Chapter 3

Association Mining Technique
without Candidate Generation

This chapter presents an efficient One Pass Association Mining technique called OPAM, which finds all frequent itemsets without generating any candidate set. OPAM is an integration of two techniques: a correlogram matrix based technique to generate all frequent 1- and 2-itemsets in a single scan over the database and a technique that uses a vertical layout concept to generate the rest of the frequent itemsets. We experiment with several synthetic and real datasets and compare the performance of OPAM with competitors viz., Apriori and FP-growth and obtained satisfactory results.

3.1 Introduction

Association rules are of the form “80% of the customers who buy bread also buy butter”. Association rules have numerous applications in real world, such as decision support, understanding customer behaviour, tele-communication alarm diagnosis and prediction. A formal definition of the association rule-mining problem is given by Agrawal\textsuperscript{21} is as follows.

Definition 3.1.1 (Association Rule) : An association rule is an implication in the form of $X \Rightarrow Y$, where $X, Y \subseteq I$ are sets of items called itemsets, and
$X \cap Y = \emptyset$. $X$ is called the antecedent while $Y$ is called the consequent. The rule simply means that $X$ implies $Y$.

Two basic measures called support and confidence and two corresponding thresholds, minimum support and minimum confidence, are used to measure the goodness of an association rule.

**Definition 3.1.2 (Support)**: The support of an association rule is defined as the percentage or fraction of records that contain $X \cup Y$ to the total number of records in the database. The count for each item is increased by one every time the item is encountered in a different transaction $T$ in database $D$ during the scanning process. Support is calculated as follows:

$$
Support(X, Y) = \frac{|X \cup Y|}{|D|}.
$$

Before the mining process, users can specify the minimum support as a threshold, meaning that they are interested only in association rules that are generated from itemsets whose support exceeds that threshold.

**Definition 3.1.3 (Confidence)**: Confidence of an association rule is defined as the percentage or fraction of the number of transactions that contain $X \cup Y$ to the total number of records that contain $X$. Confidence is calculated by the following equation:

$$
Confidence(X, Y) = \frac{Support(X, Y)}{Support(X)}.
$$

If the above percentage exceeds the threshold of minimum confidence, an interesting association rule $X \Rightarrow Y$ is generated. Confidence is a measure of strength of the association rule. The goal of association rule mining is to discover association rules that satisfy the predefined minimum support and confidence for a given database. A rule that satisfies both a minimum support threshold and a minimum confidence threshold is called a *Strong Rule*. The association rule-mining problem is usually decomposed into two sub-problems. One is to find itemsets whose occurrence frequency exceeds a predefined threshold in the database; such itemsets are called frequent or large itemsets. The second sub-problem is to generate
association rules from large itemsets with the constraint of minimal confidence. Since the second sub-problem is quite straightforward, most research focuses on the first sub-problem. The first sub-problem can be further divided into two sub-problems: generating candidate sets and generating frequent itemsets. Diagrammatic representation of the association mining technique is shown in Figure 3.1. The databases of interest are large and users are concerns only about items that are frequently purchased together (i.e., appear together in a database transaction). Usually thresholds of support and confidence are predefined by users to drop those rules that are not interesting or useful.

![Diagram](image)

**Figure 3.1**: Various steps in association mining technique

### 3.2 Related Work

Association mining came into existence as market basket analysis on boolean datasets. In association mining, the size of databases are semi-large so that they can usually be accommodated in main memory. They are static in nature and sometimes referred as sequential association mining. Several efficient and improved sequential association mining techniques have been proposed throughout the last two decades\textsuperscript{26}. Next, we discuss in brief some contributions in the area of the sequential association mining.

#### 3.2.1 AIS

The AIS (Agrawal, Imielinski, Swami’93)\textsuperscript{21} algorithm is the first algorithm proposed for mining association rules. During the first pass over the database, the
support count of each individual item is accumulated. Those items whose support counts are less than the support threshold are eliminated from the list of frequent items. From these frequent items, candidate 2-itemsets are generated by extending frequent items that occur with other items in the same transaction. To avoid generating the same itemsets repeatedly the items are ordered. Candidate itemsets are generated by joining a large item in the previous pass with another item in the transaction, which appears later than the last item in the frequent itemsets. To make this algorithm more efficient, an estimation method is introduced to prune itemset candidates that have no hope of becoming large. Consequently the unnecessary effort of counting such itemsets can be avoided. Since all candidate itemsets and frequent itemsets are assumed to be stored in main memory, memory management is necessary for AIS when memory is not enough. The main drawback of the AIS algorithm is that it generates too many candidate itemsets that finally turn out to be small, requiring wasted effort. In addition, this algorithm requires too many passes over the whole database. In this algorithm only one item consequent association rules are generated, which means that the consequents of the rules contain only one item. For example we only generate rules like $X \cap Y \Rightarrow Z$ but not those rules as $X \Rightarrow Y \cap Z$.

