellent performance scaling that continues even when the number of processors, unique items and items per transaction are at their highest.

We believe that our approach has much broader applicability in the field of co-occurrence counting, and can be used to generate much more interesting results when mining very large data sets.

1. Introduction

1.1 Overview Of The Problem

An effective way to illustrate our counting problem is to consider a basket of products being purchased at a supermarket checkout counter. As they are scanned the products (items) are recorded in a unique transaction for that specific customer. Each supermarket stocks tens of thousands of unique items, each of which has a unique numeric SKU. Each supermarket chain generates thousands of these transactions per day for each busy store.

These transactions can be mined to determine which pairs of items are likely to be purchased together both at that store, or any other store in the chain. Some pairings such as bread and butter happen so often that they are obvious. Others such as caviar and crackers occur much less frequently, but when they do occur they are highly correlated, making them much more interesting. It is these cases that we are trying to discover.

Generating the pairs of items from a transaction is extremely fast. Doing so billions of times and counting those pairs takes only a few minutes for a typical processor if the same few thousand unique pairs are incremented millions of times. The reason why this problem sometimes takes days to solve is because it requires a huge amount of effort to access and update the counters once they no longer conveniently fit into memory.

To illustrate the problems that arise, and to make the discussion very concrete, we selected the Webdocs data set from FIMI 04. This dataset is derived from real-world data, and is reasonably representative of the sort of data that needs to be mined. We chose it primarily because many published papers on frequent item mining have used this data set, and we wanted our results to be comparable.

Papers on efficiently mining frequent patterns such as [7], [8] and [9] imply that this dataset is no longer the challenge it was when it was first used. But in all cases that we looked at, the support level was set so high (7.5% or more) that it meant that very few pairs were actually generated. Webdocs contains 2.5 million transactions, 5.25 million unique items, and many items in the longest transaction. This is a lot of information.

Mining with a 10% support level means we are only interested in those items that appear at least 250,000 times in the data. Using this support level reveals that only 122 unique items are frequent enough to be used for counting pairs. (See Table 2). And none of the frequent pairs that can be generated from these 122 items are even moderately correlated. (See Table 13).

We are interested in doing much better. Webdocs is an ideal vehicle for demonstrating the algorithms and

techniques that are needed to extract highly correlated information from this type of raw data. It has the following characteristics:

Number of transactions	2,482,485
Number of unique items	5,267,656
Maximum items/transaction	281

Table* 1 - *Webdocs Statistics

Support	Support %	Surviving k=1
250,000	10.0	122
25,000	1.0	2,047
2,500	0.1	9,919
250	0.01	47,286
25	0.001	223,282

Table* 2 – *Surviving k*=1 *Candidates

For example, as shown in Table 2, when the support level is set to 0.01%, we need to examine all items that appear at least 250 times in the data set. In this case the number of unique items that exceed this support level is 47,286.

We set ourselves the task of using a support level of 0.001%. This increases the number of surviving frequent items to 223,282. This support level allows us to discover the highly correlated pairs that are missed when the support level is set at a much higher level. It also guarantees that we will hit the performance bottlenecks we are interested in exploring.

1.2 Our Contributions

We have analyzed the underlying performance characteristics and performance bottlenecks associated with this problem and devised a pair of techniques that make it easy to discover highly correlated pairs that only occur a few times. Our techniques work even when PCY fails and Map-Reduce falters. Furthermore we are able to generate these results quickly with good scalability. We believe that these techniques have a wide applicability to the field of data mining.

During this analysis we discovered that much of the information in the literature that talks about mining very large datasets does not adequately clarify the relationships between raw data size, support, relevance and the generated output. As stated earlier, none of the frequent pairs mined from this dataset at a 10% support level are even moderately correlated.

Our techniques are completely scalable. They work equally well with single computers, multi-core computers and networked computers. They also deliver results in time frames that can be predicted with a reasonable degree of accuracy.

1.2 Organization

The rest of the paper is as follows: Section 2 presents the preliminaries needed to work on the problem. Section 3 discusses the challenges faced by different techniques at very low support levels. Section 4 presents our solutions including an efficient algorithm for multi-core mining at a support level of 0.001%. Section 5 analyzes the results we obtained and attempts to draw useful conclusions. Section 6 documents the equipment used to generate our results. Section 7 suggests future research directions.

2 Preliminaries

Problem definition: Given a support level s, determine all the interesting pairs that occur in at least s transactions. A given pair (i, j) is considered to be interesting if the support for that pair is a reasonable percentage of the support for either i or j.

