Parallel Frequent Set Counting

D.B. Skillicorn
skill@cs.queensu.ca

December 1999
External Technical Report
ISSN-0836-0227-
1999-434

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared December 17, 1999

Abstract

Computing the frequent subsets of large multi-attribute data is both computation- and data-intensive. The standard parallel algorithms require multiple passes through the data. The cost of data access may easily outweigh any performance gained by parallelizing the computational part. We address three opportunities for performance improvement: using an approximate algorithm that requires only a single pass over the data; statically partitioning the attributes to provide rudimentary load- and storage-balancing; and using a probabilistic technique to avoid generating most of the lattice of subsets implied by each object's data. The computation required is only slightly larger than levelwise algorithms, but the amount of data access is much smaller.

Parallel Frequent Set Counting

D.B. Skillicorn
skill@cs.queensu.ca

Abstract: Computing the frequent subsets of large multi-attribute data is both computation- and data-intensive. The standard parallel algorithms require multiple passes through the data. The cost of data access may easily outweigh any performance gained by parallelizing the computational part. We address three opportunities for performance improvement: using an approximate algorithm that requires only a single pass over the data; statically partitioning the attributes to provide rudimentary load- and storage-balancing; and using a probabilistic technique to avoid generating most of the lattice of subsets implied by each object's data. The computation required is only slightly larger than levelwise algorithms, but the amount of data access is much smaller.

1 Problem Setting

The problem of finding frequent sets is a basic component of several standard data mining algorithms such as association rules [10] and episodic data analysis [8]. We assume that the data are given as a binary array whose n rows represent objects, and whose m columns represent attributes. A 1 in the array at position (i, j) denotes that object i has attribute j. In most practical problems, the array is sparse.

The problem is this: find those subsets of the attributes that occur together in rows more often than some given frequency, called the *support*. Such subsets are called the *frequent sets* in the data. Since a typical array may have billions of rows and hundreds or thousands of attributes, the problem is difficult because of the amount of computation required, the size and effort involved in examining the data, and the fact that even sets of modest size have large numbers of subsets. Because of the sparseness of the data, lower frequency bounds (supports) are typically less than 1% and the likelihood of a subset of size more than a fraction of the total number of attributes being frequent is small.

The standard algorithms for this problem exploit the insight that a set can only be frequent if all of its subsets are frequent. Accordingly, they first generate sets of size 1, make a pass through the array to determine which of these are frequent, and then use these to generate those sets of size 2 which could potentially be frequent. This process is repeated for sets of increasing

size. If the largest frequent set is of size l then this algorithm executes $l + 1$ phases and makes $l + 1$ passes through the array. Such algorithms have been called *levelwise*. In practice, it is these repeated accesses to the data that limit the performance of the algorithm. (In some applications, the large number of potentially frequent sets of small size (2 or 3) also creates a storage management performance bottleneck.)

Sequential levelwise algorithms for finding frequent sets have been well studied [1, 9], although performance increase of two orders of magnitude have been recently reported by clever strategies for representing and generating large candidates from smaller ones [7]. These algorithms are relatively straightforward to parallelize, although a number of different strategies have been used to reduce the extra communication that arises from parallelizing storage manipulation in the sequential case [5].

We show that approximate sampling algorithms can be adapted to compute most frequent objects, reducing the data access required to a single pass. However, it is also critical to exploit the levelwise property for otherwise the work of generating and considering subsets that a levelwise algorithm would never build could outweigh the performance gains from reduced access. We also show that there exists a simple static allocation scheme that can be used to avoid the storage problems caused by the fact that there are many small subsets of a set of substantial size, and they are inherently more likely to be frequent than larger subsets.

2 Approximate Frequent Set Counting

Gibbons and Matias [4] have designed two sequential approximate sampling algorithms that can be adapted for frequency counting. Their chief attraction is that they require only a single pass through the dataset, and hence are much faster than levelwise algorithms. They compute approximate (bounded error) frequencies.

Both algorithms assume a limited memory windows for accumulating samples. When counting frequencies, it is more important to ensure that all sufficiently frequent objects remain in the sample than to limit the window size. It is straightforward, however, to adapt their algorithms to achieve this behavior.

For the time being, let us consider a simpler problem than frequent set counting, in which the data is simply a list of object identifiers, and we wish to identify those objects that occur at least as frequently as the given support.