### 3.2.2 Apriori

Among the popular algorithms to find large itemsets, the Apriori algorithm\textsuperscript{27} stands at the top because of its simplicity and effectiveness. The basic property that characterizes a large itemset is that all subsets of a large itemset are large. The Apriori algorithm exploits this fact. 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. The support of each candidate itemset is found during a 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 no additional large itemsets are found. The algorithm uses a function, called $apriorigen(L_{k-1})$, which takes the set of all large $k - 1$
support count of each individual item is accumulated. Those items whose support counts are less than the support threshold are eliminated from the list of frequent items. From these frequent items, candidate 2-itemsets are generated by extending frequent items that occur with other items in the same transaction. To avoid generating the same itemsets repeatedly the items are ordered. Candidate itemsets are generated by joining a large item in the previous pass with another item in the transaction, which appears later than the last item in the frequent itemsets. To make this algorithm more efficient, an estimation method is introduced to prune itemset candidates that have no hope of becoming large. Consequently the unnecessary effort of counting such itemsets can be avoided. Since all candidate itemsets and frequent itemsets are assumed to be stored in main memory, memory management is necessary for AIS when memory is not enough. The main drawback of the AIS algorithm is that it generates too many candidate itemsets that finally turn out to be small, requiring wasted effort. In addition, this algorithm requires too many passes over the whole database. In this algorithm only one item consequent association rules are generated, which means that the consequents of the rules contain only one item. For example we only generate rules like $X \cap Y \Rightarrow Z$ but not those rules as $X \Rightarrow Y \cap Z$.

### 3.2.2 Apriori

Among the popular algorithms to find large itemsets, the Apriori algorithm\textsuperscript{27} stands at the top because of its simplicity and effectiveness. The basic property that characterizes a large itemset is that all subsets of a large itemset are large. The Apriori algorithm exploits this fact. 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. The support of each candidate itemset is found during a 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 no additional large itemsets are found. The algorithm uses a function, called $apriorigen(L_{k-1})$, which takes the set of all large $k-1$
itemsets ($L_{k-1}$) as input and produces the candidates for large $k$-itemsets ($L_k$).

Despite its simplicity, the Apriori algorithm suffers from shortcomings. It is not scalable with the size of the database because it scans the database in each iteration to generate large itemsets. It produces a large number of candidate itemsets, out of which only a few are actually frequent itemsets. As a result, the ratio between the number of candidate large itemsets and the number of actual frequent itemsets becomes very high. The same technique is independently proposed by Mannila et al.\textsuperscript{28}. Both works integrated later in\textsuperscript{29}.

3.2.3 SETM

The SETM algorithm\textsuperscript{30} was motivated by the desire to use SQL to calculate large itemsets. In this algorithm each member of the set of large itemsets, $L_k$, is in the form $<TID, Itemset>$ where TID is the unique identifier of a transaction. Similarly, each member of the set of candidate itemsets, $C_k$, is in the form $<TID, Itemset>$. Similar to the AIS algorithm, the SETM algorithm makes multiple passes over the database. In the first pass, it counts the support of individual items and determines which of these are large or frequent in the database. Then, it generates the candidate itemsets by extending large itemsets from the previous pass. In addition, SETM remembers the TIDs of the generating transactions with the candidate itemsets. The relational merge-join operation can be used to generate candidate itemsets. The SETM algorithm saves a copy of the candidate itemsets together with TID of the generating transaction in a sequential manner. Afterwards, the candidate itemsets by sorted on itemsets, and small itemsets are deleted using an aggregation function. If the database is in sorted on the basis of TID, large itemsets contained in a transaction in the next pass are obtained by sorting $L_k$ on TID. This way, several passes are made on the database. When no more large itemsets are found, the algorithm terminates. The main disadvantage of this algorithm is the number of candidate sets $C_k$. Since for each candidate itemset there is a associated TID, it requires more space to store a large number of TIDs. Furthermore, when the support of a candidate itemset is counted at the end of the
pass, $C_k$ is not in ordered. Therefore, again sorting is needed on itemsets.

