Chapter 3

Frequent Itemsets in Static Databases

Association rule was introduced by Agarwal et al. in 1993 [AIS93]. Basically, association rule finds influence of one set of items/objects on another set of items/objects. One example of association rule may be of the form “80% of the customers who buy bread also buy butter”. Association rules have found numerous applications in real world such as decision support, understanding customer behavior, telecommunication alarm diagnosis, prediction, etc. Departmental stores also can use association rules in many fields such as catalog design, add-on sales, store layout, etc. The terms used in connection with association rule mining are itemset, frequent/large itemset, support, confidence, etc.

This chapter reports some of the existing popular frequent itemsets finding algorithms and presents a detailed analysis of these algorithms in terms of their efficiency in execution time and storage utilization. It also presents two enhanced versions of Apriori [AMS+94], which overcomes (i) too many candidate generations and (ii) execution time bottleneck due to huge repository of market-basket databases. The following section gives some basic concepts of association rule mining techniques.
3.1 Basic Concepts

Basic concept of association rule is best explained in [AMS+96]. Let $I$ be a set of items in a super market and $D$ be a database of customers' transactions, where each transaction $t$ is a set of items such that $t \subseteq I$. $I$ also can be considered as a set of attributes over the binary domain $\{1, 0\}$. In that case, one transaction will be a string of 0's and 1's, where 1 represents that corresponding item has been bought and 0 represents that corresponding item has not been bought. A set of items $X \subseteq I$ is called an itemset. Support of an itemset $X$, denoted by $\text{Sup}(X)$, is defined as the percentage of transactions in $D$, that contain $X$. An itemset with support greater than a pre-defined value (called minimum support) is called frequent or large itemset. An association rule between two frequent itemsets $X, Y$ (denoted by $X \Rightarrow Y$) may exist with support $s$ and confidence $\text{conf}$, if $X \cap Y = \emptyset$. Support of $X \Rightarrow Y$ is the $\text{Sup}(X \cup Y)$. The confidence of $X \Rightarrow Y$ is defined as the percentage of transactions in $D$ that contain $X$, also contain $Y$, and is calculated as $\text{Sup}(X \cup Y)/\text{Sup}(X)$.

Example 3.1

Let us consider a database $D$ (Table 3.1) with 5 items $(A, B, Q, R, S)$ and 5 transactions. There are 31 possible itemsets. Some of them are $\{A\}, \{B\}, \{Q\}, \{AB\}, \{ABQ\}$, etc. Support of $\{A\}$ is 60% because it has occurred in 3 transactions. Similarly, support of $\{B\}, \{Q\}, \{AB\}, \{ABQ\}$ are 80%, 60%, 60%, and 40% respectively.

<table>
<thead>
<tr>
<th>TID</th>
<th>A</th>
<th>B</th>
<th>Q</th>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>300</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>400</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.1: A Sample Database - I

Suppose, minimum support is 30%. Then, all the above mentioned itemsets will be frequent, because support of all of them are at least 30%. It can be observed
that $AB$ has occurred in 3 transactions and $Q$ has occurred in 2 transactions out of those 3 transactions. So, association rule $AB \Rightarrow Q$ exists with confidence 66.6% and support 40%. Similarly, $A \Rightarrow B$ exists with support 60% and confidence 100%.

Given a database $D$, the problem of mining association rules is to generate the association rules that have certain pre-defined minimum support and confidence. The problem of association rule mining can be divided into two subproblems.

1. Find all the frequent itemsets in the given database.

2. Find the association rules using the large itemsets found in the first step.
   As for example, suppose $ABQR$ and $AB$ are large itemsets. Then the confidence of the rule $AB \Rightarrow QR$ is calculated as $\text{conf} = \frac{\text{Sup}(ABQR)}{\text{Sup}(AB)}$. If $\text{conf} \geq \text{minconf}$, then the rule $AB \Rightarrow QR$ is said to exist with confidence $\text{conf}$.

Out of these two steps, the first step is important and difficult one. That's why, most of the algorithms concentrate on finding the frequent itemsets from a large database in minimum possible time and using minimum resources. Once large itemsets are known, finding association rules are straightforward. Next section discusses some popular algorithms to find frequent itemsets.