2.1 The Apriori Principle

Each transaction of n items generates approximately $n^2/2$ possible pairs. This means that for large data sets we can easily end up with hundreds of billions of pairs that need to be examined. Almost all of which are uninteresting.

We can use the Apriori [4] principle to eliminate as many items as possible from the input data before starting to count pairs. Briefly put a pair will only be frequent if both its constituent items are also frequent. This allows us to eliminate all pairs where one or more of the constituent items are not frequent.

If the support level is set too high very few items survive, and all the interesting ones are filtered out. If it is set too low too many pairs will be generated, making the problem unmanageable. Using our market basket analogy, we are interested in discovering relationships like caviar/crackers. To do so we have to mine the data at a support level that is no higher than the total number of times that caviar was purchased. In many real-world cases this is lower than 1%.

2.2 High Confidence Pairs

Our end objective is making it easy to find the high-confidence pairs like caviar/crackers that only exist at low support levels. This means that we need to mine with a suitably low support level and then eliminate

almost all of the results, rather than just generating those few results that we can probably guess if we are familiar with the information encoded in the data we are analyzing.

Working with large datasets and low support levels produces enormous numbers of results. When we used a support level of 0.001% on Webdocs we computed (in Table 4) that we had to generate and count more than 24 billion pairs to discover that there are more than 950 million non-zero pairs; with more than 50 million of those pairs being frequent.

For our results to be usable we need to distinguish which of these 50 million pairs are interesting and eliminate the rest. Eliminating those results that have a confidence of less than 95% reduces this 50 million to about 90K highly correlated pairs.

2.3 The Park Chen Yu (PCY) Algorithm

PCY [1] is the most common method used to compute frequent pairs from very large datasets. It was first proposed in 1995, and is still the method of choice appearing in current textbooks [2].

The item pairs are hashed to (counting) buckets that are each represented by a single counter (say of 32 bits). A given counter serves several item-pairs, namely all those that hash to that particular bucket. In the end, such a counter will hold the sum of frequency counts of all the item pairs that hashed to this bucket. The set of all counters is assumed to fit in main memory.

Once all the pairs have been processed, each counter is represented as a single bit that is true if the counter is higher than the specified support level. This shrinks the memory required to represent this information by a factor of 32. A subsequent pass through the data only counts those pairs whose hash bit is set.

This algorithm works really well when most of the counters are still below the support level once all the pairs have been counted. For those cases where too many pairs hash to the same bucket causing a lot of false positives, this algorithm uses a second and possibly a third pass with a different hash function to possibly eliminate false positives. Each pass only reduces the amount of memory available for subsequent processing by 3% (one bit instead of a 32 bit counter), so three passes still leave over 90% of the available memory for counting those pairs that have all three hash bits set.

This algorithm fails when most of the counters exceed the support level no matter what hash functions are used. We argue that this is exactly case for Webdocs (see Section 3.5).

2.4 The Google Map-Reduce Framework

Map-Reduce [3] is a framework developed by Google for working on computing problems that can be solved by distributing subsets of that problem to a large number of clustered computers and then combining the results from these subsets. The number of computers (or nodes) in a cluster varies greatly, ranging from tens to tens of thousands. It can be readily applied to this type of problem because the individual transactions are independent of each other.

There are two main steps required to solve a problem using the Map-Reduce framework. Map processes each take a subset of the data (in our case a unique subset of the transactions) and process that data outputting key-value (KV) pairs. In our case a given map process would take each individual transaction and generate all the potential KV pairs that can be constructed from that transaction. The key of each KV would be the item pair (such as bread/butter or beer/diapers) and the value would be a 1.

Once the map processes have all finished, the Map-Reduce framework groups all of these generated KVs and delivers them to a set of reduce processes that generate the final results. In this case the reduce processes sum all the output KVs that have the same key (bread/butter) and outputs a single KV as bread/butter:1234 (assuming that the key occurred 1234 times).

Map-Reduce uses a lot of hidden functionality. Data has to be stored and partitioned. The output from the map processes has to be grouped and delivered to the reduce processes. The entire process has to be managed to deliver all the results in a timely manner.

Despite looking simple and attractive Map-Reduce does not work well for this problem because it unnecessarily generates hundreds of gigabytes of intermediary data.

This said, our techniques can in some sense be considered as Map-Reduce approaches. However they do not create any significant amount of intermediary data.

2.5 Optimizing Performance by Eliminating Bottlenecks

Disk IO is well understood to be a problem for data mining algorithms. Sequential disk IO is extremely fast. It is often faster than random main memory access (cf. [13]). In some circumstances it can even be faster than writing randomly to memory. On the other hand a single random disk IO usually takes about 10

milliseconds to complete, during which time the process often has to stall. During that same amount of time the computer's CPU can perform one random disk IO it can perform about 10 million instructions, so minimizing random disk IO must be a very successful strategy no matter how expensive that strategy is in terms of cpu cycles.