### 3.2.4 SEAR

SEAR (Sequential Efficient Association Rules) algorithm\textsuperscript{31} is identical to Apriori, except that SEAR stores candidates in a prefix tree instead of a hash tree. In a prefix tree (also called a trie), each edge is labeled by items. Common prefixes are represented by tree branches, and unique suffixes are stored at the leaves. SEAR uses a pass-bundling optimization, where it generates candidates for multiple passes if the candidates fit in memory.

### 3.2.5 DHP

Shortly after the Apriori algorithm was published, Park et al. proposed another optimization algorithm, called DHP (Direct Hashing and Pruning) to reduce the number of candidate itemsets\textsuperscript{32}. During the $k^{th}$ iteration, when supports of all candidate $k$-itemsets are counted by scanning the database, DHP looks ahead and gathers information about candidate itemsets of size $k + 1$ in such a way that all $(k + 1)$-subsets of each transaction are stored in a hash table. Each bucket in the hash table consists of a counter to represent how many itemsets have been hashed to that bucket so far. When a candidate itemset of size $k + 1$ is generated, the hash function is applied on that itemset. If the counter of the corresponding bucket in the hash table is below the minimal support threshold, the generated itemset is not added to the set of candidate itemsets. During the support counting phase of iteration $k$, every transaction is trimmed in the following way. If a transaction contains a frequent itemset of size $k + 1$, any item contained in that $k + 1$ itemset will appear in at least $k$ of the candidate $k$-itemsets in $C_k$. As a result, an item in transaction $T$ can be trimmed if it does not appear in at least $k$ of the candidate $k$-itemsets in $C_k$. These techniques result in a significant decrease in the number of candidate itemsets that need to be counted, especially in the second iteration. Nevertheless, creating the hash tables and writing the adapted database to disk,
at every iteration, causes significant overhead.

3.2.6 Partitioning approach

The partition approach\textsuperscript{33} divides the database into small partitions such that each partition can be handled in the main memory. Let the partitions of the database be $D_1, D_2, \cdots, D_p$. In the first scan, it finds local large itemsets in each partition $D_i$ ($1 \leq i \leq p$). A local large itemset, $L_i$, can be found by using an algorithm such as Apriori. Since each partition can fit in the main memory, there is no additional disk I/O for a partition after the partition is loaded into the main memory. In the second scan, it uses the property that a large itemset in the whole database must be locally large in at least one partition of the database. The union of the local large itemsets found in each partition is used as candidates and are counted through the whole database to find all the large itemsets.

3.2.7 Sampling

Sampling\textsuperscript{34} reduces the number of database scans to one in the best case and two in the worst. A sample which can fit in main memory is first drawn from the database. The set of large itemsets in the sample is then found from this sample using Apriori. Let the set of large itemsets in the sample be $PL$, which is used as a set of probable large itemsets and used to generate candidates which are to be verified against the whole database. The candidates are generated by applying the negative border function, $BD$, to $PL$. Thus the candidates are $BD(PL) \cup PL$. The negative border of a set of itemsets $PL$ is the minimal set of itemsets which are not in $PL$, but all their subsets are. After the candidates are generated, the whole database is scanned once to determine the counts of the candidates. If all large itemsets are in $PL$, i.e., no itemsets in $BD(PL)$ turn out to be large, all large itemsets are found and the algorithm terminates. Otherwise, there are misses in $BD(PL)$; some new candidate itemsets must be counted to ensure that all large itemsets are found, and thus one more scan is needed. In this case, $L \cap PL \neq \emptyset$, 

30
and the candidate itemsets in the first scan may not contain all candidate itemsets of Apriori.

3.2.8 DIC

DIC (Dynamic Itemset Counting)\textsuperscript{35} initially identifies certain 'stops' in the database. It is assumed that we read the records sequentially as we do in other algorithms, but pause to carry out certain computations at the 'stop' points. It defines four different structures: Dashed Box, Dashed Circle, Solid Box, Solid Circle. Each of these structures maintains a list of itemsets. Itemsets in the 'dashed' category of structures have a counter and the stop number with them. The counter is to track of the support value of the corresponding itemsets. The stop number is to keep track whether an itemset has completed one full pass over a database. The itemsets in the 'solid' category structures are not subjected to any counting. The itemset in the solid box is the confirmed set of frequent sets. The itemsets in the solid circle are the confirmed set of infrequent sets. The algorithm counts the support count of the itemsets in the dashed structure as it moves along from one stop point to another. During the execution of the algorithm, at any stop point, the following events take place,

- Certain itemsets in the dashed circle move into the dashed box. These are the itemsets whose support-counts reach minimum threshold value during this iteration.