### 3.2 Some Existing Algorithms

There are many algorithms to find frequent itemsets. However, different algorithms target different types of database. Here, three basic algorithms for frequent itemset generations i.e. Apriori, AprioriTid and AprioriHybrid [AMS+94] have been chosen and their performance have been analyzed.

#### 3.2.1 Apriori

Among the popular algorithms to find the large itemsets, this algorithm stands at the top because of its simplicity and effectiveness. The algorithm is based on
the fact that all subsets of a frequent itemset are also frequent. The algorithm first makes one pass over the database and finds the large items. Then, the algorithm makes many passes over the data. Each pass starts with the seed set of large itemsets which are used to generate new potentially large itemsets called candidate itemsets. Then, support of each candidate itemset is found during the pass over the data and the actual large itemsets are determined. These large itemsets become the seed for the next pass. This process continues till large itemsets can be found.

Let c[1], c[2], c[3], ..., c[k] be a k-itemset stored in the lexicographic order i.e. c[i] < c[i + 1] (i=1,.., k-1). Let L_k be a set of large k-itemsets with two fields : itemset and the support count; C_k be a set of candidate k-itemsets with two fields : itemset and the support count; D be the database of transactions; minsup and minconf be the minimum support and minimum confidence. With these symbols, the algorithm is given in Algorithm 3.1.

```
Input : D, minsup
Output : All frequent/large itemsets

1. L_1 = {Large 1-itemsets};
2. For(k=2; L_{k-1} \neq \phi; k++) do {
3. \hspace{1cm} C_k = Apriori-gen(L_{k-1}); // New candidates generation
4. \hspace{1cm} For all transactions t \in D do {
5. \hspace{2cm} C(t) = Subset(C_k, t); // Candidates contained in t
6. \hspace{2cm} For all candidates c in C(t) do
7. \hspace{3cm} c.count++;
8. \hspace{1cm} L_k = \{c \in C_k | c.count \geq minsup\};
9. \hspace{1cm} }
10. \hspace{1cm} Return all large itemsets = \bigcup L_k;
```

Algorithm 3.1: Apriori
Candidate Generation and Pruning

The algorithm uses the function \textit{Apriori-gen} to generate candidate sets. The function takes set of all large \(k-1\) itemsets \((L_{k-1})\) as input and produces the candidate \(k\)-itemsets \((C_k)\). The function generates a candidate \(k\)-itemset from two \(k-1\) frequent itemsets with the same first \(k-2\) items. If \(a\) and \(b\) are two large \(k-1\) itemsets such that first \(k-2\) items of the itemsets are same, then the algorithm of the function can be described as follows.

1. Insert \(a \cup b\) into \(C_k\).

2. Select \(a[1], a[2], ..., a[k-1], b[k-1]\) from \(a\) and \(b\) where \(a[i] = b[i](i = 1, ..., k-2)\) and \(a[k-1] < b[k-1]\) i.e., \(a \cup b\) will be inserted into \(C_k\) if \(a\) and \(b\) have first \(k-2\) items in common and \(a[k-1] < b[k-1]\).

As for example, let \(\{ABQR\}\) and \(\{ABQS\}\) be two frequent 4-itemsets such that first 3 items of the itemsets are same. These two itemsets can be combined to form a candidate 5-itemset \(\{ABQRS\}\) (i.e. \(\{ABQ\} \cup \{R\} \cup \{S\}\)). However, \(\{ABQ\}\) and \(\{ARS\}\) cannot be combined, because only first items are same.

Pruning is one important step in the algorithm because the algorithm may generate a lot of redundant candidate sets. The basic principle of \textit{Apriori} is that if an itemset is frequent, then all the subsets will also be frequent. So, any candidate itemset, with a subset which is not frequent, is pruned away. Thus, it reduces number of candidate itemsets to a great extent.