Disk IO happens for two reasons. The first is when the application explicitly reads or writes data. The other case is when the program's memory usage exceeds the amount of real memory available on that computer causing it (or some other process) to be swapped out to disk to free up memory.

Explicit reads and writes are easy to recognize and can often be minimized or eliminated by careful coding. The implicit reads and writes caused by paging are much harder to handle. They are almost always the difference between algorithms running at the expected speed or thousands of times slower than would otherwise be expected.

Both of our counting techniques eliminate all explicit and implicit random disk IO associated with the counting process, making run times both scalable and predictable.

Another performance issue is the effect of CPU caches on the way that a computation performs. Storing counters in a hash-tree structure (as documented in various text books including [12]) causes the program to jump around in memory as it follows the tree structure to reach the appropriate counter. It also causes new counters to be added to the tree at random intervals and on a random basis. This is not good for memory cache utilization. The difference between efficient and inefficient cache utilization is typically about one order of magnitude. Practical confirmation of this estimate can be found in [8] that shows a performance improvement by a factor of 5:1.

3. The Challenge

In order to eliminate the implicit random disk IO associated with counting we first need to know how much memory will be needed to store the required counters. Table 2 shows that when the support level is set to 0.01% for Webdocs, the number of unique items that exceed this support level is 47,286. Table 3 shows the number of potential pairs exceeds 1 billion; and the minimum amount of memory needed to count these pairs would be 2.25GB if we were to use exactly 4 bytes per cell to count these pairs. When the support level is set to 0.001%, this increases the number of frequent items to 223,282 and the memory requirement to 50GB.

3.2 Apriori Counting Optimizations

The first and easiest way to reduce the memory requirement is to rely on the Apriori principle and perform an initial pass through the data to eliminate all items whose count fails to meet a predefined minimum support level. Comparing the different support levels against the bottom row of Table 3 where no Apriori reduction is used clearly demonstrates its effectiveness.

After using Apriori to prune the k=1 candidates, a second inexpensive strategy is to renumber the surviving items such that the most frequent item is renumbered to 1, the next most frequent to 2 and so on. This makes it possible for the data to be written back to disk without the infrequent items. It also allows the counts to be stored in a much more compact two-dimensional array.

3.1 Counting in A Memory Resident Table

In order to make the following discussion concrete, we need to document that the computers we used to produce the results each had 4 Xeon cores sharing 6GB of memory. This gave us three possible processing configurations: 4 separate processes per computer with each process using 1.5GB memory, 2 processes with each one using 3GB memory, and a single process using all available memory. In each of these cases the entire memory space could not be used for counters. Allowances had to be made for the operating system, disk buffering, program code and variables etc. Experimentally we were able to determine that this reduced the amount of memory we could use for storing counters to 1.25 GB per core.

We calculated Table 3 so we could see how much memory is needed to count pairs in memory, assuming that we use Apriori to eliminate infrequent items, and then use 4 bytes of memory per counter for each possible k=2 frequent pairs. The table assumes that the counters are stored in a triangular array of size $[F_1]^2/2$, where $[F_1]$ is the number of surviving singletons.

Support	Support %	Surviving k=1	Possible k=2	GB
250,000	10.0	122	7,442	~0
25,000	1.0	2,047	2,095,105	0.01
2,500	0.1	9,919	49,193,281	0.79
250	0.01	47,286	1,117,982,898	2.25
25	0.001	223,282	24,927,425,762	50
1		5,267,656	$\sim 13.9*10^{12}$	

Table 3 – All Possible k=2 Candidates

If we want to use all 4 cores simultaneously we need to limit memory use per process to be 1.25GB to avoid implicit disk paging. This means that we can count all the possible ~50 million pairs for the 0.1% support level in memory because that only needs 0.2GB. We can also see that this method fails with a support level of 0.01% because 1.12 billion possible pairs require 2.25GB memory. It does work if we use only 2 processes. We can also see that using 2 byte counters instead of 4 byte counters would have just allowed us to once again use all 4 cores.

3.3 Counting in a Hash Tree

A different strategy has often been used for counting. Rather than creating an in-memory table that has one counter for each potential pair, the other possibility is to only store and increment each unique pair that is generated from the data set. This will use less counters because some combinations will not exist in the input data. It requires using some sort of hash-tree to store the results. This is the underling counting technique that has been used over the years by most data mining algorithms. It is very successful when the number of non-zero pairs that have to be counted will fit in memory.

