Association rule mining using list representation

F. Wang¹, N. Helian¹ & Y. J. Yip²
¹School of Computing, Communication and Mathematics, London Metropolitan University, UK
²School of Computing and Mathematics, University of Teesside, UK

Abstract

Typically 80% of the data in the logical OLAP datacube, the core engine of data warehouses, are zero. When it comes to sparse, the performance quickly degrades due to the heavy I/O overheads in sorting and merging intermediate results. In this work, we first introduce a list representation in main memory for storing and computing datasets. The sparse transaction dataset is compressed as the empty cells are removed. Accordingly, we propose a new algorithm for association rule mining on the platform of list representation, which just needs to scan the transaction database once to generate all the possible rules. In contrast, the well-known a priori algorithm requires repeated scans of the databases, thereby resulting in heavy I/O accesses, particularly when considering large candidate datasets. In our opinion, this new algorithm using list representation economizes storage space and accesses.

1 Introduction

With huge amounts of data collected in various kinds of applications, data warehouse is becoming a mainstream information repository for decision support and data analysis mainly because a data warehouse facilitates on-line analytical processing (OLAP). The core engine of data warehouses is the datacube, which is exemplified in Fig. 1. This model views data in the form of a data cube. A data cube allows data to be modeled and viewed in multiple dimensions. It is defined by dimensions and facts. In general terms, dimensions are the perspectives or entities with respect to which an organization wants to keep records. The fact
table contains the names of the facts, or measures, as well as keys to each of the related dimension table.

Typically 80% of the data in the logical OLAP cube are zero [1]. When it comes to sparse data cubes, the performance quickly degrades due to the heavy I/O overheads in sorting and merging intermediate results. So a simple way is to compress the sparse array such that the empty cells are removed. In this work, we introduce an efficient method of list representation for storing and computing datasets to support OLAP and data mining.

![Three-dimensional data cube representation of a university example](image)

Figure 1: The three-dimensional data cube representation of a university example, according to the dimensions student, module and time. The measure displayed is the mark of a particular student under a particular module title at a particular time.

2 List representations

There are several list implementations, as shown in Fig.2. The linked list consists of a series of nodes, which are not necessarily adjacent in memory. Each node contains the element and a link to a node containing its successor [2]. We might need to examine every node of the list when searching a linked list (Fig.2a). To execute find(x), we merely start at the first node in the list and then sequentially traverse the list by following the next links. The operation is clearly linear-time.

To avoid such a proportional increase in search time, skip-lists [3] have been introduced. Fig.2(b) shows a linked list in which every other node has an additional link to the node two ahead of it in the list. Because of this, at most \(\lceil N/2\rceil+1\) nodes are examined in the worst case. We can extend this idea and obtain Fig.2(c). Here, every fourth node has a link to the node four ahead. Only \(\lceil N/4\rceil+2\) nodes are examined. In general, every 2ith node has a link to the node 2i ahead of it. The total number of links has only doubled, but now at most \(\lceil \log N \rceil\) nodes are examined during a search. It is not hard to see that the total time spent for search is \(O(\log N)\), because the search consists of either advancing to a new node or dropping to a lower link in the same node. Each of these steps consumes...
at most $O[\log N]$ total time during a search. Notice that the search in this data structure is essentially a binary search. Roughly half the nodes are level 1 nodes, roughly a quarter are level 2, and, in general, approximately $\frac{1}{i}$ nodes are level i.

To perform a find, we start at the highest link at the header. We traverse along this level until we find that next node is larger than the one we are looking for (or NULL). When this occurs, we go to the next lower level and continue the strategy. When progress is stopped at level 1, either we are in front of the node we are looking for, or it is not in the list.

![Diagram](https://via.placeholder.com/150)

(a) Simple linked list:

![Diagram](https://via.placeholder.com/150)

(b) Skin linked list with links to two cells ahead:

![Diagram](https://via.placeholder.com/150)

(c) Skin linked list with links to four cells ahead:

Figure 2: Different list implementations.

For a sparse matrix, what is needed is a list for each row containing the non-zero elements in that row. We also need a list for each column containing the non-zero elements in that column. We call the result an orthogonal list. In principle, we can combine more than two lists into one Fig.3 shows orthogonal linked list implementation for the university sample datacube. As the figure shows, all lists use a header and are circular. To list all of the students in module M1 (Art), we start at M1 and transverse its list (by going down). The first cell belongs to student S1. We can continue and find S3 is also in this module. In a similar manner, we can determine, for any student, all of the modules in which the student is registered. Using a list saves space but does so at the expense of time. The time to perform a search is proportional to the number of nodes that have to be examined, which is at most N (the length of the list). In the worst case, if the first student were registered for every module, then every entry would need to be examined to determine all the module names for that student. Because in this application there are relatively few modules per student and few students per module, this is not likely to happen. To avoid such a proportional increase in search time, skip-lists are integrated to such a sparse matrix implementation.

