Data Organization and Access for Efficient Data Mining

Brian Dunkel Nandit Soparkar

The University of Michigan
Electrical Engineering & Computer Science
1301 Beal Avenue, EECS Building
Ann Arbor, MI 48109-2122
{bedunkel, soparkar}@eecs.umich.edu

Abstract

Efficient mining of data presents a significant challenge due to problems of combinatorial explosion in the space and time often required for such processing. While previous work has focused on improving the efficiency of the mining algorithms, we consider how the representation, organization, and access of the data may significantly affect performance, especially when I/O costs are also considered. By a simple analysis and comparison of the counting stage for the Apriori association rules algorithm, we show that a "column-wise" approach to data access is often more efficient than the standard row-wise approach. We also provide the results of empirical simulations to validate our analysis. The key idea in our approach is that counting in the Apriori algorithm with data accessed in a column-wise manner significantly reduces the number of disk accesses required to identify itemsets with a minimum support in the database — primarily by reducing the degree to which data and counters need to be repeatedly brought into memory.

1. Introduction

The algorithms and techniques of *data mining* (DM), or more generally *knowledge discovery in databases* (KDD), attempt to find useful patterns that characterize very large datasets. However, the development of DM faces the obstacle of large requirements in time and space for processing. Especially given that DM techniques are generally applicable and of interest when the target dataset is relatively large, these difficulties are inherent in KDD. We argue that the solutions will require novel approaches to data access and manipulation, together with the careful application of new and existing algorithms. The tasks of KDD should be supplemented by existing data management technologies, in particular, appropriate data representations and access methods. Motivated by the particular DM technique of finding association rules [2], and a number of representative datasets, we explore one manner in which database techniques could be used profitably.

The growth of new sources of non-standard data and metadata bring situations where problems of combinatorial explosion become manifest. The collection of HTML documents that comprise a significant portion of the Web is one source for such non-standard data. Once a set of keywords has been extracted from a set of documents, they can be stored in a tabular form. Application of association rules to such metadata may include the identification of document characteristics based on an association with the keywords in the document. For example, it may happen that many documents containing the keyword *football* also contain some synonym for *luxury car*. Using this information, the marketing efforts for luxury cars may direct informational e-mail to users whose homepages contain *football*, even if they have no mention of luxury cars. Other applications of KDD to the Web include the discovery of interesting patterns in the statistics gathered as a record of Web usage. Similar datasets can be found in more standard association rules tasks, such as retail sales marketing. An instance of such data might be a highly specialized catalog retailer, whose inventory consists of a large number of different items, each of which is only is purchased only by a small (but loyal) group of customers. In these cases, the number of items to be considered becomes significantly large.

The efficient discovery of association rules in keyword metadata, presents a challenge because the tables tend to be "wide" (i.e., many different items may possibly occur in any given record) rather than "long" (i.e., any given item tends to appear in relatively few records). The number of different combinations of the items grows exponentially, and becomes more problematic than the size of the original database itself. At the same time, data may exhibit other characteristics (e.g., the itemsets tend to be sparsely distributed, and there tends to be a limit to the size of the rows or columns) that may be exploited to make the counting process more efficient. The problems arising from the growth in the number of items (rather than the number of records) is an issue even for market-basket data to which association rules have been applied in the past (e.g., as noted in [5]). That is, several types of datasets have the wide and sparse characteristics for which our research is most applicable.

The majority of available techniques for discovering association rules have aimed to improve the performance of *row-wise* algorithms (i.e., which scan the database by rows) for counting items in the database (work such as [10] being a notable exception). Our approach, in contrast, accesses the database in a *column-wise* fashion, which leads to a reduced number of I/O operations required for some datasets, especially of the type that we discussed above. The key idea is that counters, which maintain the identity and frequency of occurrence for a particular set of items, need not be repeatedly (and expensively) brought into the memory. As in similar efforts, we provide a comparative analysis and experimental validation against the Apriori algorithm [4].

The remainder of this paper is organized as follows: Section 2 discusses some of the related work and describes briefly the standard Apriori association rules algorithm. Section 3 examines analytically the problem of counting the support for itemsets in data with certain characteristics using a column-wise approach, with Section 4 comparing the processing and I/O costs. Section 5 briefly describes some of our empirical results, and Section 6 concludes the paper.