With this approach we have an unpredictable situation. We have a fixed amount of memory for storing results and no real knowledge of how many unique pairs will produce at least one hit. So we have a process that runs quickly in the beginning, gradually slows as the tree grows, and then suddenly runs tens of thousands of times slower or aborts once the amount of available memory is exceeded.

The only thing we are certain of is that as the support count is lowered more unique pairs will need to be counted, and the chances of running out of memory increase. We also know that this technique will use the cache inefficiently because elements will be added to the tree at random.

With this data set and a 1% support level we determined experimentally that we generate 112,394 frequent unique pairs after applying the Apriori principle. Assuming that we use 16 bytes to store the pair (4 bytes per item) plus the counter (4 bytes) plus the required linkages (4 bytes), this gives us a memory requirement of 1.6 megabytes. For 0.01% this increases to approximately 7GB and for a 0.001% support level we need almost 16 gigabytes just to store our counters. We see that this method fails for a support level of 0.01% when even 6GB of memory is available.

These observations agree with the intuitive prediction that all the extra complexity will not help once the support level is sufficiently low and we are no longer working with a sparse set of non-zero pairs.

Supt %	Possible k=2	Generated Pairs	Non Zero k=2	Frequent k=2
10.0	7,442	$1.1*10^9$	7,381	729
1.0	$2.1*10^6$	$14.2*10^9$	$2*10^6$	112,394
0.1	$49.2*10^6$	$20.8*10^9$	$48.2*10^6$	$1.25*10^6$
0.01	$1.1*10^9$	$23.4*10^9$	$437.8*10^6$	$8.37*10^6$
0.001	$24.9*10^9$	$24.5*10^9$	$952.4*10^6$	$51.1*10^6$

Table 4 – Actual k=2 Frequent Pairs

3.4 Counting Using Map-Reduce

Map-Reduce offers a different way to count the generated pairs. It relies on a Map process outputting the generated pairs as a linear stream of KV pairs and then organizing that stream such that KV pairs that group together appear together to an associated Reduce process.

Map-Reduce eliminates the need to store the generated pairs in memory, removing that restriction no matter how low we set the support level. Instead of using memory based counters we generate approximately 15 bytes per KV pair that are sequentially (and very quickly) written to a file system. These pairs then need to be sorted and reduced.

When we are dealing with a support level of 10% we generate 16GB of KV data and this technique works nicely. Unfortunately, for a 0.001% support level we emit about 350GB of KV data. This is far too large a number. We start with only 1.25GB of raw data and are only going to produce a few kilobytes of relevant results. The intermediate disk space requirement (and its associated disk IO) is very excessive.

3.5 Why the PCY Algorithm Can Fail with This Dataset at a 0.001% Support Level

The PCY algorithm has problems once support levels are so low that the number of unique potential pairs that need to be counted causes many of the counters to exceed the specified support level.

Also, observe that once someone uses the PCY idea, holding the exact counters of each item pair has to be done by using a hash-tree. Using a matrix of counts would not make sense, as it would defeat the purpose of the extra PCY filtering. Therefore, PCY suffers from the same problems as the other approaches using hash-trees for keeping the item pair counts.

Given our particular dataset it is apparent that for a support level of 0.001%, PCY has problems when we only have 1.25GB of memory. Assuming 4 bytes per

counter, 1.25GB memory can store up to 312.5 million counters. According to Table 4 we have 952.4 million unique non-zero pairs to count in these 312.5 million counters, so every counter is very likely to be set by at least three different pairs. At least 51 million of these counters will exceed the support level because 51.1 million unique pairs exceed this support level. Assuming the pairs, which PCY indicates have to be counted, are subsequently held in a hash tree, (and that each of these hash tree counters requires 16 bytes) we can see that this can fail even with multiple hash passes.

We could guarantee that PCY works by using a single process that uses all 6GB of memory, but that does not change the underlying observation.

4. Our Techniques

From a performance viewpoint the problem we are trying to solve can best be summarized as:

1. Generate results when the number of non-zero pairs that need to be counted will not fit in memory.
2. Generate these results in a reasonable predictable time frame.
3. Make effective use of available multi-CPU multi-core hardware to generate the results, and have the time taken to produce these results decrease in a near linear manner as more processors are added.

We have developed two distinct techniques that will achieve these goals. One uses all available memory on a single computer to generate the results. The other efficiently shares that memory between multiple cores. Both share many common features.