\textbf{Example 3.2}

Let us again consider the \textit{Table} 3.1 on page 40 and take minimum support as 60\% (3 transactions). If \textit{Apriori} is run, results are obtained as given in \textit{Figure} 3.1 on the next page. The item \(R\) is not included in \(L_1\) because \(\text{Sup}(R)\) is less than minimum support(3).
3.2.2 AprioriTid

This algorithm is a modification of the Apriori algorithm. This algorithm also generates the candidates using the same generating function Apriori-gen as in the Apriori. The main feature of the algorithm is that the original database is not used after the first pass. Instead of that, a data structure $C'_k$ is used. Each member of the set $C'_k$ is of the form $<TID, \{X_k\}>$, where $X_k$ is a potentially large $k$-itemset present in the transaction with the identifier $TID$. For $k=1$, $C'_k$ is the database itself with each item $i$ is replaced by itemset $\{i\}$. For $k>1$, the member of $C'_k$ corresponding to a transaction $t$ is $<t.TID, \{c \in C_k|c \text{ contained in } t\}>$. If a transaction does not contain any candidate set, then $C'_k$ will not have any entry for that transaction. So, number of entries in $C'_k$ gets reduced in the successive passes resulting in fewer transactions to be scanned in each subsequent passes. One shortcoming of the algorithm is the creation and updating of $C'_k$, which takes considerable amount of execution time. The algorithm is given in
Algorithm 3.2.

Example 3.3

Let us apply the algorithm on the same sample database of Table 3.1 on page 40 with minimum support 60% (3 transactions). The result is given in Figure 3.2.

<table>
<thead>
<tr>
<th>Input: $D$, $minsup$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: All frequent/large itemsets.</td>
</tr>
</tbody>
</table>

1. $L_1 = \{\text{large 1-itemsets} \}$;
2. $C'_1 = \text{database } D$;
3. For $(k = 2; L_{k-1} \neq \phi; k++)$ do begin
4. \hspace{1em} $C_k = \text{Apriori-gen}(L_{k-1});$ //New candidates
5. \hspace{1em} $C_k' = \phi$;
6. \hspace{1em} For all entries $t \in C_{k-1}'$ do begin
7. \hspace{2em} //determine candidates contained in the transaction $t.TID$
   \hspace{2em} $C(t) = \{c \in C_k | (c[1], c[2], \ldots, c[k-1]) \in t.set.of.itemsets \land (c[1], c[2], \ldots, c[k-2], c[k]) \in t.set.of.itemsets\};$
8. \hspace{2em} For all candidates $c \in C(t)$ do
9. \hspace{3em} $c.count++$;
10. \hspace{3em} If $(C(t) \neq \phi)$ then $C_k' += <t.TID, C(t)>;$
11. \hspace{1em} End
12. \hspace{1em} $L_k = \{c \in C_k | c.count \geq minsup\};$
13. \hspace{1em} End
14. \hspace{1em} Answer $= \bigcup_k L_k$;

Algorithm 3.2: AprioriTid
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Transaction with TID 300 has been dropped from $C'_2$ onwards because the transaction does not contain any more frequent itemset. Thus, number of transactions are reduced which results in reduction in execution time.

Figure 3.2: Example of AprioriTid Algorithm
3.2.3 \textit{AprioriHybrid}

The algorithm is basically a fusion of \textit{Apriori} and \textit{AprioriTid}. It uses \textit{Apriori} for the first few passes and \textit{AprioriTid} for the remaining passes based on some threshold values i.e. when it is found that candidates can be stored in memory, it uses \textit{AprioriTid}. It uses good characteristics of both the algorithms and superior to both the algorithms. It has been found that \textit{AprioriHybrid} is better than \textit{Apriori} by 30\% and \textit{AprioriTid} by 60\% [AMS+96].

3.2.4 Experimental Results

Experiments were carried out to compare the performance of \textit{Apriori}, \textit{AprioriTid} and \textit{AprioriHybrid}. Experiments were carried out on a Pentium IV PC with 256 MB RAM. The experimental results are given in Figures 3.3 on the following page and 3.4 on page 49.

\textit{Data Sources}: Two synthetic databases (T20.I4.D100K and T20.I6.D100K) and one real database \textit{Connect4}, publicly available in UCI machine learning repository (http://www.ics.uci.edu/mlearn/mlrepository.html) have been chosen for the experiments. The synthetic databases were generated using the technique given in [AMS+96] with the value of parameters as given in Table 3.2 on the following page and with dimensionality 500. Here, \(|T|\), \(|ML|\) and \(|D|\) represent the average size of a transaction, mean size of a potentially large itemset and database size (number of transactions) respectively. Each of the synthetic datasets contains 100K records. \textit{Connect-4} database contains around 65K instances with 43 attributes. Each attribute can have one of three values. Each distinct value of the attributes has been considered as one item, resulting in total 129 items.
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

| Name           | $|T|$ | $|ML|$ | $|D|$ |
|----------------|-----|-------|------|
| T20.14.D100K   | 20  | 4     | 100K |
| T20.16.D100K   | 20  | 6     | 100K |

Table 3.2: Parameters for Synthetic Databases - I

Figure 3.3: Experimental Results of Apriori, AprioriTid & AprioriHybrid - I
Figure 3.4: Experimental Results of Apriori, AprioriTid & AprioriHybrid - II
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

3.2.5 Discussion

The algorithm Apriori is very easy to implement and finds all possible frequent itemsets. However, it suffers from some serious drawbacks, which increase the execution time.