2. Related Work

A particular DM technique which has shown promise in the area of “market-basket analysis” for retail sales data, is the use of association rules algorithms described in [3]. The goal of these algorithms is to identify relationships between sets of items, or simply itemsets, in the database, where two items are related if they appear together in the same record (or row). Anecdotal evidence (e.g., [2]) suggests that association rules can be used to describe relationships within large, highly structured datasets, such as the sales records database of a retail outlet chain. Efficiency and scalability concerns continue to remain significant problems for these algorithms.

2.1. The Apriori Algorithm

Our description of the Apriori association rules algorithm and frequent itemsets has been adapted from [4, 8].

An *item* in a database is an (attribute, value) pair, and an *itemset* is a collection of items. An *association rule* is a rule of the form:

$$A_1, A_2, \ldots, A_n \Rightarrow B_1, B_2, \ldots, B_m$$
$$\text{with confidence} = c\% \text{ and support} = s.$$

Here each of $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ are items. The *support* of an itemset is the number of distinct records in the database in which the full itemset appears. Let A denote the itemset containing $A_1, \ldots, A_n$, and B the itemset containing $B_1, \ldots, B_m$. The support of an association rule is the support of the union of the two itemsets involved. That is, $s = \text{support}(\mathrm{A} \cup \mathrm{B})$. The *confidence* of a rule is the percentage of records containing itemset A that also contain itemset B. That is, $c = \frac{\text{support}(\mathrm{A \cup B})}{\text{support}(\mathrm{A})} \times 100$. Support is a measure of the likely statistical significance of an association rule, while the confidence is an indication of the likely causality implied by the rule.

```
1) $L_1 = \{$frequent 1-itemsets$\}$;
2) **for** ($k = 2$ **until** $L_{k-1} \neq \emptyset$ **step** 1) **do begin**
3)     $C_k$ = apriori-gen($L_{k-1}$); // New candidates
4)     **forall** records $r \in \mathcal{D}$ **do begin**
5)         $C_r$ = subset($C_k, r$); // Candidates in r
6)         **forall** candidates $c \in C_r$ **do**
7)             c.count = c.count + 1;
8)     **end**
9)     $L_k = \{c \in C_k \mid$ c.count $\geq$ minsup; $\}$
10) **end**
11) Answer = $\cup_k L_k$ ;
```

Figure 1. The Apriori Counting Algorithm

The (general) association rules problem is to find *all* collections of items in a database whose confidence and support meet or exceed user-specified minimum levels. The general algorithm iteratively considers itemsets from the smaller to the larger ones, pruning away unlikely candidates where possible. The counting portion of the Apriori algorithm for discovering frequent itemsets is given using structured pseudo-code in Figure 1. First, the support of individual items is calculated by counting the number of records or rows in which each appears. The resulting set of frequent 1-itemsets is denoted by L_1. In each subsequent pass over the database $\mathcal{D}$, the frequent itemsets from pass $k-1$ are used to generate potentially frequent *candidate* k-itemsets (i.e., k-itemsets that could possibly be frequent) for the kth pass over the database.

The set of candidate k-itemsets, C_k, is generated by combining elements of the frequent itemsets identified in the previous pass (i.e., itemsets from the collection L_{k-1})

in two stages, represented collectively by the subroutine *apriori-gen()*. The "join" step produces itemsets that are the result of combining two input itemsets sharing $k-2$ common items. The single items by which the two $(k-1)$-itemsets differ are added to the common items, forming a k-itemset. The "prune" step removes, *a priori*, any of the newly generated k-itemsets that could not be frequent, because one of its $(k-1)$-subsets is known to be infrequent. For example, if $\{ABC\}$ and $\{ABD\}$ were frequent 3-itemsets, the join would produce a candidate $\{ABCD\}$ and the prune would verify that all of $\{ACD\}$, $\{BCD\}$, etc. are also frequent.

Once the candidates have been generated, a row-wise pass is made over the database and the count is incremented for any subset of a row which is a candidates. Once the pass is complete, those itemsets that are not frequent (i.e., do not have a count in the database greater than or equal to $minsup$, the specified minimum support parameter) are eliminated before the next pass, giving the collection of frequent k-itemsets, L_k.

