e-PGPathshala
Subject : Computer Science
Paper: Data Analytics
Module No 25: CS/DA/25 - Data Analytics – Association Rule Mining - III
Quadrant 1 – e-text

1.1 Introduction

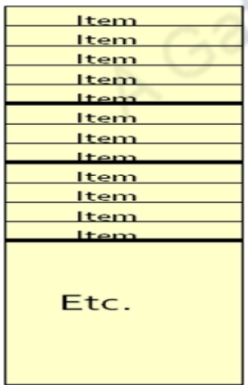
Mining association rules is one of the most used functions in data mining. However when it comes to very large transactional databases, scalability becomes the major challenge. This chapter gives the basics of association rule analysis (ARA) and applications of ARA and also gives a glance on apriori approach for ARA.

1.2 Learning Outcomes

- To Understand the computational challenges in deriving association rules
- To know about alternate and modified approaches for apriori

1.3 Computation Model for Finding Frequent Itemsets

Typically, data is kept in flat files rather than in a database system, stored on disk, and basket-by-basket. The baskets are small but we have many baskets and many items. Expand baskets into pairs, triples, etc. as you read baskets and use *k* nested loops to generate all sets of size *k.* **Note:** Items are positive integers, and boundaries between baskets are –1.



Computation Model

- The true cost of mining disk-resident data is usually the **number of disk I/Os**
- In practice, association-rule algorithms read the data in **passes** – all baskets read in turn
- We measure the cost by the **number of passes** an algorithm makes over the data

1.3.1 Use of Main Memory for Itemset Counting

There is a second data-related issue that we must examine, however. All frequent-itemset algorithms require us to maintain many different counts as we make a pass through the data. For example, we might need to count the number of times that each pair of items occurs in baskets. If we do not have enough main memory to store each of the counts, then adding 1 to a random count will most likely require us to load a page from disk. In that case, the algorithm will thrash and run many orders of magnitude slower than if we were certain to find each count in main memory. The conclusion is that we cannot count anything that doesn't fit in main memory. Thus, each algorithm has a limit on how many items it can deal with.

1.3.2 The Triangular-Matrix Method

Even after coding items as integers, we still have the problem that we must count a pair {i, j} in only one place. For example, we could order the pair so that $i < j$, and only use the entry a[i, j] in a two-dimensional array. That strategy would make half the array useless. A more space-efficient way is to use a one-dimensional triangular array. We store in a[k] the count for the pair {i, j}, with $1 \le i < j \le n$, where

$$\mathbf{k = (i - 1)(n - i/2) + j - i}$$

The result of this layout is that the pairs are stored in lexicographic order, that is first {1, 2}, {1, 3}, . . . , {1, n}, then {2, 3}, {2, 4}, . . . , {2, n}, and so on, down to {n − 2, n − 1}, {n − 2, n}, and finally {n − 1, n}.

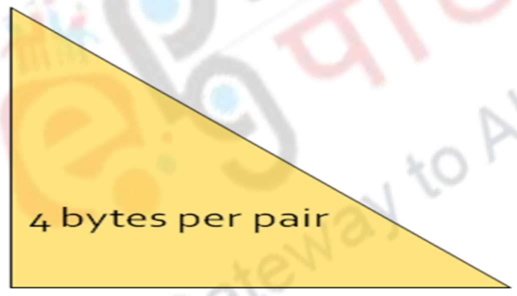


Figure 2(a). Triangular method

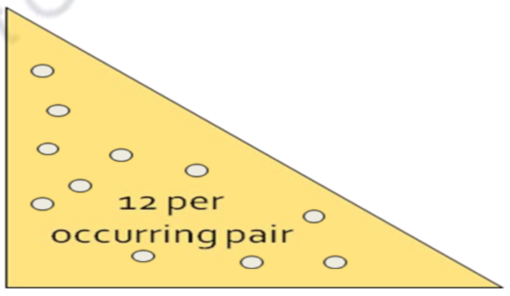


Figure 2(b). Triples Method

1.3.3 The Triples Method

There is another approach to storing counts that may be more appropriate, depending on the fraction of the possible pairs of items that actually appear in some basket. We can store counts as triples [i, j, c], meaning that the count of pair {i, j}, with $i < j$, is c. A data structure, such as a hash table with i and j as the search key, is used so we can tell if there is a triple for a given i and j and, if so, to find it quickly. We call this approach the triples method of storing counts.