A *concise sample* is a uniformly random sample in which the probability

of occurrence of each object in the list is $1/\tau$. In the obvious way, we actually store such a sample by replacing multiple occurrences of the same object with its identifier and its multiplicity. However, in algorithm descriptions, each occurrence is treated as a unique object. A concise sample may be maintained incrementally using the following algorithm:

```
τ := 1
for all t in the dataset
    with probability 1/τ
        add t to the window (new entry or increment counter)
    if window overflows
        τ' := τ×factor
        for each (unique) s in the window
            remove it with prob τ/τ'
        τ := τ'
```

This algorithm maintains a uniformly random sample; and the costs of insertion and increasing the threshold can be amortized so that the expected time per insert is $\mathcal{O}(1)$.

Gibbons and Matias also define a slightly more accurate sample called a *counting sample*. Here, once an object is added to the window, any further occurrences of that object in the list are noticed (that is, its count is incremented). The incremental insertion algorithm is:

```
τ := 1
for all t in the dataset
    if t already in the window
        increment its count
    else
        add t to the window with prob 1/τ
    if window overflows
        τ' := τ×factor
        for each s in the window
            flip a biased coin, decrementing the count on each
            tails, until a head is flipped.
        τ := τ'
```

Notice that there is a lookup of the window for every element of the list. Thus this algorithm is more expensive than that for building a concise sample, but the expected time per insertion is still $\mathcal{O}(1)$.

The counts in a counting sample are underestimates of the actual frequency of occurrence, since some early occurrences have been missed. More accurate counts can be computed by adding a compensation factor ($\approx 0.418\tau - 1$) to each count.

For a counting sample, any object that occurs with frequency no less than τ is expected to be in the sample. In fact, the probability that an object which occurs f times will be in the sample is $1 - (1 - \frac{1}{\tau})^f$ [4].

There is a simple way to postprocess counting sample values to get concise sample values without any further access to the data.

These algorithms were designed to compute a sample assuming a fixed window size. For frequent object counting, we are less concerned with memory bounds, and more concerned with ensuring that all sufficiently frequent objects remain in the sample. Accordingly, both algorithms can be modified to increase the window size when increasing the threshold would cause the removal of objects that appear to be frequent so far. Here is the modified logic:

```
if window overflows
    if all window entries needed
        (i.e. relative freqs > α× support)
        window size := window size + increment
    else
        increase τ as before
```

The window size is increased only if the relative frequency of all of the elements in the window is large enough that they might be frequent. The choice of an appropriate α ($0 < \alpha \leq 1$) for this decision is arbitrary. It has no influence on the correctness of the algorithm; it only avoids removing data which is highly likely to be reinserted. Whereas τ is increased by a multiplicative factor, the window size is increased only by an additive factor.

3 Parallel Versions of Approximate Counting

We now turn to parallelizing approximate frequency counting. The difficulty is that, if the array is partitioned and each partition sent to a different processor, then the distribution of objects in one processor may be quite different from that in another.

There are theoretical reasons to expect that this is unlikely, provided that the partitioning is done randomly, rather than by simply partitioning the array

into p segments by rows. For example, the technique of *bagging* uses small samples from a dataset to generate predictors [2]. Small ensembles of such predictors commonly outperform a single predictor determined from the whole dataset. Frequent set counting can be regarded as a form of bagging in which each processor generates a predictor for the most frequent elements in the array.

It is straightforward to parallelize the computation of concise samples:

partition data randomly across p processors
forall p do
    $\tau_p := 1$
    for all t in the processor's partition of the data
        execute the sequential algorithm
total exchange of the values of τ_p
forall p do
    compute $max\tau :=$ maximum of the τ_p
    remove each sample in the window with prob $\tau / max\tau$
merge the windows into a single concise sample
remove all entries whose relative frequency,
$$f / \mid \text{total sample size} \mid,$$
    is less than the support

The random allocation of data to processors ensures that the union of the concise samples built by each processor, after their τ values have been normalized to the largest, is also a uniformly random sample.