3. Association Rules over Wide, Sparse Tables

We present an analysis and compare to the Apriori algorithm our new strategy for counting the support of itemsets in a database. Given the structure of the target data, we consider a *column-wise* (CW) approach, rather than a *row-wise* (RW) approach to counting. This shift leads to the development of data representation and manipulation techniques that are more efficient — which we demonstrate through an analysis of the I/O requirements for both the standard RW approach as well as the new CW approach. Our new approach to the association rules counting problem is a novel departure from the standard method of accessing tabular data in conventional databases.

3.1. Column-Wise Apriori Counting Algorithm

Figure 2 provides a pseudo-code description of an algorithm similar to the Apriori algorithm described in Section 2.1, where those lines that differ are marked with a prime ($'$) symbol. An important distinction between this algorithm and the standard Apriori algorithm is that the database is accessed by column, rather than by row, where a column is regarded as a list of row identifiers for those rows in which the item appears or as a value in a bit-mapped index of the database. As with the standard algorithm, the set L_1 is calculated by a single pass over the database. Unlike the RW algorithm, however, which may possibly access the entire set of candidate counters as it scans perhaps even one row, the CW algorithm updates only a single counter for the column currently being counted. This has significant ramifications with respect to the efficiency of the processing, as

```
1')    $L_1 = \{$frequent 1-itemsets$\}$; // Column-wise scan
2)     **for** ($k = 2$ **until** $L_{k-1} \neq \emptyset$ **step** 1) **do begin**
3)     $C_k$ = apriori-gen($L_{k-1}$); // New candidates
3a')       **forall** candidates $c \in C_k$ **do begin**
4')            **forall** $(k-1)$-subsets of $c$ **do**
5')                Verify the subset is frequent;
6')            **if** $c$ has no infrequent subset
7')                Count the support of $c$ by conjoinment;
8)         **end**
9)         $L_k = \{c \in C_k \mid$ c.count $\geq$ minsup $\}$;
10)    **end**
11)    Answer $= \cup_k L_k$ ;
```

Figure 2. Column-Wise Apriori Counting

the number of disk accesses can be substantial.

It is convenient to discuss the data for association rules in terms of a bit-mapped representation: each set of items is conceptually represented by a bit vector in which present items are represented as "1" and absent items are represented as "0". For sparse data, a list representation where each present item is described explicitly may more efficient. In this latter representation, a single item or row identifier is assumed to be the same size as a single counter in memory (e.g., the size of an integer or machine word, but in any case more than a single bit). Since each column represents a single item, the number of row identifiers present in the column in question (or, equivalently, the number of 1 bits in a bit-mapped index value) determines the support for a given item.

In a manner similar to the RW algorithm, until there are no more frequent itemsets of size k, the CW algorithm uses a "join-and-prune" strategy to generate the set C_k. For each candidate, all $(k-1)$-subsets are verified by lookup in L_{k-1} to confirm that they are frequent. If any of these subsets are found to be infrequent, then the candidate cannot be frequent and it is discarded. If the candidate has no infrequent subsets, then the columns representing its constituent items are read and used to count the support for the itemset, using an operation we refer to as *conjoinment*.

Conjoinment can be thought of as a vector operation that combines a bit-wise intersection with a series of increment operations to produce a new vector and a count, such that the output "column" has items present where *both* of the input columns contain the same item. Items absent from either of the two inputs will be absent from the output. For example, if the input to this operation is a pair of database columns, representing the 1-itemsets $\{X_a\}$ and $\{X_b\}$, where $\{X_1, \ldots, X_n\}$ is the set of all items in the database, then the result of the conjoin will be a single vector representing the 2-itemset $\{X_a X_b\}$, as well as a count of the support for this itemset in the database. If this result and the column representing $\{X_c\}$ were then used as inputs for the conjoin, the result will be a "column" representing $\{X_a X_b X_c\}$ and the support count for this itemset.

Once all the candidates have been generated and counted, candidates for which the support in the database meets a user-specified minimum are added to L_k (in preparation for the next pass of the loop). If none of the candidates has sufficient support, the loop terminates, at which point the identity and support of all frequent itemsets is known.

4. Cost Analysis

We assume that the items in a database are represented by identifiers, which can be lexicographically ordered, and the representation of an itemset maintains the order of its items. The manipulation of itemsets (e.g., the join) preserves this order. Also, we assume for purposes of analysis that the representation of the data (for a row, column, or itemset) is a list of identifiers. We suggest this representation, rather than a bit-vector representation, since it is more efficient for sparse datasets. The size of a single identifier is the unit of space used in the following analysis, and this is also assumed to be the size of a single counter without an associated identifier.