4.1 Pre-Processing Steps

Webdocs is a 1.5GB file of 2.5 million transactions that contain our 5.25 million unique items. The first step is to count the data discovering the number of occurrences for each unique item. This pass only takes a couple of minutes, and uses relatively little memory to compute these occurrences.

Once we have counted the occurrences it is easy to sort the items and substitute their frequency (FID) for their item numbers, such that the item with the most occurrences becomes FID 1, the next most frequent becomes FID 2 and so on. (Ties are broken arbitrarily). Infrequent items are eliminated at the same time. This process happens completely in memory and only takes a few seconds.

Reading the file a second time and writing it out again with the substituted FIDs (trimming infrequent items and dropping empty transactions) takes only a couple of minutes. Our total pre-processing time is less than 10 minutes.

Sections 4.2 and 4.4 illustrate our techniques. To make our explanations easier to understand we have chosen to document our techniques using square counting arrays (as opposed to triangular).

4.2 Single Core 0.01% Support Using All 6 Gigabytes of Memory

Our computer has 6GB of memory. Some of that memory is needed for the operating system and for other tasks. Assuming that we use 32-bit numbers as counters it is safe to assume we have sufficient memory space available (5GB) to store approximately 1.25 billion counters without needing to do any disk I/O. This means that we can safely create a two-dimensional square array whose side is 33,000.

Counting now proceeds as follows. We generate all possible pairs for every row, which as we described contains only frequent items. For each generated pair we examine both FIDs. If both are less than 33,000 the pair is counted in our memory resident table (See Figure 6). Otherwise the pair is emitted as a KV pair with a value of 1. Once the process finishes, all the frequent pairs found in the memory table are emitted as KV pairs with their values being their counts rather than 1s. A final pass reads in all the emitted pairs, aggregates them “a la Map-Reduce”, and eliminates the infrequent ones.

A support level of 0.1% produces 9,919 frequent k=1 FIDs. The potential pairs generated will fit in a square table of size 9,919 x 9,919. We have sufficient memory to accommodate a table that has a side of 33,000. So for this support level the entire computation proceeds in memory making it extremely fast. And no further optimization is needed.

k=1 items	47,286
Total number of KV pairs generated from webdocs data	23,382,204,891
Pairs counted in memory	22,938,692,793
Pairs emitted during the process because they could not be counted in memory	443,512,098
Frequent pairs emitted at the end of the calculation	307,401,070
Total pairs emitted by process	750,913,168
Percentage pairs emitted because they did not fit in memory	1.9%

Table 5 – Pairs generated for 0.01% support level

A support level of 0.01% produces 47,286 k=1 FIDS. We once again count pairs that have both FIDS

less than 33,000 in memory. This time we emit KV pairs when either FID is over 33,000. Because we are using frequency based item IDs most of the generated KV pairs are counted in memory. We only need to emit 443 million instead of 23,382 billion KV pairs. This reduces the amount of data emitted from 350 gigabytes (which is the amount of data emitted by a simple Map-Reduce) down to 6.5 gigabytes, allowing us to generate our results in under 20 minutes.

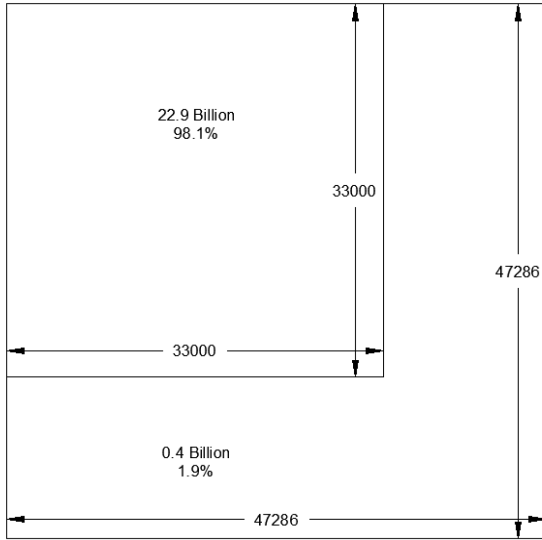

Figure 6 – Distribution of pairs for 0.01% support. 22.9 Billion KV-pairs are counted in the in-memory 33,000×33,000 matrix of counts. 0.4 Billion KV-pairs are emitted.