- Certain itemsets enter afresh into the system and get into the dashed circle. These are essentially the supersets of the itemsets that move from the dashed circle to the dashed box.

- The itemsets that have completed one full pass, move from the dashed structure to the solid structure. That is, if the itemset is in a dashed circle while completing a full pass, it moves to the solid circle. If it is in the dashed box, it moves into the solid box after completing a full pass.
Though this method drastically reduces the number of scans of the database, its performance is heavily dependent on the distribution of the data.

3.2.9 FP-growth

FP-growth\textsuperscript{36} finds frequent itemsets without candidate generation. The algorithm is based on a special data structure called FP-tree, which is a prefix tree of the transactions of the database such that each path represents a set of transactions that share the same prefix. The algorithm works as follows. The algorithm first scans the database once to find the one-element frequent itemsets in the database. Infrequent items are removed from the database and items in the transactions are rearranged in the descending order of the frequencies of items. Then, all transactions containing the least frequent item are selected and the item is removed from the transactions, resulting in a reduced (projected) database. This projected database is processed to find frequent itemsets. Obviously, the removed item is prefix of all frequent itemsets. The item is removed from the database and the process is repeated with the next least frequent item. It is to be noted that FP-tree contains all the information about the transactions and the frequent itemsets. So, to find any information about the transactions and frequent itemsets, one needs to just search the tree. FP-Growth is one of the fastest frequent itemsets finding algorithms. It is robust enough to find the complete set of frequent itemsets. Although the algorithm has many advantages, it suffers from two significant disadvantages.

1. The time taken to construct the FP-tree is quite large, particularly when the dimensionality is large.

2. With the decrease in minimum support threshold value, its performance degrades and at certain instance of time it becomes almost similar to Apriori.

A summary of the algorithms discussed above is given in Table 3.1.

32
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Database Layout</th>
<th>Data Structure</th>
<th>No. of DB Scan</th>
<th>Candidate generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIS</td>
<td>Horizontal</td>
<td>None</td>
<td>K</td>
<td>Yes</td>
</tr>
<tr>
<td>Apriori</td>
<td>Horizontal</td>
<td>Hash Tree</td>
<td>K</td>
<td>Yes</td>
</tr>
<tr>
<td>SETM</td>
<td>Horizontal</td>
<td>None</td>
<td>K</td>
<td>Yes</td>
</tr>
<tr>
<td>SEAR</td>
<td>Horizontal</td>
<td>Prefix Tree</td>
<td>K</td>
<td>Yes</td>
</tr>
<tr>
<td>DHP</td>
<td>Horizontal</td>
<td>Hash Tree</td>
<td>K</td>
<td>Yes</td>
</tr>
<tr>
<td>Partitioning</td>
<td>Vertical</td>
<td>None</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>Sampling</td>
<td>Horizontal</td>
<td>None</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>DIC</td>
<td>Horizontal</td>
<td>Tries</td>
<td>( \leq K )</td>
<td>Yes</td>
</tr>
<tr>
<td>FP Growth</td>
<td>Horizontal</td>
<td>FP-Tree</td>
<td>2</td>
<td>No</td>
</tr>
</tbody>
</table>

\( (K: \text{size of the longest frequent itemset}, \ DB: \text{database}) \)

### 3.3 Motivation

All these algorithms are based on candidate generation and suffer from the following two major problems.

1. They need to scan the database multiple times, which is costly, particularly when the database is very large.

2. They generate huge candidate sets in comparison to the actual frequent itemsets.

So, it is desirable to develop an algorithm, which not only obviates scanning the database repeatedly but also does not generate candidate sets. We present an one-pass association mining technique that addresses this problem by introducing an integrated approach to find the frequent itemsets.

### 3.4 OPAM: One Pass Association Mining Technique

OPAM adopts an integrated approach to solve the frequent itemset finding problem in a single pass over the database. Initially, it attempts to generate all frequent 1- and 2-itemsets directly using a \textit{correlogram matrix} based technique. In the next
phase, to find the remaining higher order frequent itemsets, it exploits a vertical layout concept for the database. Next, we provide the background of each of these techniques.

3.4.1 Correlogram matrix based technique

Correlogram matrix is a co-occurrence frequency matrix. It is a matrix of size, $N \times (N + 1)/2$, for a transaction database with $N$ items. Each cell of the matrix contains the frequency of co-occurrence of an item pair. Item pairs are specified by the row index and the column index of the cell.

