Chapter 1

Mining Associations

1.1 Introduction

During the 1980s, rapid growth in database technology has resulted in massive amount of data being collected and stored in scientific, business and government databases. Traditional data analysis tools simply cannot handle the sheer volume of data streams that are being continuously generated. Fueled by an increasing need to automatically summarize and analyze the data, researchers began to look into methods to convert the raw data into more compact patterns in the form of logical rules, decision trees, data clusters, etc.

Association rule was originally formulated in the context of transaction data at supermarket [1]. This *market basket* data, as it is commonly known, consists of transactions made by each customer at a retail store. The goal of mining association rules is to see if occurrence of certain items in a transaction can be used to deduce occurrence of other items. For example, one might find rules such as "70% of the customers who buy Coke, also buy chips." If indeed such relationships are found, then they can be put to various profitable uses such as cross-selling, shelf management, inventory management, etc.

An association rule is often expressed as $X \Longrightarrow Y$, where X and Y are sets of items. Simply put, given a set of items, association rules predict the presence of other items in the same transaction to certain degree of probability (called *confidence*). For example, an association rule may indicate that out of all customers who buy bread, 75% of them would end up buying milk too. This rule can be written as:

Example 1 : $\{Bread\} \Longrightarrow \{Milk\}$ with 75% confidence.

In a way, confidence can be regarded as a measure of the strength of a rule. Quite often, knowing the confidence value alone is insufficient. One may need to find the actual coverage of the rule, i.e. what percentage of the total transactions contains both Milk and Bread. This measure is called the *support* of a rule. For instance, consider the next example :

Example 2 : $\{Bread, Peanut\ Butter, Shampoo\} \Longrightarrow \{Milk\}$ with 90% confidence.

Suppose the rule in example 1 has a 10% support (i.e. one-tenth of the transactions involve the purchase of bread and milk) whereas the support for example 2 is less than 0.01%. For business analysts, the second example is not interesting, despite having a higher confidence, because it may not be feasible to promote the items together due to their lack of support. This shows that both support and confidence should be taken into consideration when assessing the significance of a rule.

The problem of deriving association rules from data is far from trivial due to the exponential number of ways in which items can be grouped together to form a rule. In recent years, much research effort has been put into formulating efficient solution to this problem. Section 1.3 provides an overview of the Apriori algorithm, one of the earliest algorithm that successfully address the combinatorial explosion problem.

Also, in many practical setting, the support and confidence measures are not adequate to determine how interesting a rule might be for data analysts. For instance, the rule in Example 1 may seem too obvious - one does not need data mining to know that bread and milk are often bought together! On the other

Table 1.1: Transactions from supermarket.

Transaction ID	Items
1	Bread, Milk
2	Beer, Diaper, Bread, Eggs
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Bread, Diaper, Milk

hand, a rule that says "Customers who buy beer tend to buy diaper" is more interesting because it is novel, unexpected and potentially actionable. The various ways to characterize the interestingness of a rule is the subject of Section 1.5.

The concept of association rules can be generalized in many ways. For example, each supermarket transaction can be augmented with a timestamp associated with the time at which a transaction is made. If this information can be put to use, one can find useful relationships such as if an item A was bought by a customer, then he/she will likely buy an item B in a few days time. This type of pattern is known as *sequential patterns* or *sequential associations*. The formulation of sequential associations is not only important because they represent more general relationships, but they are also important from an algorithmic point of view. By bringing in sequential relationships, the combinatorial complexity of the problem increases enormously. The various techniques for generating sequential associations will be described in Section 1.6.

1.2 Association Rule Discovery: Problem Definition

Let I denotes the set of all items and T be the set of transactions, where each transaction $t \in T$ is a subset of I. In association rule terminology, a set containing k items is also known as a k-itemset. Given an itemset $C \subseteq I$, we define the *support count* of C with respect to T, $\sigma(C)$, to be the number of transactions that contain C, i.e.

$$\sigma(C) = |\{t|t \in T, C \subseteq t\}|.$$

Example 3 : Consider the set of transactions shown in Table 1.1. The set of all items in this database is $I = \{$Bread, Beer, Coke, Diaper, Milk, Eggs$\}$. The support count for the 2-itemset {Diaper, Milk} is $\sigma(\{Diaper, Milk\}) = 3$; while the support count for the 3-itemset {Diaper, Milk, Beer} is $\sigma(\{Diaper, Milk, Beer\}) = 2$.

Definition 1 : An *association rule* is an implication expression of the form $X \stackrel{s,\alpha}{\Longrightarrow} Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \emptyset$. The *support* s of the rule $X \stackrel{s,\alpha}{\Longrightarrow} Y$ is defined as $\sigma(X \cup Y)/|T|$, and the *confidence* α is defined as $\sigma(X \cup Y)/\sigma(X)$, where $|T|$ denotes the total number of transactions.