- It makes several passes over the databases, which can be found to be very expensive for large databases.
- Though it prunes away some candidates, yet it generates a lot of unnecessary candidates. So, for a database with large number of items, it can be found too expensive from storage as well as execution time point of view.

AprioriTid is also robust enough to find all the frequent itemsets. It differs from Apriori in the sense that it scans the database once and uses a better data structure for the rest of iterations. The size of the data structure is smaller than that of original database, which is the main advantage of the algorithm. However, it also suffers from the problems of Apriori. In addition to that, some extra memory and extra disk space are required for the data structure. Some extra time also spent to maintain the data structure.

AprioriHybrid uses the benefits of both the algorithms. That is why, it is the best algorithm among all the three algorithms. However, it also suffers from the problems of both the algorithms.

The basic problem of all these algorithms is the ratio of number of candidate sets to frequent sets. Based on exhaustive experiments in the light of several real and synthetic datasets, it has been observed that, on the average, this ratio is as high as 50:1. So, some new techniques are required to address this problem to bring down this ratio to as low as possible.

It can be observed from the experimental results that AprioriHybrid is the best algorithm in terms of execution time among the three algorithms. Poor performance of AprioriTid can be attributed to two possible reasons: one is that it could not reduce the database size and the other is that it took maximum time to maintain the data structure. Apriori performed better than AprioriTid and worse than AprioriHybrid. The reason is that it has to make a pass over
the database in every iteration and it does not employ any database reduction technique.

As mentioned above, main disadvantage of all these algorithms is the generation of too many candidate sets. Some possible solutions of this problem are given below.

- **Using hash based techniques**: Hash based techniques has been found to be useful in reducing the redundant candidate sets. Some hash-based algorithms can be found in [PCY95a]. These techniques work well, when the hash tables are small and can be kept in memory. However, these algorithms suffer from the drawbacks of having to create hash tables and maintaining them.

- **Using sample**: Sampling is also an useful technique to reduce the database size and number of candidate sets. Use of sampling technique to find frequent itemsets can be found in [LCK98]. The idea is to pick a random sample, use it to determine all association rules in the sample. However, there is a trade off between accuracy and efficiency.

- **Using efficient data structure**: Performance of the algorithms for frequent itemsets generation can be improved to a great extent using efficient data structures. One such efficient data structure is FP-tree [HPY00], which has been used in *FP-growth* algorithm [HPY00].

- **Using probability**: Probability is one useful technique, which can be used to predict whether an itemset will be frequent or not. If the probability of a candidate set is high enough, then it can be kept for calculating its frequency count. Otherwise, it can be rejected with least risk.

Next section presents two enhanced versions of *Apriori* to address the problem of too many candidate generations and for faster implementation.

### 3.3 The Modified *Apriori* Algorithm

It has been observed that *Apriori* algorithm is robust enough to find all frequent itemsets from a database. However, it suffers from two major drawbacks.
1. It is not scalable.

2. It generates huge number of candidate sets, out of which only a few are actually large.

In Modified_Apriori, Boole's inequality [GK84] has been used to reduce the number of candidate itemsets. The main part of the algorithm is same as Apriori. The point by which it differs from the Apriori algorithm is the use of the Boole's inequality in generating the candidate itemsets. The heart of the algorithm is the function Comp_Apriori_gen, which computes the candidate sets reasonably by exploiting the inequality (reported in the next subsection).

3.3.1 Background

The main objective of the Modified_Apriori is to reduce the number of unnecessary candidate sets in the frequent itemset generation by exploiting the Boole's inequality.