4.3 Single Core, 0.001% Support Using All 6 Gigabytes of Memory

The above technique does not work as well when we are mining with a 0.001% support level. In this case Table 7 shows that we are dealing with a square whose side is 223,282.

k=1 items	223,282
Total number of KV pairs generated from webdocs data	24,481,791,476
Pairs counted in memory	22,938,692,793
Pairs emitted during the process because they could not be counted in memory	1,543,098,683
Frequent pairs emitted at the end of the calculation	307,401,070
Total pairs emitted by process	1,850,499,753
Percentage pairs emitted because they did not fit in memory	6.3%

Table 7 – Pairs generated for 0.001% support level

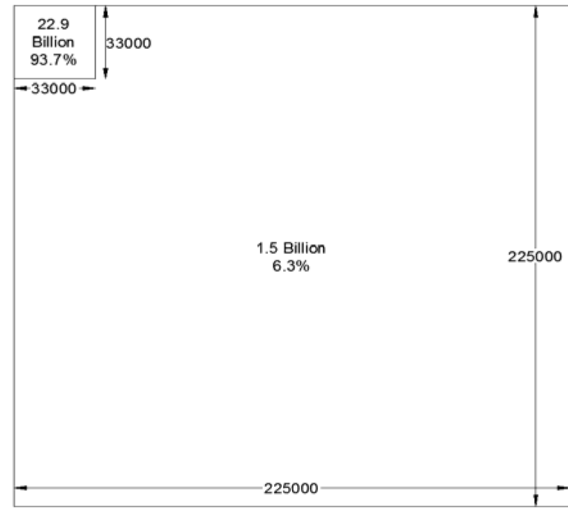

Figure 8 – Distribution of pairs for 0.001% support. 22.9 Billion[1] KV-pairs are counted in the in-memory 33,000×33,000 matrix of counts. 1.5 Billion KV-pairs are emitted.

Only emitting 6.3% of the KV pairs sounds good until one realizes that this still causes 1.5 billion pairs to be emitted occupying over 20 gigabytes of disk space. Although this is 18 times better than a simple Map-Reduce (which would generate over 360 gigabytes of data), we can do much better.

4.4 Multi Core 0.001% Support Using 100 Cores Belonging to 25 CPUs

Our second technique uses multiple processes to generate the results. In our illustrative example we use 100 processes to handle a support level of 0.001%. Because each computer blade has 4 cores and 6 gigabytes of memory we know that each process can safely use 1 gigabyte of memory without causing any disk IO. (The rest is reserved for the operating system, other variables and disk buffers). This means that each process can safely hold 256 million 32-bit counters in memory. We know that we are dealing with 223,282 *k*=1 FIDs. This means that each process can fit into memory a range of 1,150 values for the first FID of the pair when combined with all the values for the second FID of the pair. This 1,150 rows represents 0.5% of the total number of rows. This means that we can generate all our results using 200 processes as shown in Table 9.

[1]This is the same number as for Figure 6, and it is not a coincidence. The frequency rank of items depends on the dataset, not the support level used.

Rows	←Columns 1 thru 230,000→	Process
1-1150	…..256 million counters…..	1
1151-2300	…..256 million counters…..	2
2301-3450	…..256 million counters…..	3
	…..256 million counters…..	…
226551-227700	…..256 million counters…..	198
222701-228850	…..256 million counters…..	199
228851-230000	…..256 million counters…..	200

Table* 9 – *Counting When 200 Cores Are Available

We do not want to use FIDs with this technique because the process that handled the first few FID numbers (Process 1 in Table 9) would take a lot longer to complete. So we need to replace FID numbers with something more random. Rather than sorting the items by the number of hits we simply assigned 1 to the first frequent item, 2 to the next frequent item and so on. These sequential items (SID)s distributed the processing adequately as can be seen in Table 11.

If we only have 100 cores available we have to create a script that processes two different sets of rows one after the other. Doing things this way ensures that the second set cannot start running until the first one completes, and therefore we won't accidentally set up a situation where the computer has to page memory.

Rows	←Columns 1 thru 230,000→	Process
1-1150	…..256 million counters…..	1
1151-2300	…..256 million counters…..	2
2301-3450	…..256 million counters…..	3
	…..256 million counters…..	…
226551-227700	…..256 million counters…..	3
222701-228850	…..256 million counters…..	2
228851-230000	…..256 million counters…..	1

***Table* 10 – Counting When 100 Cores Are Available**

Passing through the entire Webdocs data set once with a Unix wc command takes about 60 seconds. Running our java program that looks at all the transactions, generates all the pairs, and counts the appropriate ones in memory takes an average of 4.5 times as long, with the fastest process taking 211 seconds and the slowest process taking 419 seconds. The final set of results from the final process completes less than 12 minutes after the mining starts.

k=1 items	223,282
Total pairs generated from webdocs data	24,481,791,476
Frequent Pairs	51,053,891
Time taken by fastest pass	211 secs
Time taken by slowest pass	419 secs

Table* 11 – *Pairs generated for 0.001% support level

Because of the way we are distributing the data we can be certain that everything will scale reasonably linearly. For example, using 25 cores belonging to 7 blades, with each process making 4 passes takes 25 minutes. Also, as expected, we didn't see any significant differences in performance when we used 25 cores on 25 different blades.