Example 4 : Consider the rule {Diaper, Milk} $\Longrightarrow$ {Beer}. Using the result of the previous example, the support of this rule is $2/5 = 40\%$ since $\sigma(\{Diaper, Milk, Beer\}) = 2$ and $|T| = 5$. The confidence of this rule is $2/3 = 67\%$ where $\sigma(\{Diaper, Milk\}) = 3$.

Definition 2 :**Problem Definition.** The goal of mining association rules is to discover all rules $X \stackrel{s,\alpha}{\Longrightarrow} Y$, having support greater than or equal to some minimum support threshold, *minsup*, and confidence greater than or equal to a pre-specified minimum confidence threshold, *minconf*.

Support is an important measure because it provides a good indication on the number of samples covered by the rule. Rules that have very small support are often unreliable, since they may not represent a significant proportion of the population. Filtering based on the minimum required support is also critical in reducing the number of possible association rules to a manageable size. This is because for each itemset C, the total number of possible rules is proportional to $2^{|C|}-2$. Since the number of possible itemsets is $2^{|I|}-1$, the total number of possible rules can be exponentially large. Hence, support-based filtering is absolutely needed for large dimensional databases.

Confidence indicates the strength of the implication. Unlike support, it is an asymmetric measure, i.e. confidence($X \Longrightarrow Y$) $\neq$ confidence($Y \Longrightarrow X$) unless $\sigma(X) = \sigma(Y)$. Furthermore, it is non-transitive because the presence of highly confident rules $X \Longrightarrow Y$ and $Y \Longrightarrow Z$ does not necessarily mean that $X \Longrightarrow Z$ will have minimum confidence.

The problem of mining association rules is often decomposed into two subtasks :

1. to find all itemsets that satisfy the *minsup* threshold. These are called *frequent* itemsets.

2. to generate highly confident rules from the frequent itemsets.

Frequent itemset generation is the most time-consuming operation. A naive approach of enumerating all possible itemsets would require $O(2^{|I|})$ operations. The first algorithm that handled this problem of exponential explosion elegantly was the Apriori algorithm [2]. Apriori used a very fundamental property of the support measure to prune the search space. This property basically says that the support of an itemset can never exceed the minimum support of all of its subsets. For example, the support of the 3-itemset { Bread, Butter, Milk } will always be less than or equal to the support of 2-itemsets {Bread, Butter}, {Bread, Milk} and {Milk, Butter}. This is because any transactions that contain {Bread, Butter, Milk} must also contain {Bread, Butter}, {Bread, Milk} and {Butter, Milk}; but not vice-versa.

This *anti-monotonicity*[1] property can be used to systematically prune the search space of the desired itemsets, by progressively increasing the length of the itemsets being discovered. In the next section, we will take a look at Apriori and several other algorithms for generating frequent itemsets [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. Later, we will see how rules that satisfy the minimum confidence threshold can be extracted from these frequent itemsets.

1.3 Frequent Itemset Generation

A number of serial algorithms have been developed for discovering frequent itemsets. Most of them use the anti-monotone property of the itemset support criterion. This property states that as the size of an itemset increases, its support either decreases or stays the same. Consequently, a k-itemset is frequent only if all of its $(k-1)$-subitemsets are also frequent. We will illustrate the importance of this property using Figure 1.1.

We begin by representing the set of all possible itemsets of I using a a lattice structure. Briefly, given a set I, a lattice is a systematic enumeration of all the subsets of I, starting with the empty set, followed by all the singleton itemsets at the first level, all 2-itemsets at second level, and so on. In the lattice structure, a k-itemset appears at the k^{th} level of the lattice, and is linked to all its $(k-1)$-subitemsets appearing at level $k-1$.

Figure 1.1 illustrates the lattice structure of the itemset $I = \{A, B, C, D, E\}$. Let us consider the 2-itemset $\{A, B\}$. If the support of this itemset is below the minimum support requirement, then any of its supersets (the shaded itemsets of size 3, 4 and 5) will be infrequent due to the anti-monotone property of support. Thus, the entire subgraph containing the supersets of $\{A, B\}$ can be pruned immediately, thereby reducing the size of the search space.

The *Apriori* algorithm by Agrawal et al[2] was one of the earliest algorithm which pioneered the use of this anti-monotone property. This property can be used to systematically control the exponential growth in the number of possible itemsets that need to be searched to get the desired frequent itemsets. In the following, we will see how *Apriori* works.

[1] A function f is anti-monotone if $f(x) > f(y)$ whenever $x < y$.