Unlike the triangular matrix, the triples method does not require us to store anything if the count for a pair is 0. On the other hand, the triples method requires us to store three integers, rather than one, for every pair that does appear in some basket. In addition, there is the space needed for the hash table or other data structure used to support efficient retrieval. The conclusion is that the triangular matrix will be better if at least 1/3 of the (n 2) possible pairs actually appear in some basket, while if significantly fewer than 1/3 of the possible pairs occur, we should consider using the triples method.
However, even if there were ten or a hundred times as many baskets, it would be normal for there to be a sufficiently uneven distribution of items that we might still be better off using the triples method. That is, some pairs would have very high counts, and the number of different pairs that occurred in one or more baskets would be much less than the theoretical maximum number of such pairs.

Comparing the two approaches

Approach 1: Triangular Matrix	**Approach 2: Triples Method**
▪ **Triangular Matrix** requires 4 bytes per pair ▪ Keep pair counts in lexicographic order ▪ Total number of pairs ***n*(*n* –1)/2**; total bytes= $\boldsymbol{2n^2}$	▪ uses **12 *bytes*** per occurring pair *(but only for pairs with count > 0)* ▪ Beats Approach 1 if less than **1/3** of possible pairs actually occur

1.4 **A-Priori Algorithm**

If we have enough main memory to count all pairs, using any method (triangular matrix or triples), then it is a simple matter to read the file of baskets in a single pass. For each basket, we use a double loop to generate all the pairs. Each time we generate a pair, we add 1 to its count. At the end, we examine all pairs to see which have counts that are equal to or greater than the support threshold s; these are the frequent pairs. However, this simple approach fails if there are too many pairs of items to count them all in main memory.

The A-Priori Algorithm is designed to reduce the number of pairs that must be counted, at the expense of performing two passes over data, rather than one pass.

1.4.1 The First Pass of A-Priori

In the first pass, we create two tables. The first table, if necessary, translates item names into integers from 1 to n, The other table is an array of counts; the ith array element counts the occurrences of the item numbered i. Initially, the counts for all the items are 0.As we read baskets, we look at each item in the basket and translate its name into an integer. Next, we use that integer to index into the array of counts, and we add 1 to the integer found there.

1.4.2 Between the Passes of A-Priori

After the first pass, we examine the counts of the items to determine which of them are frequent as singletons. It might appear surprising that many singletons are not frequent. But remember that we set the threshold s sufficiently high that we do not get too many frequent sets; a typical s would be 1% of the baskets. If we think about our own visits to a supermarket, we surely buy certain things more than 1% of the time: perhaps milk, bread, Coke or Pepsi, and so on. We can even believe that 1% of the customers buy diapers, even though we may not do so. However, many of the items on the shelves are surely not bought by 1% of the customers: Creamy Caesar Salad Dressing for example.

For the second pass of A-Priori, we create a new numbering from 1 to m for just the frequent items. This table is an array indexed 1 to n, and the entry for i is either 0, if item i is not frequent, or a unique integer in the range 1 to m if item i is frequent. We shall refer to this table as the frequent-items table.

1.4.3 The Second Pass of A-Priori

During the second pass, we count all the pairs that consist of two frequent items. A pair cannot be frequent unless both its members are frequent. Thus, we miss no frequent pairs. The space required on the second pass is $2m^2$ bytes, rather than $2n^2$ bytes, if we use the triangular matrix method for counting. Notice that the renumbering of just the frequent items is necessary if we are to use a triangular matrix of the right size. The complete set of main-memory structures used in the first and second passes is shown in Fig. 1.

Also notice that the benefit of eliminating infrequent items is amplified; if only half the items are frequent we need one quarter of the space to count. Likewise, if we use the triples method, we need to count only those pairs of two frequent items that occur in at least one basket.

The mechanics of the second pass are as follows:

1. For each basket, look in the frequent-items table to see which of its items are frequent.
2. In a double loop, generate all pairs of frequent items in that basket.
3. For each such pair, add one to its count in the data structure used to store counts.

Finally, at the end of the second pass, examine the structure of counts to determine which pairs are frequent.