Boole's inequality: For $k$ events $A_i (i=1,2,3,...k)$,

$$ P(\bigcap_{i=1}^{k} A_i) \geq \sum_{i=1}^{k} P(A_i) - (k - 1) $$

The above inequality can be found very useful in the generation of candidate sets reasonably. As for example, let us consider an itemset $X = \{A, B, Q\}$. The support of $X$ can be calculated as the probability of occurrence of the items in $X$ together in the database. So, support of $X = \frac{\text{Number of records containing } X}{\text{Total number of records}}$. Using Boole's inequality, minimum support of $X$ (in %) can be estimated as $P(A)+P(B)+P(Q) - 2$. Obviously, this is not the actual support. It just gives an representative value of the support of the itemset. Here, probabilities of individual items i.e. $P(A), P(B)$ and $P(Q)$ are required, which can be calculated in the first pass of the Apriori.

Using Boole's inequality, representative values of minimum supports of the candidate itemsets can be calculated easily. Candidate itemsets with representative
values greater than some threshold, will have high probability of being frequent, which can be used as an additional pruning step. However, calculating the threshold is not an easy task. The optimum threshold is the minimum support itself. However, it is too high. Very few candidate itemsets will cross that threshold. So, the threshold is calculated as

\[ k \cdot \text{minsup} - (k - 1) + \frac{(k - 1)(1 - \text{minsup})}{\beta} \]

where \( k \) is the size of the itemset and \( \beta > 1 \).

### 3.3.2 The Algorithm

The algorithm is given in Algorithm 3.3 on the next page. Most of the symbols used in the algorithm are same as those of Apriori. One additional symbol used here is \( PA \), which is an array to store the probabilities of occurrences of each large item in a transaction. Although the algorithm is similar to Apriori, yet the first line needs a little explanation. The first line of the algorithm finds the large 1-itemsets and calculates the probability of occurrences of each large item \( i \), i.e. \( PA[i] \), which is calculated as

\[ PA[i] = \frac{\text{Number of records containing } i}{|D|} \]

### 3.3.3 Candidate Generation

Candidates are generated by the function \( \text{Comp.Apriori.gen} \), which uses the Boole's inequality to generate candidate sets. The function \( \text{Comp.Apriori.gen} \) is given in Function 3.1 on page 55, which takes in \( L_{k-1} \) and returns \( C_k \). The union of itemsets \( a, b \in L_{k-1} \) (i.e. \( a \cup b \)) is inserted in \( C_k \) if they share their first \( k-2 \) items and the value of the Boole's inequality of the items in \( a \cup b \) is at least

\[ k \cdot \text{minsup} - (k - 1) + \frac{(k - 1)(1 - \text{minsup})}{\beta} \]

The rationale behind calculating the Boole's inequality is that if the value is very low, there is little chance that the itemset will be frequent. So, these itemsets can
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Input: $D, \text{minsup}, \beta$
Output: Frequent/large itemsets

1. $L_1 = \text{Large 1-item sets and calculate } PA[i]$;
2. For ($k = 2; L_{k-1} \neq \emptyset; k++$) do {
3. $C_k = \text{Comp.Apriori.gen}(L_{k-1})$;
4. For all transaction $t \in D$ do {
5. $C(t) = \text{subset}(C_k, t)$;
6. For all candidates $c \in C(t)$ do
7. $c.\text{count}++$; } }
8. $L_k = \{c \in C_k | c.\text{count} \geq \text{minsup}\};$
9. Return the set of all large itemsets $= \bigcup_k L_k$;

Algorithm 3.3: Modified_Apriori

easily be rejected without much overhead, resulting in smaller size of candidate itemsets. The algorithm generates candidate itemsets with the probability of being frequent very high. Thus, it reduces the difference between number of candidate itemsets and that of actual large itemsets to a great extent, resulting in significant reduction of the I/O overhead.

Selecting $\beta$ : Here is a very simple technique to choose the value of $\beta$. The algorithm is run with a small random sample of the database and an initial value of $\beta$. If the result is not satisfactory, $\beta$ is tuned according to requirement and the algorithm is rerun. The process is continued several times to obtain optimum results. When optimum results are obtained, the corresponding value of $\beta$ can be used for the entire database. As a guideline to choose the values of $\beta$, it has been observed that values of $\beta$ within the range of 3 to 6 give better result.
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Function $\text{Comp}_\text{Apriori}.\text{gen}(L_{k-1})$