3 A new algorithm for association rule mining

Association rule mining is a powerful method for so-called shopping basket analysis, which aims at finding regularities in the shopping behaviour of customers of supermarkets, mail-order companies, on-line shops and the like. With the association rules one tries to find sets of products that are frequently
bought together, so that from the presence of certain products in a shopping cart one can infer (with a high probability) that certain other products are present. Such information, expressed in the form of association rules, can often be used to increase the number of items sold, for instance, by appropriately arranging the products in the shelves of a supermarket (they may, for example, be placed adjacent to each other in order to invite even more customers to buy them together) [4].

Figure 3: Orthogonal linked list implementation for the university sample datacube.

Among the best known algorithms is the a priori algorithm [5]. This algorithm works in two steps: In a first step the frequent itemsets are determined. These are sets of items that have at least the given minimum support (i.e., occur at least in a given percentage of all transactions). In the second step association rules are generated from the frequent itemsets found in the first step. Usually the first step is the more important, because it accounts for the greater part of the processing time. In order to make it efficient, the a priori algorithm exploits the simple observation that no superset of an infrequent itemset (i.e., an itemset not having minimum support) can be frequent (can have enough support). It is generally acknowledged that the a priori technique works well in terms of reducing the candidate set. However, there are some criticisms [6] where there are many patterns, long patterns or low support thresholds:

1. Many candidate items sets must still be generated;
2. Requires repeated scans of the databases thereby resulting in heavy I/O accesses particularly when considering large candidate sets.

In this paper we proposed a new association rule mining algorithm on the platform of list representation, which just needs to scan the transaction database once to generate all the possible rules. In order to find the frequent itemsets, we have to count the transactions. The simplest way of processing the transactions is to handle them individually and to apply to each of them the recursive counting procedure described in a priori algorithm [5]. However, the recursion is a very expensive procedure involving frequent I/O operations and degrading the system performance.
for (int i=1;i<=max;i++) { //scan a database
    for (current=header[i];current!=NULL;current=current->right){
        //scan a transaction represented by a list
        if((current->col==1) && (current->value==1)) { //examine the first item
            a=a+1; //counter for 1-itemset a is incremented by 1 as a=1
            ab=ab+0.5; //counter for 2-itemset ab is incremented by 0.5 as a=1
            ac=ac+0.5; //just contributes 1/2 to ab
            ad=ad+0.5;
            ae=ae+0.5;
            abc=abc+0.334; //counter for 3-itemset abc is incremented by 0.334
            abd=abd+0.334; //as a=1 just contributes 1/3 to abc
            aco=aco+0.334;
            ace=ace+0.334;
            ade=ade+0.334;
            abcd=abcd+0.25; //counter for 4-itemset abcd is incremented by 0.25
            abce=abce+0.25;
            abde=abde+0.25;
            acde=acde+0.25;
            abcd=abcd+0.2; //counter for 5-itemset abcd is incremented
            //by 0.2 as a=1 just contributes 1/5 to abcd
        }
        if((current->col==2) && (current->value==1)) { //examine the second item
            b=b+1; //counter for 1-itemset b is incremented by 1 as b=1
            ab=ab+0.5; //counter for 2-itemset ab is incremented by 0.5 as b=1
            //just contributes 1/2 to ab
            .......
            abc=abc+0.334; //counter for 3-itemset abc is incremented by 0.334
            //as b=1 just contributes 1/3 to abc
            .......
            abcd=abcd+0.25;
            .......
            abcd=abcd+0.2;
        }
    }
    ab=floor(ab); //post-processing for each transaction
    //to remove incomplete contribution less than 1
    .......
    abc=floor(abc);
    .......
    abcd=floor(abcd);
    .......
    abcd=floor(abcd);
} //one transaction inspected
} //the whole database scanned

Figure 4: Pseudo-code for a new algorithm described in this article.
On the other hand, it does not suit the naturally linear data structure of the list representation, which is introduced to make the data mining more space efficient. In a linked list, we might need to examine every node of the list when searching for a desired node. The time to perform such a search (a sequential node-by-node procedure) is proportional to the number of nodes that have to be examined, which is at most \( N \) (the length of the list). Therefore it is worth considering how it can be improved.