For example, to specify the frequency of co-occurrence of item pair \{4,5\}, corresponding to the sample market basket dataset depicted in Table 3.2, the content of the cell \(4,5\) in the correlogram matrix (see Figure 3.2) with an index of row 4 and column 5 will indicate the co-occurrence frequency of the item pairs \{4,5\}. On the other hand, a cell for which, the row and column indices are the same, specifies the occurrence frequency of a single item. Thus, seen in Figure 3.2, the cell \(3,3\) indicates the occurrence frequency of the single itemset \{3\}.

3.4.2 Construction of correlogram matrix

The correlogram matrix is constructed by a single scan of the database. In order to construct the correlogram matrix, we model the situation graph theoretically. All
items participating in a particular transaction are considered nodes. As items appear in the transaction in a lexicographical order, we say that they form a directed graph involving all items as node of the graph. Each item is linked by a single link or edge. Thus, only a directional path exists between any two nodes. To illustrate, let us consider sample market basket dataset given in Table 3.2. Items I1, I2, I4 and I5 participate in transaction T4. Thus, they form a directed graph as shown in the Figure 3.3.

To count the co-occurrence frequency of all items participating in a particular transaction, we count links among all pairs of nodes and correspondingly increment the content of a cell with the corresponding indices. Thus, if we consider the example in Figure 3.3, we increment the contents of cells (1,2), (1,4), (1,5), (2,4) and (2,5). We also increment the count of first node of a pair. For example, when incrementing the content for the pair (1,2), we also increment the content of the cell (1,1) for storing the frequency of item I1. The scenario after incrementing the content of correlogram matrix becomes the one shown in Figure 3.4. Thus by following the procedure discussed above, one can construct the correlogram matrix by scanning the database only once. From the correlogram matrix, we can extract the frequent 1- and 2-itemsets with a given minimum support threshold in a straightforward manner.

The advantages of this technique are as follows.
Table 3.3: Vertical layout of sample market basket data

<table>
<thead>
<tr>
<th>Item Purchased</th>
<th>Transaction Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>T1,T4,T5,T7,T8</td>
</tr>
<tr>
<td>I2</td>
<td>T1,T2,T3,T4,T6,T8</td>
</tr>
<tr>
<td>I3</td>
<td>T3,T5,T6,T7,T8</td>
</tr>
<tr>
<td>I4</td>
<td>T1,T2,T4,T8</td>
</tr>
<tr>
<td>I5</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>I6</td>
<td>T1,T3,T5</td>
</tr>
</tbody>
</table>

1. Candidate generation step is no more required to find 1- and 2- frequent itemsets.

2. Unlike other algorithms, it requires only one scan over the database for finding all the frequent 1- and 2-elements itemset.

3. Since it is memory based, it is fast.

3.4.3 Mining frequent itemsets using vertical transaction layout

Transaction layout is a method that can be used to format items in a transaction database. Currently, there are three approaches: horizontal\textsuperscript{21}, vertical\textsuperscript{37} and the hybrid\textsuperscript{38}. Horizontal layout combines items in a transaction row-wise. This layout suffers from the problem of superfluous processing since there is no index on the items. In a vertical layout, each item is associated with a column of values representing the transaction in which it is present. A vertical layout creates an index on the items and reduces the effect of large data sizes since there is no need to rescan the whole database each time. The technique we propose adopts the vertical layout approach for mining the remaining higher order frequent itemsets.

To take the advantages of the vertical format, we transform our database into vertical form for mining frequent itemsets. The corresponding vertical layout of the sample market basket data given in Table 3.2 is presented in Table 3.3.

Once we create the vertical layout of the original transaction database, the next
Table 3.4: 2-element frequent item sets

<table>
<thead>
<tr>
<th>Item Set</th>
<th>Transaction List</th>
</tr>
</thead>
<tbody>
<tr>
<td>{I1,I2}</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>{I1,I3}</td>
<td>T3,T5,T8</td>
</tr>
<tr>
<td>{I1,I4}</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>{I1,I5}</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>{I2,I3}</td>
<td>T3,T6,T8</td>
</tr>
<tr>
<td>{I2,I4}</td>
<td>T1,T2,T4,T8</td>
</tr>
<tr>
<td>{I2,I5}</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>{I4,I5}</td>
<td>T1,T4,T8</td>
</tr>
</tbody>
</table>