We assume for the datasets considered here that the number of columns, n, is greater than the number of rows, l. We use B^C_{max} to represent the maximum number of items in any column and B^R_{max} to represent the maximum number of items in any row. In most cases, this should be an upper bound, but for neither the CW nor the RW algorithm will it be an under-estimate, and it simplifies our comparative analysis. We further assume that $B^C_{max} \ll l$, $B^R_{max} \ll n$, and that lB^R_{max}, nB^C_{max}, and the size of the dataset are of similar magnitude, which is justifiable for the type of datasets with which we are concerned. Possible values could be $B^C_{max} = 50$, $l = 1000$, $B^R_{max} = 30$, and $n = 5000$.

Our analysis uses reasonable worst-case assumptions about the distribution of the data and the number of itemsets that need to be processed. Such "rough" analysis is warranted since it may be difficult to analyze statistically the data distribution, which may change frequently in practice. In previous analyses (e.g., [4]), market-basket data is assumed, or in some cases observed empirically, to yield fewer frequent itemsets as the size of candidate itemsets increases. For our analysis, we make no assumptions about the likely amount of support for itemsets of any size, except to assume a maximum length for the candidate itemsets to be considered.

4.1. Row-Wise Processing

We consider the processing cost of the RW strategy for counting frequent itemset support.

In the first pass, each single item must be counted to establish the support of the 1-itemsets, and this is done by a RW scan of the entire database. This requires $lB^R_{max}\ \mathrm{space}$ for the database itself. For each row, the RW algorithm must update the counter for each item actually present in the row. Since any item may be present in any row, all n items must have a counter along with a representation of the item itself, which requires $2n\ \mathrm{space}$ for the counters and single itemsets.

For this comparative analysis, since the two algorithms are assumed to use the same mechanism for itemset lookup, and both produce the same number of candidates, we assume the existence of a fixed-time, hash-based lookup scheme, and drop this constant term from further expressions of time cost. Thus the entire step of calculating L_1 takes, in the worst case, $lB^R_{max}\ \mathrm{time}$ to increment all counters, and $n\ \mathrm{time}$ to determine which of the 1-itemsets are indeed frequent.

Assume that all potentially relevant counts are to be maintained (which is necessary for the later generation of association rules involving all frequent itemsets). Then, the required number of counters will be $|L_k|$ for iterations $k = 2, \ldots, k_{max}$, and $|C_{k_{max}+1}|$ for the last iteration, where $|L_k|$ represents the number of elements in the set L_k. The total amount of space needed for the counters without their identifying candidate sets for all iterations of this loop will be $\left(\sum_{k=2}^{k_{max}} |L_k|\right) + |C_{k_{max}+1}|$. The space required for the candidate sets themselves is described below. Note that in the worst case, the support in the database will be such that all k-itemsets for $k \leq k_{max}$ will be frequent, so that $|L_k| = \binom{n}{k}$ and $|C_k| = \binom{n}{k}$, and the space needed will be $\sum_{k=2}^{k_{max}+1} \binom{n}{k}$.

Since the join step actually produces all possible extensions before pruning, and each new candidate will contain k items and there will be $|C_k|$ such candidates, this step requires $k|C_k|\ \mathrm{space}$ for the generated itemsets. At a minimum, each of the $|C_k|$ candidate sets of size k must be the result of joining at least two members of L_{k-1}, and producing each member of C_k will require $k-1$ comparisons ($k-2$ of which are to determine equality, and the last of which is to determine the order of the final two items). Thus, at least $(k-1)|C_k|\ \mathrm{time}$ is required for the join of items from L_{k-1} to produce C_k.

For the prune step, each of the possible $(k-1)$-subsets of each k-itemset must be verified for minimum support, and we assume that the amount of time to verify a single $(k-1)$-subset of any member of C_k is fixed, regardless of the size of L_{k-1}. In the worst case, all $k-1$-subsets of an element in C_k will be frequent, and therefore all must be verified for a total cost of $k|C_k|$ time for the prune step.

For the counting, since each row has at most B^R_{max} items, it lends support to no more than $\binom{B^R_{max}}{k}$ k-itemsets, and it may take as much as $l\binom{B^R_{max}}{k}$ time to increment the necessary counters (assuming a unit cost for locating the counters). Finally, we compare each count with the user-specified minimum support, for a cost of $|C_k|$ time. Since all of the counters are being accessed in a single scan, there is no additional cost for locating each counter.