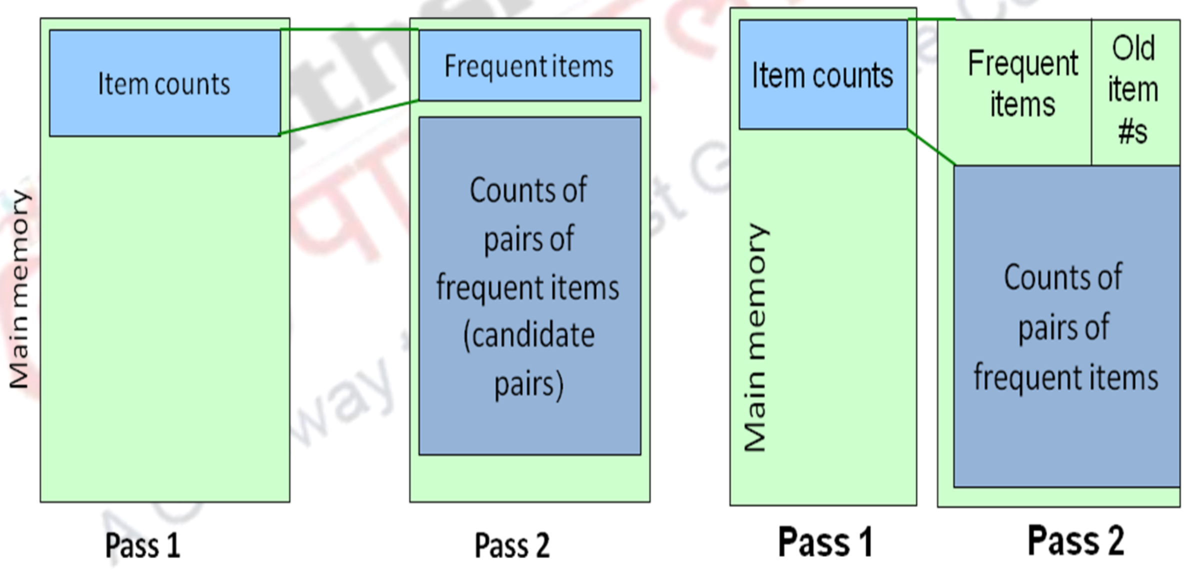


Figure 1: Schematic of main-memory use during the two passes of the A-Priori Algorithm

You can use the triangular matrix method with ***n*** = number of frequent items. May save space compared with storing triples by re-numbering the frequent items 1,2,… and keep a table relating new numbers to original item numbers

1.5 Handling Larger Datasets in Main Memory

The A-Priori Algorithm is fine as long as the step with the greatest requirement for main memory – typically the counting of the candidate pairs C2 – has enough memory that it can be accomplished without thrashing (repeated moving of data between disk and main memory). Several algorithms have been proposed to cut down on the size of candidate set C2. Here, we consider the PCY Algorithm, which takes advantage of the fact that in the first pass of A-Priori there is typically lots of main memory not needed for the counting of single items. Then we look at the Multistage Algorithm, which uses the PCY trick and also inserts extra passes to further reduce the size of C2.

1.6 Modified Apriori - Multistage Algorithm

In pass 1 of A-Priori, most memory is idle. We store only individual item counts. We use the idle memory to reduce memory required in pass 2. During Pass 1, in addition to item counts, maintain a hash table with as many buckets as fit in memory. Keep a count for each bucket into which pairs of items are hashed. For each bucket just keep the count, not the actual pairs that hash to the bucket.

```
FOR (each basket) :
        FOR (each item in the basket) :
                add 1 to item's count;
New  {             FOR (each pair of items) :
                          hash the pair to a bucket;
                          add 1 to the count for that bucket;
```

Few things to note in the above algorithm are: Pairs of items need to be generated from the input file; they are not present in the file and we are not just interested in the presence of a pair, but we need to see whether it is present at least ***s*** (support) times.

It can be observed that, if a bucket contains a frequent pair, then the bucket is surely frequent. However, even without any frequent pair, a bucket can still be frequent. So, we cannot use the hash to eliminate any member (pair) of a "frequent" bucket. But, for a bucket with total count less than *s*, none of its pairs can be frequent. Pairs that hash to this bucket can be eliminated as candidates (even if the pair consists of 2 frequent items). In pass 2, we count only pairs that hash to frequent buckets.