Table 3.5: 3-element frequent item sets

<table>
<thead>
<tr>
<th>Item Set</th>
<th>Transaction List</th>
</tr>
</thead>
<tbody>
<tr>
<td>{I1,I2,I4}</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>{I1,I2,I5}</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>{I1,I4,I5}</td>
<td>T1,T4,T8</td>
</tr>
<tr>
<td>{I2,I4,I5}</td>
<td>T1,T4,T8</td>
</tr>
</tbody>
</table>

Table 3.6: Largest frequent item sets

<table>
<thead>
<tr>
<th>Item Set</th>
<th>Transaction List</th>
</tr>
</thead>
<tbody>
<tr>
<td>{I1,I2,I4,I5}</td>
<td>T1,T4,T8</td>
</tr>
</tbody>
</table>

phase is straightforward. We intersect two item records from the vertical table. If the resultant record contains number of transaction IDs greater than or equal to a given minimum support threshold, the item pairs in the intersection form the frequent itemset. Support counting is performed simply by counting the number of transaction IDs that are common in both item records in the intersection. We term such an intersection as a successful intersection.

To avoid unnecessary computation of intersection we use the same union and prune step as used in Apriori algorithm. We intersect two records if both the target itemsets pass through the union and pruning steps successfully. As the possible number of 2-element itemsets are huge, to avoid working with them directly, we use correlogram matrix as introduced in previous section to find all frequent 1- and 2-element itemsets directly. After generating all the 1- and 2-element frequent sets using the correlogram matrix, we simply update the vertical table by eliminating all the records corresponding to non-frequent itemset of size one. Next, we perform intersection among the pair of records from 1-element frequent set, which are actually in frequent 2-element itemset. The intersection of transaction records then
continue until no successful intersection is possible. For illustration, intermediate results during iterations to obtain all the frequent itemsets from market basket in Table 3.2 with minimum support threshold 3 are shown in tables 3.4, 3.5, 3.6.

3.4.4 Proposed algorithm and its implementation issues

This section presents the algorithm for the proposed integrated approach (see Algorithm 1). It also discusses some of the issues related to efficient implementation of the algorithm. The algorithm accepts the market-basket database $D$ and minimum support $\sigma$, as input and it generates all the frequent itemsets as output. Steps 1 to 4 of the algorithm are dedicated to the first phase of the approach, i.e., finding of 1- and 2-element frequent itemsets using correlogram matrix of the original database. After step 4, we get an alternative representation of the database as discussed above in the second phase of the approach. Generally, compared to the number of transactions the numbers of items or dimensions are relatively much lower. Thus such a vertical database can be easily stored in main memory. However, if the number of transactions are very large, it becomes very difficult to store such transaction list in main memory. To handle such cases, we use a compact representation of the transaction list.

At line 9, the union operation returns the new itemset if union is possible, otherwise it returns null. Following downward closure property, pruning operation returns false if all the subsets of the new itemset generated by union operation, are frequent. In step 12, the intersection between two item record sets are carried out.
| input : D (Original Dataset), σ (Minimum Support) |
| output: L (List of frequent itemsets) |

1. Generate Correlogram Matrix $M$ from $D$;
2. Construct vertical database $V$ from $D$;
3. Traverse the $M$ to generate one and two element frequent itemsets;
4. Write all the one and two frequent itemsets to $L$;
5. Update $V$ with frequent two element itemsets;
6. while successful intersection possible do
   7. for $i ← 1$ to $|V|$ do
      8. for $j ← i + 1$ to $|V|$ do
         9. $\text{NewItemSet} = \text{Union}(V[i].\text{ItemSet}, V[j].\text{ItemSet})$;
         10. if NewItemSet $\neq \text{Null}$ then
              11. if Pruning (NewItemSet) $=$ False then
                  12. $\text{NewTransList} = \text{Intersection}(V[i].\text{TransList}, V[j].\text{TransList})$;
                  13. if Count (NewTransList) $\geq \sigma$ then
                      14. Write the NewItemSet and NewTransList into $L$;
                      15. Update $V$;
              16. end
         17. end
    18. end
   19. end
20. end

Algorithm 1: OPAM: The Algorithm

To perform intersection, we apply simple bit wise and operation. It is very fast compare to normal intersection, which is performed by comparing elements from both participating records. In line 13, Count returns support count of the new itemset. All the itemsets satisfy minimum support criteria are stored in the list $L$. The vertical database $V$ is updated by eliminating all the no-frequent itemsets. The process of intersection continues with new records until no successful intersection is possible.