Our algorithm is described in the pseudo-code in Fig.4. This is a purely sequential (rather than recursive) count procedure which is well compatible with the list representation. To count a certain transaction (represented by a list), we merely start at the first node (item) in the list and then sequentially traverse the list by following the next links. In computing the counters for 1-itemset \( a, b, c, d, e \), we will have scanned each of the 5 items in the transaction. “Is there a way to avoid having to rescan all these items for the computation of other counters, such as \( ab, abc, abcd \) and \( abcde \)?” The answer is. When the any item (if exists) is scanned, its contribution will be comprehensively taken into account. For example, when the first item \( a \) in the transaction is being scanned (say, for the computation of the counter for the 1-itemset \( a \)), all of other \( k(k>1) \)-itemsets \( ab, ac, abc, abcd \) and \( abcde \), relating to \( a \) (inversely, \( a \) is subset of these \( k \)-itemsets), can be simultaneously computed. That is, each counter of the four 2-itemsets, \( ab, ac, ad \) and \( ae \), each counter of the six 3-itemsets, \( abc, abd, abe, acd, ace \) and \( ade \), each counter of the four 4-itemsets, \( abcd, abce, abde \) and \( acde \), the counter of the 5-itemset, \( abcde \), should be incremented then as well. In other words, such a multiway computation aggregates to each of the relating \( k(k>1) \)-itemsets while a 1-itemset is being examined. In computation, the counter for the 1-itemset \( a \) is incremented by 1. Note that the counter for the 2-itemset \( ab \) should be incremented by 0.5 only as \( a=1 \) just contributes 1/2 to the 2-itemset \( ab \), and so on. Similarly the counter for the 3-itemset \( abc \) is incremented by 0.334 (0.334 rather than 0.333 is used to compensate the machine representation error of the decimal number) as \( a=1 \) contributes 1/3 to \( abc \). For 4-itemset \( abd \) is incremented, for 5-itemset \( 1/5 \) is incremented, and so on. After the first item \( a \) is examined, the second item \( b \) of the transaction, currently being scanned, is examined in a similar way. When \( b=1 \), \( ab=ab+0.5 \) is carried out, which makes the final count number in the counter \( ab \) is 1, fairly reflecting the contribution from its two subsets \( a \) and \( b \). After the current transaction is completely inspected, a post-processing is needed. As can be seen in the code in Fig.4, a function floor(\( x \)), which finds the largest integer not greater than \( x \), is carried out to rounds down each counter. The insufficient contributions (less than 1) will be removed, which implies the corresponding itemset is not supported by the current transaction (does not appear).

After the whole database is scanned, the counters for the all possible itemsets, organised in a full itemset tree, is obtained, as shown in Fig.5. This is a full itemset tree for the five items \( a, b, c, d \) and \( e \) exemplified in database \( D \) shown in Table 1. It is not hard to see that this tree is equivalent to the provided transaction database in terms of finding the frequent itemsets. Organising the counters in a full itemset tree not only allows us to store them efficiently (using
little memory), but also supports generating the rules. Each canonical attribute sequence \( S \) denotes a counter for an itemset \( S \). The circled itemsets (infrequent itemsets) will be pruned since they do not have minimum support.

The above is so-called first step of association rule mining, in which the frequent itemsets are determined. The second step of generating association rules from the frequent itemsets is straightforward. Note that there is no need to scan the original transaction database any longer as the counters organised in the itemset tree have retained sufficient information for rule generating.

![Itemset Tree Example](image)

**Figure 5:** A full itemset tree for the five items a, b, c, d and e exemplified in the database shown in Table 1. Each canonical attribute sequence \( S \) denotes a counter for an itemset \( S \). The circled itemsets (infrequent itemsets) will be pruned if the minimum support count is 3.

**Table 1:** A Boolean relational database D.

<table>
<thead>
<tr>
<th>Item</th>
<th>Item</th>
<th>Item</th>
<th>Item</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**4 Experiment results**

The performance of a data mining system is determined by two criteria: processing time and memory utilization. The association rule mining algorithm using list representation is designed to economize storage space and accesses, and we must show that this overriding concern for speed is compatible with a reasonable utilization of available space. Our experiment with association rule
mining algorithm using list representation is based on a simulation program coded in Borland C++ 5.02. For ease of displaying results graphically, the two-dimensional case and the three-dimensional case are considered. Because what we use are a group of one-dimensional arrays, the performance would not be worse in higher dimensional grid space in principle. The program runs on an Intel Pentium III machine under the Windows XP operating system. The CPU frequency is 1.2 GHz. The physical memory size is 512 MB and virtual memory size is 1GB. The simulation has the following objectives: 1. Estimation of array size; 2. Estimation of list size; 3. Estimation of transaction preparation time; 4. Real-time visualization of memory usage; 5. Real-time visualization of CPU usage; 6. Estimation of mining time.

Table 2: Experiment results on Windows XP machine. While disk-block accesses are involved, the average retrieval time increase dramatically.