1. $C_k = \emptyset$;
2. $x = k \times \text{minsup} - (k - 1) + \frac{(k - 1)(1 - \text{minsup})}{\beta}$
3. For all $a, b \in L_{k-1}$
4. Set $\text{Temp}$ to 0;
5. For ($i = 1$ ; $i = k-2$; $i++$) do
6. $\text{Temp} = \text{Temp} + PA[a[i]]$;
7. $\text{Temp} = \text{Temp} + PA[a[k-1]] + PA[b[k-1]]$;
8. Insert $a \cup b$ into $C_k$, if $a[i] = b[i]$ ($i = 1, 2, ... k-2$) and $a[k-1] < b[k-1]$ and $\text{Temp} \geq x$;
9. End for
10. Return $C_k$;

Function 3.1: $\text{Comp}_\text{Apriori}.\text{gen}$

3.3.4 Experimental Evaluation

Experiments were carried out for performance comparison of the proposed $\text{Modified}_\text{Apriori}$ with $\text{Apriori}$, $\text{AprioriTid}$ and $\text{AprioriHybrid}$. Experiments were carried out on a Pentium IV PC with 256MB RAM. The method discussed above has been used to find value of $\beta$. Average results of the experiments are reported in Figures 3.5 through 3.13.

Data set: Two synthetic databases (T20.I4.D100K and T20.I6.D100K) and one real database $\text{Connect4}$ (http://www.ics.uci.edu/mllearn/mlrepository.html) were used for the experiments. The synthetic databases were generated using the technique given in [AMS+96] with the parameters given in Table 3.3 on the next page and with dimensionality 500. Here, $|T|$, $|ML|$ and $|D|$ represent the average size of a transaction, mean size of a potentially large itemset and
number of transactions respectively. Each of the synthetic datasets contains 100K records. Connect-4 database contains around 65K instances with 43 attributes. Each attribute can have one of three values. Here also, each distinct value of the attributes has been considered as one item, resulting in total 129 items.

**Observations**: It can be seen from the results (Figures 3.6 on the following page, 3.9 on page 59 and 3.12 on page 60) that Apriori, AprioriTid and AprioriHybrid generate huge volume of candidate sets, which increases the I/O overhead and execution time. On the other hand, Modified_Apriori, as can be seen in the results, generates much less number of candidates. As far as frequent sets (Figures 3.7 on page 58, 3.10 on page 59 and 3.13 on page 61) are concerned, Modified_Apriori finds almost same number of frequent itemsets as that of other algorithms. Figures 3.5 on the following page, 3.8 on page 58 and 3.11 on page 60 show the execution time comparison of the four algorithms. It can be observed that Modified_Apriori has taken much less time than that of other algorithms. This can be attributed to the fact that Modified_Apriori generates and has to process less number of candidates than that of other algorithms. Another point to be observed is that frequent itemsets in Modified_Apriori is a subset of those of Apriori, AprioriTid and AprioriHybrid.

| Name            | |T| | ML| | D |
|-----------------|---|---|---|---|
| T20.I4.D100K    | 20 | 4 | 100K |
| T20.I6.D100K    | 20 | 6 | 100K |

Table 3.3: Parameters for Synthetic Databases - II
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Figure 3.5: Experimental Results of Modified Apriori - I

Figure 3.6: Experimental Results of Modified Apriori - II
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Figure 3.7: Experimental Results of Modified_Apriori - III

Figure 3.8: Experimental Results of Modified_Apriori - IV
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Figure 3.9: Experimental Results of Modified_Apriori - V

Figure 3.10: Experimental Results of Modified_Apriori - VI
Figure 3.11: Experimental Results of Modified_Apriori - VII

Figure 3.12: Experimental Results of Modified_Apriori - VIII
3.3.5 Using Multiplication Law of Probability to Generate Candidate Sets

*Multiplication law of probability* was also used to reduce unnecessary candidate itemsets in *Apriori* [DB04]. The multiplication law of probability states that the probability of occurring independent events together is equal to the product of the probabilities of occurring of the events individually [GK84]. Support of an itemset is the probability of occurrence of the itemset in the database. If it is assumed that items are independent in the database, estimated support of a candidate itemset can be calculated as the product of the probabilities (supports) of the items in the itemset. If this probability(support) is greater than some threshold, the candidate can be kept for further processing, otherwise it can be pruned away. Thus, unnecessary candidates are reduced to a great extent in *Apriori*. However, this approach can be found to be suitable in those cases, where faster frequent itemsets is essential and all the frequent itemsets may not be required to be generated.
3.4 Using Bitmaps to Find Frequent Itemsets