For compact representation of transaction list, we adopt the concept used in creating binary coded decimal (BCD) representation for integers. OPAM initially stores the transaction list associated with a itemset record in a bit vector. It is then converted into BCD of $M$ bit size. Thus, if the maximum transaction ID is $T$, it requires $T/M$ sized array of a data type that can accommodate $M$ bit data.
For example, let us consider a transaction that has maximum transaction ID i.e. \( T = 9 \) and a BCD scheme of 3 bits (i.e., \( M \)). For a transaction list \{2, 3, 4, 6, 7, 8\}, the compact BCD representation is shown in Figure 3.5. After converting the item records into BCD form, one can easily count the support as shown in Figure 3.6.

Intersection between two BCD array is performed through bitwise and operation. It is very fast and effective. The interesting fact is that as the iteration moves to a higher level, the number of the transaction IDs per records goes down. The number of records also gradually decreases. This is because of the fact that the higher order frequent itemsets (itemsets of size 3 or more) are normally fewer compared to possible lower order itemsets. Thus, the performance gradually increases and consumption of memory space decreases.

### 3.5 Analysis of Our Algorithm

Here, we present proof of correctness and completeness of OPAM and then we analyse our algorithm in terms of computational complexity.

#### 3.5.1 Completeness and correctness

**Lemma 3.5.1.** Correlogram matrix based technique generates all the 2-element itemsets which are frequent w.r.t. minimum support (\( \text{minsup} \)), a user defined threshold.

**Proof.** Correlogram matrix based technique computes support counts of all the 2-element itemsets by using exhaustive search in the transaction database. Next, it extracts only those 2-element itemsets which satisfy the \( \text{minsup} \) condition. Hence, the proof. \( \square \)

**Lemma 3.5.2.** OPAM is complete, i.e., OPAM extracts all the frequent itemsets w.r.t. \( \text{minsup} \).

**Proof.** It can be proved in two steps. First, the 2-element frequent itemsets generated is complete, which is evident from Lemma 3.5.1. Second, OPAM generates
all those itemsets of size $> 2$ based on the output of first step and their support counts, satisfy the $\text{mins}_{\text{up}}$ condition. Similarly, it is true for any itemsets of size $\geq k$. Thus OPAM generates all the frequent itemsets which satisfy the $\text{mins}_{\text{up}}$ condition and hence the proof.

\[ \square \]

**Lemma 3.5.3.** **OPAM is correct**, i.e., frequent itemset generated by OPAM satisfy $\text{min}_{\text{sup}}$ criterion.

**Proof.** This lemma can be proved by contradiction. Let us assume that an itemset $I_{k+1}$ is frequent, generated by intersecting itemset $X_k$, with transaction record $T_x$, where cardinality of $T_x$ is above or equal to $\text{mins}_{\text{up}}$, i.e., $|T_x| \geq \text{mins}_{\text{up}}$ and itemset $Y_k$, with $|T_y| < \text{mins}_{\text{up}}$. Since $|T_y| < \text{mins}_{\text{up}}$, resulting intersection between $X_k$ and $Y_k$ never satisfy minimum support criterion, i.e., $|T_x \cap T_y| \notin \text{mins}_{\text{up}}$, which contradicts the assumption, hence the proof.

\[ \square \]

### 3.5.2 Complexity analysis

Below we present analysis of OPAM in terms of space and time complexity.

#### 3.5.2.1 Space complexity

OPAM requires space for correlogram matrix and transaction records of all the itemsets in each iteration. Thus, space complexity for the two data structures can be calculated as follows.

a) **Space for correlogram matrix:** For a transaction database with $N$ items, the fixed space requirement for correlogram matrix is:

$$SPACE_{CM} = O(N \times (N + 1)/2) 
\approx O(N^2/2).$$

b) **Space for frequent itemset:** Assume that $k$ is the number of frequent itemsets
in each iteration. \( T \) is the number of transactions in the database. If we consider, \( M \) as the bit size for BCD scheme, the space required in each level is:

\[
SPACE_{FI} = O(k \cdot (T/M))
\]

The value of \( k \) is normally very high in case of two element frequent itemsets and it decreases with the increase in iteration level.

It is worth mentioning that requirement of both the data structures is not simultaneous. Correlogram matrix is needed at the early stage of the algorithm. Once vertical layout is constructed based on two element frequent itemsets, correlogram matrix can be deleted from the memory.