4.2. Column-Wise Processing

As in the RW algorithm, each single column must be counted to calculate and verify its support. The database itself may need at most nB^C_{max} space and the counters and their associated identifiers will take $2n$ space. Counting the support of all of the columns will also take at most nB^C_{max} time and the verification that each meets the minimum support will take n time.

Both algorithms use the same subroutine (i.e., apriori-join of [4]) to compute the same collection of candidate sets, and the space for the counters is as in the RW algorithm. For the join step of the CW algorithm, we will produce only one new candidate, c, at a time, and count its support if it has no infrequent $(k-1)$-subsets. As before, the process of joining two elements of L_{k-1} requires $k-1$ time. There will be k subsets of size $(k-1)$ that will need to be verified, which will take k time and no additional space to verify that each is frequent. As in the RW case, we assume a fixed-time lookup for each of the subsets, regardless of the value of k.

The conjoinment operation was described in Section 3.1. For support counting in the CW algorithm, the analysis differs depending on whether the results of a conjoin operator are preserved at each stage (when the itemset is shown to be frequent) or the conjoin is recomputed for each candidate. If the results of a conjoin are preserved, a new itemset will be created together with its counter value. Therefore, each count will take up to αB^C_{max} time, where the small (e.g., 2 or 3) constant α is determined by the exact algorithm used to compute the conjoin. Each new frequent conjoin may also require as much as $k + B^C_{max} + 1$ space for the conjoin itself and its related identifiers and counter. Finally, the verification of support for all candidates takes $|C_k|$ time.

4.3. Disk Access Costs

We now examine and compare the I/O cost of both the RW and CW representations for counting the support of frequent itemsets. This comparison will allow us to draw more realistic conclusions about the relative performance of the two methods, since such frequent datasets and bookkeeping information would entail many disk accesses.

We assume the amount of memory for use by the algorithm to be M blocks. These M blocks do not include the memory needed by the operating system or disk access mechanism (e.g., index structures needed for lookup). Where appropriate, the set of M blocks is subdivided into M_D blocks of memory for data, into which blocks will only be read, and M_C blocks of memory for counters and other bookkeeping information which may need to be written again to disk. In stages of the algorithm (e.g., the join) for which three areas of memory are necessary, the M_D blocks of memory are further subdivided into M_{D_1} and M_{D_2} blocks of memory, where $M_{D_1} + M_{D_2} = M_D$.

Also, we assume that certain numbers of elements will fit into a single block (i.e., the blocking factor for itemsets, counters, rows, etc.). Each of these factors is designated by a variable, f, with an appropriate subscript. So, the blocking factor for a unit space entity (e.g., a single counter without its related identifier) is f_I, the blocking factor for rows is f_R, and the blocking factor for columns is f_C, where it is to be understood that we mean the average number of rows or columns in a block. It should be noted that these blocking factors are integer values, which means that most of the expressions in the following sections should have appropriate ceiling or floor modifiers — which we ignore for the sake of simplicity in the expressions.

As before for the row-wise approach, the initial step to determine L_1 requires time and space on the order of the database size and the main loop will iterate until $k = k_{max} + 1$. For the join, let M_{D_1} be the number of data blocks used for the "outer" loop over L_{k-1} and M_{D_2} be the number of "inner" blocks. The blocks of the outer copy are read once, at a cost of $k|L_{k-1}|/f_I$ I/O reads, where use of the multiplier k assumes that each frequent itemset also includes space for its counter. For each set of M_{D_1} outer blocks, the remaining outer blocks must also be read. Once the join and prune are complete, the support of each candidate in C_k must be counted. Each row could require access to a minimum of 0 counters (e.g., if the number of items in the row were less than k) and a maximum of $\binom{B^R_{max}}{k}$ counters (if each k-subset of the row were to support a candidate). In the best case, all $|C_k|$ counters will fit into M_C blocks, with an I/O cost of $lB^R_{max}/f_I + (k+1)|C_k|/f_I$. In the worst case, as each candidate counter is incremented a block access is entailed, giving a cost of $|C_k|$ block accesses per set of rows fitting into M_D blocks. If we examine the cost of reading the entire set of counters for each set of M_D rows (which will be necessary in the worst case) then the number of I/O block reads needed to increment all counters would be $lB^R_{max}/f_I + l/(M_D f_R)(k+1)|C_k|$. This expresses the cost of reading the entire database (once) into M_D blocks, and then for each set of M_D blocks, staging in the entire set of $|C_k|$ counters (each of which is of size $k+1$). Since each block read and updated will also have to be written, this will require an additional write cost of $l/(M_D f_R)(k+1)|C_k|$. Finally, the support count for each candidate must be compared to the minimum support, and the frequent itemsets, L_k, will need to be written out to disk, for a read cost of $(k+1)|C_k|f_I$ and a write cost of $(k+1)|L_k|f_I$.