Bitmap techniques have been applied in various application domains [BAG99, Gra94, JDO99, Joh98, MZ98, NG95]. Bitmap indexing method is popular in OLAP products because it allows quick searching in data cubes. In the bitmap index for a given attribute, there is a distinct bit vector for each value in the domain of the attribute. If the domain of the attribute consist of $n$ values, then $n$ bits are needed for each entry in the bitmap index. If the attribute has the value $V$ for a given row in the table, then the bit representing the value is set to 1 in the corresponding row of the bitmap index. All the other bits for that row are set to 0. As for example, let us consider the database (Table 3.4). The corresponding bitmap entries of the table are shown in Table 3.5.

<table>
<thead>
<tr>
<th>Item</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>V</td>
</tr>
<tr>
<td>B</td>
<td>V</td>
</tr>
<tr>
<td>A</td>
<td>V</td>
</tr>
<tr>
<td>B</td>
<td>V</td>
</tr>
</tbody>
</table>

Table 3.4: A Sample Database - II

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.5: Bitmaps for The Item Attribute Values

Bitmap indexing is very useful because bit arithmetic is very fast. Bitmap indexing leads to significant reduction in space and I/O since a string character can be represented by a single bit. However, a problem with using bitmap index for a column with high cardinality is its high storage cost and potentially high expression evaluation cost. That is why compression of bitmap is required. The
form of compression is most crucial aspect since it must be designed to save disk space and perform the basic operations AND, OR, NOT, etc. A considerable amount of work has been done in the study of bitmap index compression. The use of bitmap compression has several potential performance advantages such as less storage requirement, faster accession as well as easy caching in the memory, etc. There are several good representation and compressions techniques such as Verbatim, Run Length Encoding, Gzip, Expgol, BBC, etc. [AYJ00]. Bitmap technique also has been successfully utilized in association rule mining [LL00, MZ98]. 

*Bit_AssocRule* [HLL03] is one of the promising association rule mining techniques designed using similar concept. Next section reports the algorithm.

### 3.4.1 Bit_AssocRule Algorithm

A detailed discussion of the algorithm can be found in [HLL03]. The algorithm uses bitmaps of the attribute values to find the frequent itemsets. The algorithm works as follows. The algorithm starts with the list $L_1$, which contains the large attribute values and bitmap of large 1-itemset. The $k$-candidates consist of $k$ attribute values from $k$ attributes. The candidate is large itemset if the bit count (i.e. the number of 1's) of the intersection of all the bitmaps in the candidate is equal to or greater than the minimum count. During each cycle, combination of length $k$ i.e. $k$-candidates are generated. At the end of the cycle if any $k$-candidate becomes large, new candidate of length $k + 1$ is generated in the next cycle. The cycle stops when no $k$ candidates are found to be large itemsets or if no new $k+1$ candidates can be generated. $k$-candidate is generated by joining a $k - 1$ itemset with 1-itemset in $L_1$ that has an attribute index greater than all attribute index of elements in that $k - 1$ itemset. The algorithm is given in Algorithm 3.4 on the next page.
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Algorithm 3.4: BitAssocRule

Example 3.4

Let us consider the database $D$ (Table 3.1 on page 40) and take minimum support as 60% (3 transactions). If BitAssocRule is run in the database, the result is obtained as given in Figure 3.14 on the next page.

Discussion

The algorithm works much faster than Apriori, AprioriTid and AprioriHybrid to find the frequent itemsets because it finds the support count of the candidate sets by intersection of the bitmaps of the items. However, experiments show that it generates a huge number of unnecessary candidate sets. The number of candidate sets can be reduced to a great extent by using Boole’s inequality in the same way as has been used in Modified_Apriori. The following algorithm is a modification of BitAssocRule algorithm, which uses Boole’s inequality to generate the candidate sets.
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

### Frequent Itemsets in Static Databases

<table>
<thead>
<tr>
<th>( L_1 )</th>
<th>( C_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AB</th>
<th>AQ</th>
<th>AS</th>
<th>BQ</th>
<th>BS</th>
<th>QS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( L_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( L_1 )</th>
<th>( C_3 )</th>
<th>( L_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>S</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ABS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Answer = \{A, B, Q, S, AB, AS, BS, ABS\}