3.5.2.2 Time complexity

We compute the time complexity based on three different computational costs.

a) Construction of correlogram matrix: Assume that the database contains \( T \) transactions and a maximum of \( N \) items in each transaction. For storing and updating support count of item pairs in the correlogram matrix with respect to each transaction, it requires \( O(T \cdot N^2) \) time. The time requirements for accessing the correlogram matrix is \((N \times (N + 1)/2) \approx N^2\). The cost for construction as well as to find the 2-element frequent itemsets from the correlogram matrix is \( O(T \cdot N^2) + O(N^2) \), which become \( O(T \cdot N^2) \).

b) Construction of vertical layout: To represent each itemset with transaction list using BCD scheme, it requires to process each transaction ID from the list. Thus for \( N \) items and \( T \) transactions, the complexity is \( C_{Ver} = O(T \cdot N) \).

c) Cost of intersection: Assume that there are \( k \) frequent itemsets in each iteration and \( \xi \) is the maximum level of iteration. For each iteration it considers all pair of items for intersection. Since we are using bitwise AND operation for
intersection, it is very fast as compared to other operations and thus ignored. The cost incurred for intersection is \( C_{Int} = O(\xi * k^2) \).

The total cost of OPAM is

\[
Cost_{OPAM} = C_{CM} + C_{Ver} + C_{Int}
= O(T * N^2) + O(T * N) + O(\xi * k^2).
\]

Since, we are reading \( T \) transactions in a single scan over the database, thus the performance of OPAM mostly depends on number of items, \( N \), in the database.

### 3.6 Performance Evaluation

To evaluate the performance of OPAM in comparison to other techniques, we use two popular techniques, Apriori and FP-growth. We implemented OPAM using Java 1.6 on Windows 7 platform running in 2.53 GHz machine. For other two algorithms, Apriori and FP-growth, we used Java based SPMFa tool. SPMF (Sequential Pattern Mining Framework) is an open-source data mining platform written in Java and distributed under the GPL v3 license.

#### 3.6.1 Dataset used

We have generated synthetic datasets according to the specifications given in Table 3.7. The synthetic datasets were created with the data generator in ARMinerb software, which follows the basic spirit of well-known IBM synthetic data generatorc for association rule mining. The size of the data (i.e., number of transactions), the number of items and the number of unique patterns (incase of synthetic dataset) in the transactions are the major parameters in data generation. We also used

---

*ahttp://www.philippe-fournier-viger.com*

*bhttp://www.cs.umb.edu/laur/ARMiner/*

*cwww.almaden.ibm.com/cs/quest/*
two real life Mushroom\textsuperscript{a} and Chess dataset taken from FIMI\textsuperscript{b}. Below we present experimental results that use these synthetic and real datasets.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>No. of Transactions</th>
<th>No. of Items</th>
<th>Avg. size of Transaction</th>
<th>No. of Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10I400D100K</td>
<td>100,000</td>
<td>400</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>T10I600D100K</td>
<td>100,000</td>
<td>600</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>T10I800D100K</td>
<td>100,000</td>
<td>800</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Mushroom</td>
<td>8124</td>
<td>128</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Chess</td>
<td>3196</td>
<td>75</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### 3.6.2 Experimental results

Performance of the three algorithms is compared in terms of execution time for different minimum support values. For different synthetic datasets the performance of the three algorithms degrades gradually along with decrease in the minimum support value. Apriori always needs very high computational time for all the datasets. Compare to other two algorithms, the performance of OPAM is found effective, especially in synthetic datasets. However, in case of real datasets, FP-growth performs well compare to other algorithms, especially when data is dense. From the Figure 3.7 and 3.8, it can be observed that FP-growth always needs certain computational time even when number of frequent items are zero (when minimum support is high). This is required because of the minimum time needed to construct the tree. For the same situation, OPAM also requires minimum time for scanning the database once and constructing the correlogram matrix.

### 3.7 Discussion

In this chapter, we have presented an efficient frequent itemset finding technique. The technique works in two phases, a correlogram matrix technique to generate

\[ \text{http://www.ics.uci.edu/~mlearn/MLRRepository.html} \]
\[ \text{http://fimi.ua.ac.be} \]
Figure 3.7: Performance comparison on synthetic dataset

Figure 3.8: Comparison against execution time vs. minimum support on real data
those 1- and 2-element frequent itemsets, and a vertical layout technique to generate higher order frequent itemsets. The technique is able to generate all frequent itemsets in one scan of the database. Another advantage of OPAM is that it also supports interactive mining of 1- and 2-element itemsets. Experiments have shown that OPAM performs well in comparison to Apriori and FP-growth algorithms.

We have explored the advantage of the correlogram matrix for extracting 1- and 2- element frequent itemsets in finding strongly correlated item pairs from a transaction database. This is discussed in the next chapter.