In between the passes, we replace the buckets by a bit-vector, in which, 1 means the bucket count exceeded the support *s* (call it a frequent bucket); 0 means it did not. 4-byte integer counts are replaced by bits, so the bit-vector requires 1/32 of memory. Also, decide which items are frequent and list them for the second pass.

In pass 2, we count all pairs *{i, j}* that meet the conditions for being a candidate pair:

1. Both *i* and *j* are frequent items
2. The pair *{i, j}* hashes to a bucket whose bit in the bit vector is 1 (i.e., a frequent bucket)

Both conditions are necessary for the pair to have a chance of being frequent. Both the passes explained above are shown in figure 2.

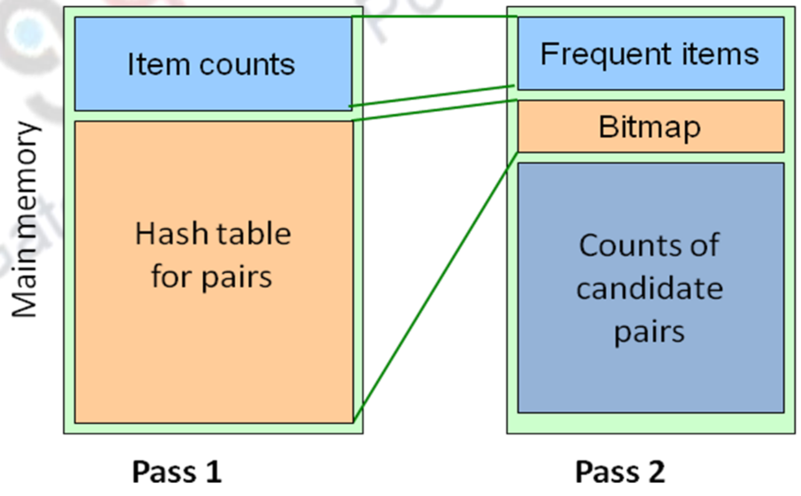
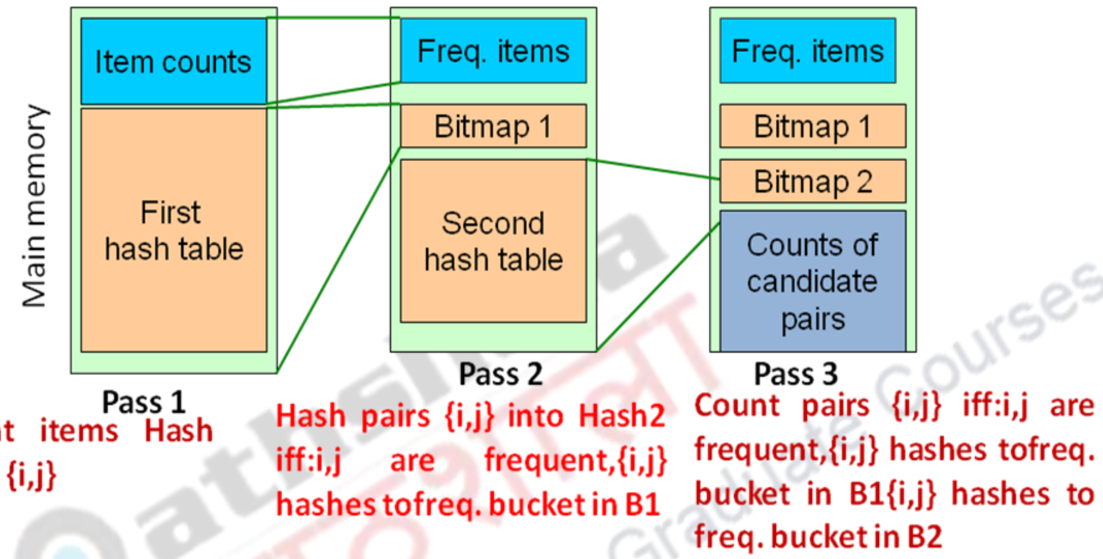


Figure2: Status of Main Memory for 2 passes of modified Apriori

In the main memory, buckets require a few bytes each buckets is O(main-memory size) provided we do not have to count past s. On second pass, a table of (item, item, count) triples is essential (we cannot use triangular matrix approach, why?)