It is less obvious that the same strategy can be used to parallelize the building of a counting sample. The problem is that a particular object may occur infrequently in one particular data partition, it may be culled from the window on that processor, and there is no reasonable way to account for its missing count in the global total over processors. The solution is to use concise samples to determine when and how to expand the window on each processor (preserving the uniform randomness of the sample), but use the results of a counting sample as the final frequencies (exploiting their greater accuracy). There are two ways to do this:

- Use both a concise and a counting frequency within the window. This increases the storage requirement for the window by 50%, but makes the program logic simple.

```
partition data randomly across p processors
forall p do
    τ_p := 1
    for all t in the processor's partition of the data
        if t already present in the sample then
            increment its count in the counting sample
            with probability 1/τ_p
                increment its count in the concise sample
        else
            with probability 1/τ_p
                add t to the concise sample
        if window overflows
            adjust as in the extended concise sample algorithm
total exchange of the values of τ_p
forall p do
    compute maxτ := maximum of the τ_p
    remove each sample in the window with prob τ/maxτ
    adjust each counting sample count by the compensation term
merge the windows into a single counting sample
remove all entries whose count < support × n
```

Figure 1: Pseudocode for a parallel counting sample algorithm.

- Use only a counting frequency but convert counts to concise sample frequencies when increasing the threshold and deciding which elements to evict from the window. This requires no extra storage, but requires extra computation (time proportional to the frequency of each entry).

Pseudocode for the first solution can be seen in Figure 1. It can be seen that the counting sample algorithm returns exactly the same objects in the sample as the parallel concise sample algorithm, but the frequencies it reports will be more accurate.

Both of these parallel techniques compute the same results as the sequential versions of Gibbons and Matias. An implementation using data small enough to allow complete counts to be maintained has been used to check that the

answer returned are indeed correct.

4 Exploiting Hierarchical Structure

So far, we have considered only the problem of counting the frequencies of single objects. However, for the frequent set problem we wish to count the frequencies of subsets of each row. A row of the array with with, say, a hundred non-zero entries denotes 2^{100} possible subsets of the attributes, all of which might potentially be frequent. Each row of the table could, in the worst case, require an amount of computation exponential in m.

If each of these subsets is treated as a single object in the algorithms of the previous two sections, then it is not clear that the performance of these single-pass algorithms will be better than levelwise techniques. Levelwise algorithms rarely generate or consider large subsets because this can only happen if all of the smaller subsets are frequent. Since a set of size k has k subsets of size $k-1$ and they must *all* be frequent, this becomes increasingly unlikely.

A naive implementation of the algorithms described in the previous section would treat all subsets equally, no matter their size, and this generates an unacceptable large amount of work. Intuitively, there is no point in even considering subsets of size, say, 50 for sampling since they are extremely unlikely to be frequent. The lattice structure of subsets must be exploited to reduce the computation per row of the array to acceptable levels.

Recall that the probability that a set that is actually frequent is missing from a concise sample is $1/\tau$. Consider a set of size k, with k subsets of size $k-1$. When we come to consider adding the set to the sample, some of its subsets may already be in the sample, and some may not. If all of its subsets are present, then it has the same chance of being frequent as any other set and should be added to the sample with probability $1/\tau$. If only j of its subsets are present ($j < k$) then we intuitively expect that we should be less likely to add it. The chance that the missing j subsets are frequent (and so our candidate set could be) but not in the sample is

$$\left(\frac{1}{\tau}\right)^j$$

so the new set should be added with probability

$$\left(\frac{1}{\tau}\right)^{(j+1)}$$

Thus large subsets are inherently less likely to be added to the sample, depending on the presence or absence of their own subsets. (Note that the sample is now a stratified uniformly random sample.)

This new strategy does not, however, reduce the amount of work required to generate and test the subsets denoted by each row of the array. It simply makes it unlikely that the larger subsets will be added to the sample. The work required to decide whether or not a subset is to be added is substantial regardless of which way the decision goes.

The chance of missing a large subset by failing to consider it for inclusion when j of its subsets are not present in the sample is

$$n\left(\frac{1}{\tau}\right)^j$$

while the chance of missing a frequent set altogether is $1/\tau$. Choosing the value of j that makes these two equal makes it as likely to make an error by ignoring a large subset as by using a probabilistic insertion technique at all. Call this value l.

Each row of the array is therefore processed in the following levelwise fashion. The lattice of sets of increasing size is generated one level at a time. Each set is considered for inclusion with probability $(1/\tau)^{(x+1)}$ where x is the number of its direct subsets missing from the sample. As a side-effect of using a counting sample, the presence in the sample of each potential set is checked, so that there is no additional overhead in determining which of a set's subsets are already present. Sets which are missing more than l of their direct subsets are not considered for inclusion in the sample. Hence the lattice extends beyond the region of sets already in the sample only with low probability; and large subsets are unlikely to be considered at all.