The total cost of the RW counting algorithm in terms of disk access is given by the cost of each phase for reading and writing, summed over all values of k up to $k_{max}+1$. The body of the summation can be expressed as:

RW algorithm cost =
Read cost of join + Write cost of join +
Read cost of prune + Write cost of prune +
Read cost of counters + Write cost of counters +
Read cost of C_k + Write cost of L_k

We do not build this expression explicitly here, since many of the terms will be common to the CW analysis, and we need only compare the terms by which the two differ.

If we choose not to keep the results of the conjoin, the I/O cost of the CW algorithm for each value of k is:

CW algorithm cost =
Read cost of join + Write cost of join +
Read cost of prune + Write cost of prune +
Read cost of counters + Read cost of columns +
Write cost of L_k

Given that the cost of the join, prune, and writing out of the frequent itemsets in L_k are common to both the RW and the CW algorithm, we need not consider them for our comparison. This leaves the cost of accessing the counters, candidates, and columns, as appropriate. For the CW approach, if we choose not to keep the results of the conjoin, we have the cost of reading a single candidate k columns for each member of C_k. In the worst case, the columns might be accessed in an order that requires a new block read each time, giving block accesses on the order of $kB^C_{max}|C_k|$ and $(k+1)|C_k|$ for each pass over the candidates of size k, where $k=\{1,\ldots,k_{max}+1\}$. The size of C_k is the dominating factor in these expressions.

The cost by which the RW algorithm differs from the CW is the cost of reading and writing the counters. In the worst case, each block of rows may require access to every block of counters and the limited memory may force each to be rewritten, so roughly speaking we find that the cost of the RW algorithm is dominated by two terms of magnitude $l(k+1)|C_k|$. Since we assumed that $B^C_{max} \ll l$, it can be seen that the CW approach is more efficient in worst-case I/O cost than the RW approach. For datasets in which the number of rows is far greater than the number of binary columns (i.e., the database is "taller"), the effect of reading the columns for the conjoin becomes more pronounced and the RW approach would at some point become more efficient. Of course, we argue that the trade-offs in the use of materialized information such as conjoins should be considered in this comparison.

5. Empirical Observations

Our key observation is that with limited memory where processing costs are overshadowed by the I/O costs and a large number of items relative to the number of rows, the CW approach is more efficient. As noted previously in this paper, in many cases of interest this happens to be the case. The I/O costs dominate because the processing that is done is more data-intensive than compute-intensive (e.g., few simple operations are performed for a large number of elements accessed from disk). Note that during the counting phase, if each block is read only when needed for a particular subset, then the number of block reads and subsequent writes significantly depends on the distribution of the data, the block replacement strategy, the particular algorithm used to select the order of the subsets, etc. We provide only a few experimental results due to space constraints; additional results are available in [7].

5.1. Experimental Framework

The level of uncertainty in the actual cost of I/O, depending on the characteristics of the data and other related factors (e.g., whether certain data structures will fit entirely into memory) is not well represented in the analysis of Section 4.3. Based on I/O and processing cost analysis for the RW and CW versions of the Apriori algorithm, we have implemented a simulation framework to verify experimentally the effect of various factors on these costs. This framework allows us to simulate and measure various implementations of both RW and CW support-counting algorithms, while controlling parameters such as the block size, the number of available memory blocks, the minimum required level of support for frequent itemsets, whether or not the results of the conjoin are kept, etc.

For the datasets, we used a standard synthetic dataset generator [1] to create simulated market-basket datasets, varying parameters such as the number of distinct items (n), the number of rows (l), and the available support for k-itemsets. We ran a number of experiments with different values for the variables in our analysis (e.g., M, f_I, etc.), in order to verify empirically our analytical results.