Our processing techniques are completely scalable. They can already handle much larger problems than mining Webdocs. The key being that we know how much memory is available and we use this information to avoid swapping.

4.5 Filtering Out Unreasonable Results

The 1.25 million frequent pairs generated at a support level of 0.1% are too many to be useful. To resolve this problem we need to filter out all the pairs where the relationship appears to be accidental. Using confidence meets this need. It only requires a minor change to the code, and did not affect the amount of time required to generate results.

Table 12 shows that using this technique reduced this 1.25 million to 1,511 (approx 1000:1) for the case where we assume that a frequent pair is interesting only if either one of its items appears 85% of the time in the pairs.

Support	Frequent Pairs	25% Confident Pairs	50% Confident Pairs	80% Confident Pairs
10%	729	0	0	0
1%	112,394	3,265	1,351	353
0.1%	$1.25*10^6$	11,457	3,732	1,680
0.01%	$8.37*10^6$	26,032	13,110	8,949
0.001%	$51.1*10^6$	304,658	173,788	118,854

Support	Frequent Pairs	85% Confident Pairs	90% Confident Pairs	95% Confident Pairs
10%	729	0	0	0
1%	112,394	301	118	118
0.1%	$1.25*10^6$	1,511	1,170	1,117
0.01%	$8.37*10^6$	8,528	7,901	7,600
0.001%	$51.1*10^6$	99,219	94,686	90,345

Table* 12 – *Confident Pairs

5. Interpretation of The Experimental Results

5.1 Support Produces Too Much Noise

It is hard to say what is a reasonable support level before the results have been produced. It depends on the data being analyzed and the information we are attempting to glean from that data. With our techniques it is possible to produce results really quickly. This makes it feasible to mine for very low support levels. Once we have our frequent pairs we can then obtain the numbers of pairs for multiple confidence levels with a few seconds worth of processing. As can be seen in Table 12 above, there are only 304,658 pairs that have even a 25% confidence level.

When we use what we think is a reasonable support level of 0.1% for this data set we generated 1.25 million frequent pairs. This was reduced to 1,511 pairs once we limited our results to those pairs with a confidence level of at least 85%.

This type of filtering ensures that the reported results are valuable. It is equally essential when mining for longer sets of frequent items. We recommend this use of confidence (or better still h-confidence [11]) at a moderate level (70-90%) for this purpose. We understand that doing so will dramatically change the way we view some of the existing data mining code. But we also believe that this is necessary if we want to evaluate how well these algorithms produce meaningful results.

5.2 Using Map-Reduce Efficiently Is Not Always Recommendable

Map-Reduce is not a magic solution to all multi-core problems. We have demonstrated that with a bit of thought we could reduce the amount of data emitted by the Map processes by a factor of 20-30. We also showed that by carefully choosing the exact way the memory is used we can almost totally eliminate the Reduce step.

It also turns out that language libraries can cause lots implicit disk IO as well as burning lots of extra cpu cycles, and that is something that needs more attention when using Map-Reduce for simple computations on large data sets. It is easy to lose the performance gains that Map-Reduce can deliver by simply making a wrong choice in programming language when that language causes unexpected paging.

5.4 Size Is Meaningless Without Support

This research makes it clear that the absolute size of the data set being evaluated is not a fair representation of the effort required to mine that data set. For example, the size of the Webdocs data set is approximately 1.25 gigabytes, but most of that size is noise at high support levels.

It is more accurate to state that it is 250 megabytes when the support level is 10% and 600 megabytes when the support level is 5%. We suggest that using the effective size of the data set (obtained after stripping out all the k=1 infrequent items) is a much better measure when comparing results. This number is still inaccurate because the number of potential candidates generated depends on the square of the number of items appearing in each transaction, but it is nonetheless a step forward by making it easier to compare results among different datasets.

5.5 Other Interesting Observations

One of the supposed weak points of the Apriori algorithm is that it requires multiple passes through the data. The timings generated by this paper show that when mining for low support levels this is irrelevant. The time to pass sequentially through the data is insignificant when compared with the time required to generate and count the candidates.

It would appear that when the Webdocs dataset is mined at 10% or higher support levels none of the k=2 pairs have even a 25% confidence level. This suggests that mining this particular dataset at this particular support level is at best an academic exercise.