Figure 3.14: Example of BitAssocRule Algorithm

#### 3.4.2 Modified_Bit_AssocRule Algorithm

The main purpose of Modified_Bit_AssocRule is to reduce the number of candidate sets by using Boole’s inequality in the same way as has been used in Modified_Apriori algorithm. The algorithm uses an additional array \( PA \) to store
the probability of each attribute value. $PA[i]$ is calculated as (bitmap count of $i^{th}$ item)/$|D|$. The algorithm is given in Algorithm 3.5. Candidates are generated by the function Bit-gen, which takes $L_1, L_{k-1}$ as input and returns $C_k$ as output.

```
Input: D, minsup, bitmaps.
Output: Frequent/large itemsets.

1. $L_1$ = Large 1-itemsets with their bitmaps;
2. Calculate $PA[i] (i=1,2,3...m)$;
3. For $(k=2; L_{k-1} \neq \phi; k++)$
   4. Remove those 1-itemsets in $L_1$ which are not included in any itemset of $L_{k-1}$;
5. $C_k$ = Bit-gen($L_{k-1}, L_1$);
6. $L_k$ = \{c $\in C_k$ | bitmap count of c $\geq$ minsup \};
7. Endfor
8. Answer = $\bigcup L_k$;
```

Algorithm 3.5: Modified_Bit_AssocRule
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Function $\text{Bit-gen}(L_{k-1}, L_1)$

1. Select $a \in L_{k-1}$ and $b \in L_1$ such that $b$ is larger than all the elements in $a$;
2. 
   \[ x = k \cdot \text{minsup} - (k - 1) + \frac{(k - 1)(1 - \text{minsup})}{\beta} \]
3. $\text{Temp} = 0$
4. for ($i = 1$; $i = k - 1$; $i++$)
5. \[ \text{Temp} = \text{Temp} + \text{PA}[a[i]] \]
6. $\text{Temp} = \text{Temp} + \text{PA}[b]$
7. Insert $a \cup b$ into $C_k$ if $\text{Temp} \geq x$
8. Return $C_k$

Function 3.2: $\text{Bit-gen}$

Experimental Results

To evaluate the performance of $\text{Modified.Bit.AssocRule}$ over $\text{Bit.AssocRule}$, experiments were carried out on the same environment as $\text{Modified.Apriori}$ algorithm. Here also, same datasets ($T20.I4.D100K$, $T20.I6.D100K$ and $\text{Connect4}$) were used. Other parameters were also kept same as those of $\text{Modified.Apriori}$. The experimental results are given in Figures 3.15 through 3.19.
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Figure 3.15: Experimental Results of Modified Bit_AssocRule - I
Figure 3.16: Experimental Results of Modified_Bit_AssocRule - II
Figure 3.17: Experimental Results of Modified_Bit_AssocRule - III
CHAPTER 3. FREQUENT ITEMSETS IN STATIC DATABASES

Figure 3.18: Experimental Results of Modified_Bit_AssocRule - IV
Observations: It can be observed from the results that Bit_AssocRule generates a large number of candidate sets, which increases the I/O overhead and execution time. On the other hand, Modified_Bit_AssocRule generates less candidate sets and almost the same frequent sets.

3.5 Discussion

Based on the possible solutions as reported in the subsection 3.2.5, this chapter has reported two modified versions of Apriori algorithm for fast frequent itemset generation. The performance of the algorithm were evaluated by comparing with its other counterparts from [AMS+94] based on two synthetic data sets and one real dataset. The algorithm outperformed Apriori, AprioriTid and AprioriHybrid in terms of both execution time and memory utilization.

FP-growth [HPY00] is one of the efficient algorithms to find frequent itemsets from databases without candidate generations. However, it suffers from two major problems: the algorithm takes much time to construct the FP-tree, specially
for higher dimensions and the performance of the algorithm degrades with the increase of minimum support [HPY00]. These problems can be solved to a great extent by using vertical partitioning approach, which is discussed in the next chapter.

Partitioning is another useful approach used in frequent itemset generation problem. Next chapter reports an existing partitioning approach and analyzed its performance in the light of several real and synthetic datasets. It also reports an enhanced partitioning algorithm for faster frequent itemset generation.