All experiments were simulated using a framework written in C++ and Sparc workstations running Sun OS 4.1.4. Itemsets, rows and columns were all represented as extensible arrays of identifiers, where each identifier is taken to be of a unit size (specifically, 4 bytes). The number of identifiers which fit into a simulated block of memory can be specified for each experiment, and this value is then used to determine the number of itemsets, rows or columns in a single block. As itemsets are created, each is added to an available block, and new blocks are "allocated" as needed. If a new block is to be allocated and the number of blocks in the simulated memory (also a parameter of the experiment) is equal to the number of occupied blocks, a victim block is chosen for replacement using one of a number of replacement strategies. If the data in the victim block has been modified since last being written to disk, it is "written," and a simulated data block write is counted. If a data element is required from a block that has previously been written to disk (e.g., for a subset lookup), that block is "read" into the simulated memory and a data block read is counted. When the counting algorithm is complete, the total number of blocks written and read during the simulation is available for examination.

In addition to the number of data block reads and writes, the simulation also counts the number of equal-cost CPU operations (as described in Section 4) and the number of index block reads and writes. This allows a comparison of the different algorithms based on the number of operations performed and the additional cost of index lookups.

The counting algorithms themselves are implemented in an I/O-efficient manner using a block pinning and release strategy. In all cases, the join step is performed using a block nested loop [9] algorithm, with each set of blocks in the outer loop being pinned and released as necessary. The number of blocks available for the inner and outer loops are both parameters of the experiment.

5.2. Effects of Dataset Parameters

The experiments were chosen to compare the effect of various distributions and relative "dimensions" (i.e., number of rows compared to the number of distinct items or columns) of the data. For the RW approach, subsets were identified in each row of the database, and counters were accessed only as needed. For the CW case, the results of the conjoin were not kept. For all experiments, an LRU replacement strategy was used to determine victim blocks in memory. Specific parameters for each of the following figures are given at the end of this section in Figure 5.

Figure 3 shows a comparison of the number of data block reads and writes, for both the RW and CW approach, as the level of minimum support is decreased (leaving all other parameters fixed). Note that as the required level of minimum support is decreased, the difference between the two approaches in the total number of block movements increases at an accelerating rate.

Figure 4 shows the results of an experiment intended to demonstrate the relative I/O cost of the RW and CW algorithms for datasets of "equal height" (i.e., the number of

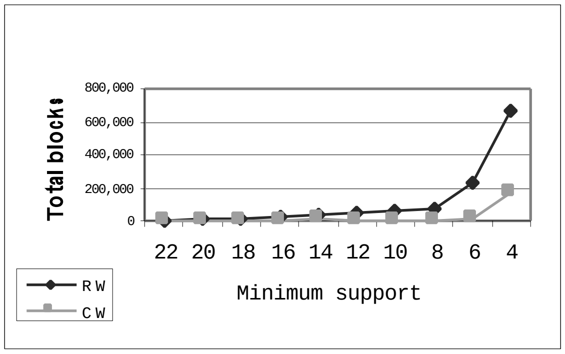

Figure 3. Effect of decreasing support

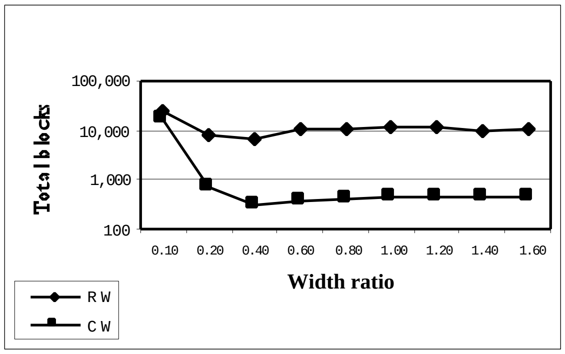

Figure 4. Effect of table width

rows in the data held constant) and varying "width" (i.e., the number of columns). The total number of blocks read or written is plotted in a logarithmic scale along the y-axis, including index blocks. The x-axis shows the relative ratio of the number of columns to the number of rows in the dataset. For example, a dataset with 1000 columns and 1000 rows would have a width ratio of 1, while a dataset with 100 columns and 1000 rows would have a width ratio of 0.1. That is, the higher the width ratio, the "wider" the table is relative to its "height". The number of rows (i.e., l) was held constant at 500, while the number of columns (i.e., n) was varied from 50 to 1000. The other parameters of the dataset (i.e., B^C_{max}, B^R_{max}) were determined by the default values of the synthetic dataset generator. As expected from the analysis of Section 4.3, the CW algorithm is far more efficient than the RW algorithm for datasets with a relatively high width ratio. Once the number of rows is sufficiently greater than the number of columns, the advantage of the CW algorithm is no longer apparent.