<table>
<thead>
<tr>
<th>No. of Trans.</th>
<th>Items/Trans.</th>
<th>Sparse ratio (non-zero / total)</th>
<th>Memory Storage</th>
<th>Transaction Preparation (ms)</th>
<th>Rule Mining (ms) [5 itemset max]</th>
<th>Memory Used (MB)</th>
<th>Disk Access?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>1000</td>
<td>10%</td>
<td>Array</td>
<td>2804</td>
<td>3014</td>
<td>382</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>20%</td>
<td>Array</td>
<td>3034</td>
<td>3084</td>
<td>382</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>25%</td>
<td>Array</td>
<td>3145</td>
<td>3054</td>
<td>382</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>30%</td>
<td>Array</td>
<td>3265</td>
<td>3074</td>
<td>382</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>60%</td>
<td>Array</td>
<td>3915</td>
<td>3105</td>
<td>382</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>10%</td>
<td>List</td>
<td>2193</td>
<td>2273</td>
<td>147</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>20%</td>
<td>List</td>
<td>4527</td>
<td>4857</td>
<td>294</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>25%</td>
<td>List</td>
<td>5608</td>
<td>5798</td>
<td>367</td>
<td>No</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>30%</td>
<td>List</td>
<td>6730</td>
<td>7060</td>
<td>440</td>
<td>Yes</td>
</tr>
<tr>
<td>100000</td>
<td>1000</td>
<td>60%</td>
<td>List</td>
<td>40989</td>
<td>158468</td>
<td>990</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Since the association rule mining algorithm using list representation is designed to handle large volume of data, real-time visualization of memory usage by far the most important point. For this purpose, the Performance Monitor in the system tools was used. The sample spaces used in the experiments are as follows: item values of each transaction are determined by a small program generating the database D shown in Table 1 repeatedly.

Table 2 summarizes the experiment results on Windows XP machine. The simulation confirms that the mining time against a list is proportional to its sparse ratio whereas the mining time against an array is independent of its sparse ratio. The simulation also confirms that the memory space against a list is proportional to its sparse ratio whereas the memory space against an array is independent of its sparse ratio. These relationships are depicted in Fig. 6.

As can be seen in Fig.6, the mining time increases dramatically when the sparse ratio exceeds 30%. This is because 512MB (physical memory size) is an important threshold for main memory datasets. While the data mining application space (commit memory) plus operating system space (kernel memory) is below this threshold, all the operations are internal without accessing
the external storage devices. It takes limited amount time. When the threshold is
overtaken, virtual memory technique would be activated and frequent disk-block
accesses, as clearly shown in Fig. 7, would be involved. It takes a relatively long
time to fulfill an operation. Normally we just count the number of disk accesses
into access time.

![Graph showing mining time and memory usage as functions of sparse ratio.](image)

**Figure 6**: Mining time and memory usage as functions of sparse ratio.

![Graph showing frequent disk access degrades the data mining system performance.](image)

**Figure 7**: Frequent disk access degrades the data mining system performance.

5 Analysis and conclusions

Each non-zero element stored in the list representation takes much more space
than an element stored in a simple n x n matrix. When is the list representation
more space efficient than the standard representation? As an example, assume
that a data value uses two bytes, a row or column index use two bytes, and a
pointer uses four bytes (32 bit machine). An n x n matrix requires 2nm bytes.
The list requires 8 bytes per non-zero element (1 pointer, two array indices, and one data value). If we set \( X \) to be the percentage of non-zero elements, we can solve for the value of \( X \) below which the list representation is more space efficient. Using the equation

\[
8\text{nm}X = 2\text{nm}
\]

and solving for \( X \), we find that the list using this implementation is more space efficient when \( X < 1/4 \), that is, when less than 25% of the elements are non-zero. There is a good agreement between the experiment and the estimation here.

The new association rule mining algorithm, specially designed on the platform of list representation, just needs to scan the transaction database once to generate all the possible rules. To count a certain transaction (represented by a list), we merely start at the first node (item) in the list and then sequentially traverse the list by following the next links. The contribution from each node (item) will be comprehensively taken into account. However, it is not without drawback with this new algorithm. In the first step of finding frequent itemsets, we even count those itemsets which are not frequent although they will be pruned eventually. Based on the observation that if any given set of attributes \( S \) is not adequately supported, any superset of \( S \) will also not be adequately supported and consequently any effort to calculate the support for such supersets is wasted. However, considering the advantage of space saving brought by the introduction of list representation and another advantage of performance improving brought by avoiding heavy I/O operations as the transaction database is just scanned once, this new algorithm using list representation presents us with a broad range of trade-offs based on speed requirement and storage requirement.

References