The insertion process for each row of the array therefore grows a lattice based on the non-zero entries of the row. However, only a small fraction of the potential lattice will be built in general. It will extend as far as sets that are in the sample, and a little further with probabilities that diminish exponentially.

Gibbons and Matias report that τ reached magnitudes of 1000 in their experiments, so that the probability of large sets being considered for inclusion when their subsets are not already included quickly becomes small, even for large arrays. The amount of the subset lattice constructed is not much larger than that used by levelwise algorithms. Thus the computation required for the new algorithm is only slightly greater than that of a levelwise algorithm.

Notice that all of the clever data structuring and incremental construction

techniques used in sequential levelwise algorithms can be immediately applied to the parallel case [7].

5 Handling Large Results

One of the drawbacks of counting frequent sets is that the size of the answer can itself be too large to fit into memory. When the data has 1000 attributes, the number of subsets of small size (say 1,2,3,4) is large, and these sets are more likely to be frequent than larger subsets. Levelwise algorithms need only keep subsets of the current and previous sizes, but may still run into memory problems.

Here we describe a static partitioning technique that can be used to allocate work to processors in a reasonably balanced way. The idea can be used as a simple parallelization for sequential levelwise algorithms also.

Suppose that we want to execute the parallel algorithm described in Section 3. Use a set of $p.q$ processors divided into p clusters each of size q (q odd). Divide the attributes into q groups.

We illustrate the technique for $q = 5$. Consider one cluster of q processors which has as its data a subset of the rows of the data array. Each processor is responsible for building subsets whose attribute members come from the regions described below, and for accumulating frequencies for such subsets:

Processor	Regions
1	A, AB, AC, ABC, ABD, ABCD
2	B, BC, BD, BCD, BCE, BCDE
3	C, CD, CE, CDE, CDA, CDEA
4	D, DE, DA, DEA, DEB, DEAB
5	E, EA, EB, EAB, EAC, EABC

Each processor must pass on some of its computed frequencies as shown in Figure 2.

The allocation of regions to processors is based on cyclically shifting difference sets. There are four non-zero differences modulo 5. The pair AB covers +1 and -1, while AC covers +2 and -2 (mod 5). Hence a cyclic shift of these pairs produces all pairs of 5 elements. Similarly, the triples ABC and ABD have different patterns of differences and hence their cycles are disjoint. Furthermore, each particular difference occurs three times, so that each pair appears exactly the required three times as part of a different triple. This

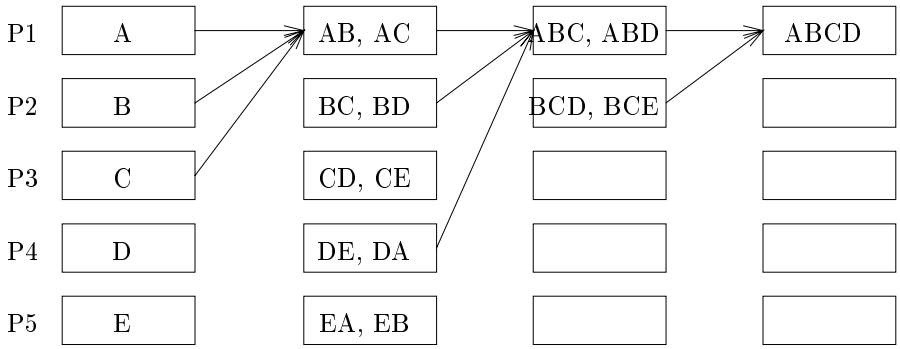

Figure 2: Data flow of partitioned algorithm (only showing communication to Processor 1)

technique can be extended to any odd value of q (a counting argument shows that it cannot work for q even).

Note that the sets of regions are independent of the sets of attributes being considered for frequency. The table below shows a typical row of the data array, and its allocation to 5 processors:

Proc	A			B			C			D			E		
Attr	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Val	0	1	1	0	1	1	0	1	0	1	1	1	0	1	1

In this allocation, Processor A is responsible for maintaining the window of frequencies for all subsets containing only the attributes 1-3; subsets containing attributes from both 1-3 and 4-6; subsets containing attributes from both 1-3 and 7-9; subsets with members from each of 1-3, 4-6, 7-9; subsets with members from each of 1-3, 4-6, 10-12; and subsets with members from each of 1-3, 4-6, 7-9, and 10-12.