6. Equipment And Software Platform Used

6.1 Hardware and Operating System

Summarizing, the results presented in this paper were generated using a 25 blades of a 42 blade IBM cluster, where each blade has 6 gigabytes memory and a quad core Intel Xeon processor rated at 2.33 gigahertz. Each blade has 4 megabytes of L2 cache that is shared by all 4 cores. To control memory usage we explicitly limited the amount of memory available to our computations, causing these programs to crash rather than page if we used too much memory. We used Linux as our operating system. This allowed us to use top and vmstat so that we could carefully monitor

memory usage to ensure that everything worked properly.

6.2 Choice of Programming Language

Early attempts at coding our algorithms failed to produce the expected results. We were using Python as our programming language and discovered that we were measuring that language's inability to efficiently store values in a large linear static array, not the performance of our algorithms. In fact almost the whole elapsed time was spent on the single Python instruction that referenced the correct counter. We attempted to use different libraries to minimize this overhead but were eventually forced to give up because the results were still orders of magnitude slower than our expectations.

We also experimented with both Disco [5] and Hadoop [6], but once again had performance and reliability problems. We eventually abandoned both these options in favor of a simple multi-process manager that we wrote ourselves. Normally this would have been a poor decision, but because none of our processes took more than a few minutes to complete handling restarts was unnecessary. And because we severely limited the size of the intermediate outputs we didn't need any of the excellent file management that these tools offer.

We eventually recoded our algorithms using a mixture of Jbase (a commercial product ideally suited to handle the scripting and reporting we needed) and Java for the computationally intensive tasks. In this environment our code behaved as predicted. In fact the Java run time performance significantly exceeded our expectations, delivering results that were within an order of magnitude of well-written C code.

7. Future Research Directions

The same techniques outlined in this paper can be used for finding longer sets of associated items. Doing so requires solving additional challenges. One problem not addressed in this paper is the ability to quickly generate all the potential candidates when dealing with longer chains of frequent items. This is addressed in our paper [10], which demonstrates a technique to quickly perform this task. The algorithm outlined in that paper is designed so that it can run in parallel on multiple cores with little or no degradation.

The other major problem mining data at these low support levels is getting rid of the noise caused by frequent items generating uninteresting results. Using confidence greatly reduced the number of pairs generated. We believe that using h-confidence [11] will do an excellent job of removing the noise when $k>2$ without losing any interesting results.

We confidently expect that we will be able to combine the techniques developed in this paper with these two additions, and that we will be able to demonstrate that by doing so we can mine very large datasets and extract useful information much more efficiently than current techniques.

We also believe that our counting techniques can be used to improve performance whenever PCY fails or any Map-Reduce solution generates so many recurring output pairs that they overwhelm the hardware available to the Map-Reduce mechanism. We look forward to using them to address other currently intractable problems.

8. References

[1] Jong Soo Park, Ming-Syan Chen, Philip S. Yu: An Effective Hash Based Algorithm for Mining Association Rules. SIGMOD Conference 1995: 175-186

[2] Molina G. H., Ullman D. J, Widom J. *Database Systems: The Complete Book.* 2nd Ed. Prentice Hall. 2009, Pages 1105-1109

[3] Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150

[4] Rakesh Agrawal, Tomasz Imielinski, Arun N. Swami: Mining Association Rules between Sets of Items in Large Databases. SIGMOD Conference 1993: 207-216

[5] Disco. http://discoproject.org/

[6] Hadoop. http://hadoop.apache.org/

[7] Gregory Buehrer, Srinivasan Parthasarathy, Amol Ghoting: Out-of-core frequent pattern mining on a commodity PC. KDD 2006: 86-95

[8] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony D. Nguyen, Yen-Kuang Chen, Pradeep Dubey: Cache-conscious Frequent Pattern Mining on a Modern Processor. VLDB 2005: 577-588

[9] Gregory Buehrer, Srinivasan Parthasarathy, Shirish Tatikonda, Tahsin M. Kurç, Joel H. Saltz: Toward terabyte pattern mining: an architecture-conscious solution. PPOPP 2007: 2-12

[10] Sean Chester, Ian Sandler, Alex Thomo: Scalable APRIORI-Based Frequent Pattern Discovery. CSE (1) 2009: 48-55

[11] Hui Xiong, Pang-Ning Tan, Vipin Kumar: Hyperclique pattern discovery. Data Min. Knowl. Discov. 13(2): 219-242 (2006)

[12] Pang-Ning Tan, Michael Steinbach, Vipin Kumar: *Introduction to Data Mining* Addison-Wesley 2005

[13] Adam Jacobs: The pathologies of big data. Commun. ACM 52(8): 36-44 (2009)