Thus, hash table must eliminate approx.2/3 of the candidate pairs for PCY to beat A-Priori.

The Multistage Algorithm improves upon PCY by using several successive hash tables to reduce further the number of candidate pairs. The tradeoff is that Multistage takes more than two passes to find the frequent pairs. An outline of the Multistage Algorithm is shown in Fig. 3.



Main memory
Item counts
First hash table
Freq. items
Bitmap 1
Second hash table
Freq. items
Bitmap 1
Bitmap 2
Counts of candidate pairs
Pass 1
Pass 2
Pass 3
Count items Hash pairs {i,j}
Hash pairs {i,j} into Hash2 iff:i,j are frequent,{i,j} hashes tofreq. bucket in B1
Count pairs {i,j} iff:i,j are frequent,{i,j} hashes tofreq. bucket in B1{i,j} hashes to freq. bucket in B2

Figure 3: The Multistage Algorithm uses additional hash tables to reduce the number of candidate pairs

The Multistage Algorithm uses additional hash tables to reduce the number of candidate pairs. The first pass of Multistage is the same as the first pass of PCY. After that pass, the frequent buckets are identified and summarized by a bitmap, again the same as in PCY. But the second pass of Multistage does not count the candidate pairs. Rather, it uses the available main memory for another hash table, using another hash function. Since the bitmap from the first hash table takes up 1/32 of the available main memory, the second hash table has almost as many buckets as the first.On the second pass of Multistage, we again go through the file of baskets.There is no need to count the items again, since we have those counts from the first pass. However, we must retain the information about which items are frequent, since we need it on both the second and third passes. During the second pass, we hash certain pairs of items to buckets of the second hash table.A pair is hashed only if it meets the two criteria for being counted in the second pass of PCY; that is, we hash {i, j} if and only if i and j are both frequent,and the pair hashed to a frequent bucket on the first pass. As a result, the sum of the counts in the second hash table should be significantly less than the sum for the first pass. The result is that, even though the second hash table has only 31/32 of the number of buckets that the first table has, we expect there to be many fewer frequent buckets in the second hash table than in the first.

After the second pass, the second hash table is also summarized as a bitmap,and that bitmap is stored in main memory. The two bitmaps together take up slightly less than 1/16th of the available main memory, so there is still plenty of space to count the candidate pairs on the third pass. A pair {i, j} is in C2 if and only if:

1. i and j are both frequent items.
2. {i, j} hashed to a frequent bucket in the first hash table.
3. {i, j} hashed to a frequent bucket in the second hash table.

The third condition is the distinction between Multistage and PCY.It might be obvious that it is possible to insert any number of passes between the first and last in the multistage Algorithm. There is a limiting factor that each pass must store the bitmaps from each of the previous passes. Eventually, there is not enough space left in main memory to do the counts. No matter how many passes we use, the truly frequent pairs will always hash to a frequent bucket, so there is no way to avoid counting them.

1.7 The Multihash Algorithm

Sometimes, we can get most of the benefit of the extra passes of the Multistage Algorithm in a single pass. This variation of PCY is called the Multihash Algorithm. Instead of using two different hash tables on two successive passes,use two hash functions and two separate hash tables that share main memory on the first pass, as suggested by Fig. 1.3.The danger of using two hash tables on one pass is that each hash table has half as many buckets as the one large hash table of PCY. As long as the average count of a bucket for PCY is much lower than the support threshold, we can operate two half-sized hash tables and still expect most of the buckets of both hash tables to be infrequent. Thus, in this situation we might well choose the multihash approach.