Because this allocation is static and data-independent, there is no way to guarantee load balancing except by appeal to the law of large numbers. It might be argued that some attributes are more frequent than others. If this is known, then the information can be exploited by allocating columns to processors so that *a priori* frequent attributes are spread evenly.

The communication requirements between stages are evenly balanced in the sense that each processor sends and receives messages from an equal-sized segment of the attributes. This is known experimentally to produce good performance on typical parallel architectures [3, 6] provided that the order of message transmission is scheduled to avoid collisions at the receivers (easy in

this setting). Note that each of the q processors could itself be replicated $q-1$ times and the data flow diagram of Figure 2 implemented by a pipeline of processors, giving further potential speedup.

If we regard the p sets of q processors each as forming a grid, with an identifier of the form $(i, j), 1 \leq i \leq p, 1 \leq j \leq q$, then the processors whose second coordinates are the same are accumulating frequencies for the same subsets of the attributes. After the pass through the data, each set of such processors resolves their frequencies and produces part of the answer. The complete answer is the *disjoint* union of these p partial answers, so that no processor need have enough memory to hold the complete solution.

Note that this static allocation technique can be used as a simple parallelization strategy for conventional levelwise algorithms also.

6 Conclusions

Effectively computing the frequent sets from a binary array of object-attribute data requires exploiting three opportunities:

1. Parallelism over the rows of the array. We have shown how to do this using a parallel form of the Gibbons-Matias approximate sampling techniques.

2. Parallelism over the columns of the array. We have shown how to use a static work-partitioning technique to do this.

3. Avoiding the computation of the entire lattice of subsets implied by the non-zero entries of each row of the data array. We have shown how to do this by reducing, and eventually eliminating, the probability of inclusion of a large subset unless its own immediate subsets are potentially frequent.

The resulting algorithm makes minimal data accesses, requiring only one pass over the data array. Its communication demands are modest, consisting of balanced exchanges of <subset identified, count> pairs. These are passed between members of each cluster of q processors in a dataflow fashion during the computation of the lattice grown from each row of the data array, and exchanged between clusters at the end of the algorithm to compute the final count totals. Both of these communication requirements are, in some sense, unavoidable: complete lattices do not have small bisection widths so we cannot

avoid the former communication by clever allocation; and communication to get the final answer is similarly unavoidable. The computational requirements are not much greater than those of levelwise algorithms; the chief difference being that the algorithm described here must extend the lattice of subsets beyond the point where a levelwise algorithm would stop. However, the probability of doing so is a power of a value $(1/\tau)$ that tends to become small, so this represents only a small amount of extra work (concentrated in the early stages of the algorithm when τ is small. For all but the smallest data sets, accessing the data is the performance bottleneck. The algorithm presented here is an improvement over both existing sequential levelwise algorithms, and their parallelized forms.

References

[1] R. Agrawal and J. Shafer. Parallel mining of association rules: Design, implementation and experience. Technical Report RJ10004, IBM Research Report, February 1996.

[2] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[3] S.R. Donaldson, J.M.D. Hill, and D.B. Skillicorn. BSP clusters: High-performance, reliable, and very low cost. Technical Report Technical Report PRG-TR-5-98, Oxford University Computing Laboratory, 1998.

[4] P.B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of ACM Conference on the Management of Data*, pages 331–342, 1998.

[5] E.H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proceedings of ACM Conference on the Management of Data*, pages 277–288, 1997.

[6] J.M.D. Hill and D.B. Skillicorn. Lessons learned from implementing BSP. In *High-Performance Computing and Networks*, Springer Lecture Notes in Computer Science Vol. 1225, pages 762–771, April 1997. Also appears as Oxford University Computing Laboratory Technical Report TR-96-21.

[7] Z. Hu, W.-N. Chin, and M. Takeichi. Calculating a new data mining algorithm for market basket analysis. In *Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, Lecture Notes in

Computer Science, Boston, Massachusetts, January 2000. Springer Verlag.

[8] N. Lesh, M. Zaki, and M. Ogihara. Mining features for sequence classification. In *5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, San Diego, CA, August 1999.

[9] H. Toivonen. Discovery of frequent patterns in large data collections. Technical Report A-1996-5, Department of Computer Science, University of Helsinki, 1996.

[10] M. Zaki. Parallel and distributed data mining: A survey. *IEEE Concurrency*, 7(4):14–25, October–December 1999.