Our experiments (more results are presented in [7]) verify that for datasets of the type we described, accessing the database in a column-wise fashion leads to a reduction in the number of I/O operations required. Again, the key point

Figure	Rows (l)	Items (n)	M	
	M_{D_1}	M_{D_2}	f_I	$minsup$
3	1000	1000	16	
	8	4	512	22-4
4	500	50-1000	32	
	10	8	512	$0.2l$

Figure 5. Experimental parameters per figure

is that the counters, which maintain the identity and frequency of occurrence for a particular set of items, need not be repeatedly (and expensively) brought into the primary storage as is the case in the row-wise method.

6. Discussion and Conclusions

We discuss a few possible improvements not explored in our current work, followed by a summary of our conclusions.

- We assumed that the RW algorithm strictly follows the strategy of identifying each subset present in a given row and then accesses the appropriate counter. However, an implementation may also read a number of blocks of counters, and then find all rows with an itemset to support that counter (as per [6]). The ordering of the rows and the counters will likely have an effect on which of these two methods is more efficient.

- An obvious improvement to the cost of the RW algorithm as presented is to perform the join for a pair of $(k-1)$-itemsets (or even a set of blocks thereof) and then immediately perform the check of all its subsets. Given the likelihood that many of the itemsets in a block of L_{k-1} will share common items, this could result in a significant savings, since the cost of writing the join and then re-reading for the prune would be avoided, probably without a significant increase in the number of counter blocks that need to be read.

- In the RW Apriori counting algorithm, the primary purpose of the verification and pruning of all $(k-1)$-subsets of any candidate before actually counting its support is to limit the number of times that a row or candidate need be brought in for support counting. Since the CW algorithm already has all information necessary for counting at one pass, it may be better to avoid the Apriori verification of subsets and simply to count the support of each candidate. This is especially true if the candidates in a given block of itemsets all share a frequent common subset, as the conjoin for that subset could be computed once and used directly in counting the support of candidates sharing that subset. Also, given the possibly larger processing cost of the CW algorithm when memory is unconstrained, and the savings in I/O when memory is limited, some combination of both the RW and CW algorithms may yield the best performance.

As organizations begin collecting their own electronic datasets, and as growing sources of information become available through the Internet, it is imperative that the data organization and access approaches be studied carefully to exploit efficient and promising processing techniques. In particular, in this paper we have examined the I/O efficiency considerations for association rules algorithms with respect to data organization. We have shown that a column-wise strategy for mining tabular data may provide an improvement in the I/O efficiency over a similar row-wise algorithm. We used a simple analysis and experimental validation to support our approach. This provides an indication that similar considerations may significantly benefit other high-cost computing problems associated with mining of datasets.

References

[1] R. Agrawal et al. The Quest Data Mining System. Technical report, IBM Almaden Research Center, 1996. http://www.almaden.ibm.com/cs/quest/.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD Int'l Conf. on Management of Data*, 1993.

[3] R. Agrawal, H. Mannilla, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast Discovery of Association Rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press / MIT Press, 1996. U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors.

[4] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. 20th Int'l Conference on Very Large Databases*, Santiago, Chile, 1994.

[5] R. J. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proceedings of the 1998 ACM SIGMOD Conf. on Management of Data*, 1998.

[6] V. Crestana, Dec. 1997. Informal correspondence.

[7] B. Dunkel and N. Soparkar. Data Organization and Access for Efficient Data Mining. Technical report, The University of Michigan, Ann Arbor, 1999.

[8] B. Dunkel, N. Soparkar, J. Szaro, and R. Uthurusamy. Systems for KDD: From concepts to practice. *Journal of Future Generation Computer Systems*, October 1997.

[9] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, third edition, 1997.

[10] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In *3rd Int'l Conf. on Knowledge Discovery and Data Mining*, pages 283–286, Newport, California, Aug. 1997.