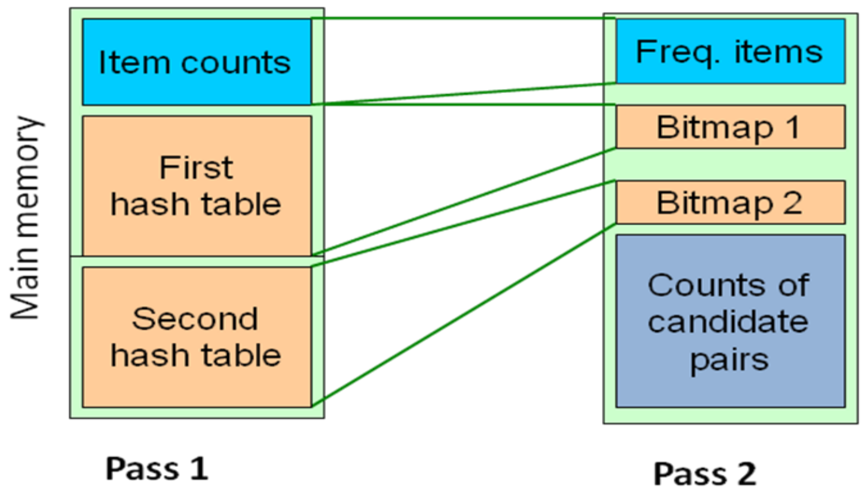


Figure 1.3: The Multihash Algorithm uses several hash tables in one pass

For the second pass of Multihash, each hash table is converted to a bitmap, as usual. Note that the two bitmaps for the two hash functions in Fig. 1.3 occupy exactly as much space as a single bitmap would for the second pass of the PCY Algorithm. The conditions for a pair {i, j} to be in C2, and thus
to require a count on the second pass, are the same as for the third pass of Multistage: i and j must both be frequent, and the pair must have hashed to a frequent bucket according to both hash tables.

Just as Multistage is not limited to two hash tables, we can divide the available main memory into as many hash tables as we like on the first pass of Multihash. The risk is that should we use too many hash tables, the average count for a bucket will exceed the support threshold. At that point, there may
be very few infrequent buckets in any of the hash tables. Even though a pair must hash to a frequent bucket in every hash table to be counted, we may find that the probability an infrequent pair will be a candidate rises, rather than falls, if we add another hash table.

2. **Other approaches**

2.1 Approach 1: Take a **random sample** of the market baskets. Run a-priori or one of its improvements in main memory So we don't pay for disk I/O each time we increase the size of itemsets (or) reduce support threshold proportionally to match the sample size. Optionally, verify that the candidate pairs are truly frequent in the entire data set by a second pass (avoid false positives). But you don’t catch sets frequent in the whole but not in the sample. We can fix a smaller threshold, e.g., *s*/125, helps catch more truly frequent itemsets, but it requires more space.

2.2 Approach 2: Repeatedly read small subsets of the baskets into main memory and run an in-memory algorithm to find all frequent itemsets. Note: we are not sampling, but processing the entire file in memory-sized chunks. An itemset becomes a candidate if it is found to be frequent in any one or more subsets of the baskets. On a second pass, count all the candidate itemsets and determine which are frequent in the entire set. Key “monotonicity” idea: an itemset cannot be frequent in the entire set of baskets unless it is frequent in at least one subset.

2.3 Approach 3 - Distributed Version: SON lends itself to distributed data mining, where the baskets are distributed among many nodes. Then compute the frequent itemsets at each node. Distribute the candidates to all nodes and finally accumulate the counts of all candidates.

Case Studies

A) Novel Approach based on Improving Apriori Algorithm and Frequent Pattern Algorithm for Mining Association Rule

The effectiveness of mining association rules is a significant field of Knowledge Discovery in Databases (KDD). The Apriori algorithm is a classical algorithm in mining association rules. This paper presents an improved method for Apriori and Frequent Pattern algorithms to increase the efficiency of generating association rules. This algorithm adopts a new method to decrease the redundant generation of sub-itemsets during pruning the candidate itemsets, which can form directly the set of frequent itemset and remove candidates having a subset that is not frequent in the meantime. This algorithm can raise the probability of obtaining information in scanning database and reduce the potential scale of itemsets

B) An Adaptive Implementation Case Study of Apriori Algorithm for a Retail Scenario in a Cloud Environment

Retail transactional databases are voluminous and traditional algorithmic approaches to mine pattern in them are time consuming. The current study presents an approach to scale Apriori a Frequent Item set Mining (FIM) algorithm, which is often used for market basket analysis. The study also compares the performance of scaled version of the algorithm (running on multiple on-demand simultaneous Azure cloud instances) with that of traditional setup (running on a fixed Azure cloud instances) using simulated data sets. The experimental results show that the response times were significantly lower and in favor of the scaled approach as the data volume increases..

Summary

- Apriori approach has its limitations
- Extensions to apriori are explored for handling various tincluding its scalabilty