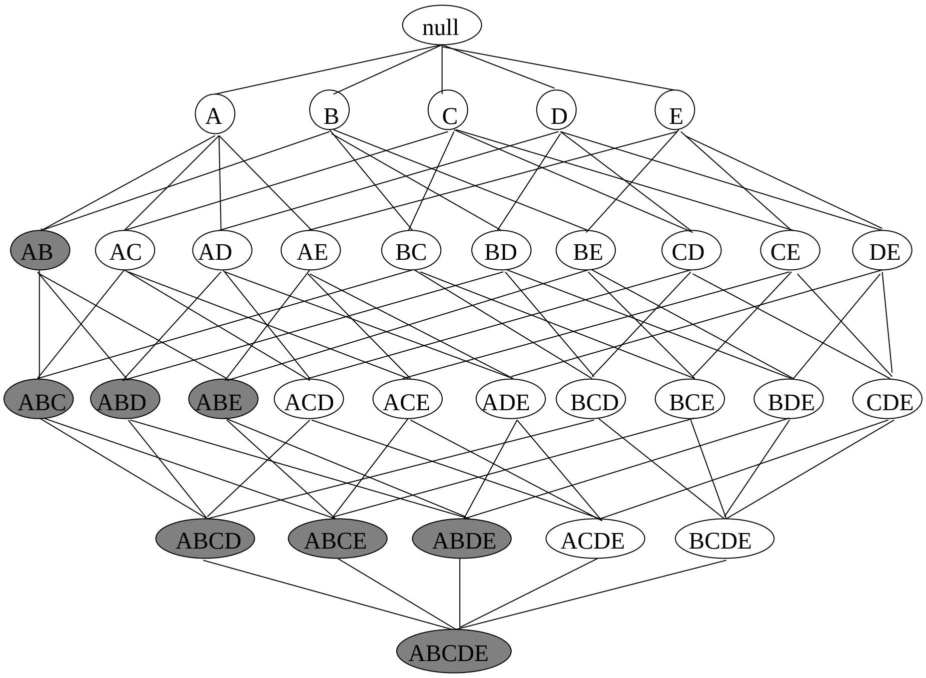

Figure 1.1: itemset Lattice and Frequent itemsets

1.3.1 Apriori Algorithm

A high level illustration of the *Apriori* algorithm for the set of transactions given in Table 1.1 is shown in Figure 1.2 . The algorithm starts by finding all frequent 1-itemset, i.e. individual items having support greater than or equal to *minsup*. With *minsup* at 60%, which corresponds to a minimum count of 3, the 1-itemset {Coke} (count=2) and {Eggs} (count=1) are removed. The remaining frequent 1-itemset are used to generate potential itemsets of size 2. In our example, there are ${}^4C_2 = 6$ candidate 2-itemset generated from the four frequent 1-itemset. However, when the actual support of these *candidate* itemsets are counted, it is found that {Bread, Beer} and {Milk, Beer} are infrequent. Thus, there remains only four frequent 2-itemset. Next, Apriori proceeds to compute all possible candidates of size 3. One possibility is to take a union of all the frequent 2-itemset, and generate all candidate 3-itemset involving items in the union. For example, since the union of all frequent 2-itemset in Figure 1.2 is {Bread, Milk, Diaper, Beer}, one will generate ${}^4C_3 = 4$ candidate 3-itemset with this approach..

However, the *Apriori* algorithm uses a more efficient candidate generation approach. It decides in an apriori fashion, without actually counting the support of the candidate, that {Milk, Diaper, Beer} cannot be frequent by making use of the anti-monotonicity property of support. In this example, it will generate only 1 candidate itemset, which is {Bread, Milk, Diaper}. The algorithm uses a specific form of self-join operation to implement this apriori pruning, as we will see in a while. A glimpse of how apriori prunes the search space can be obtained by looking at the number of subsets that need to be counted if every subset up to size-3 was to be considered for counting, which is ${}^6C_1 +{}^6 C_2 +{}^6 C_3 = 41$. As shown in the figure, the use of the apriori principle reduces this number down to 13 itemsets.

The pseudo code of the algorithm is given in Figure 1.3. The algorithm makes multiple passes over the database. Initially F_1 contains all the frequent 1-itemset. During pass k, the algorithm finds the frequent k-itemset, F_k, that passes the minimum support threshold. The algorithm terminates after l iterations when F_l is empty. In each pass, the algorithm first generates C_k, the size-k candidate itemsets using the frequent itemsets, F_{k-1}, generated from the previous pass. Once the candidate itemsets are found, their frequencies are computed by counting how many transactions contain these candidate itemsets. Finally, F_k is generated by eliminating the itemsets in C_k with frequencies below the minimum support. The union of all the frequent

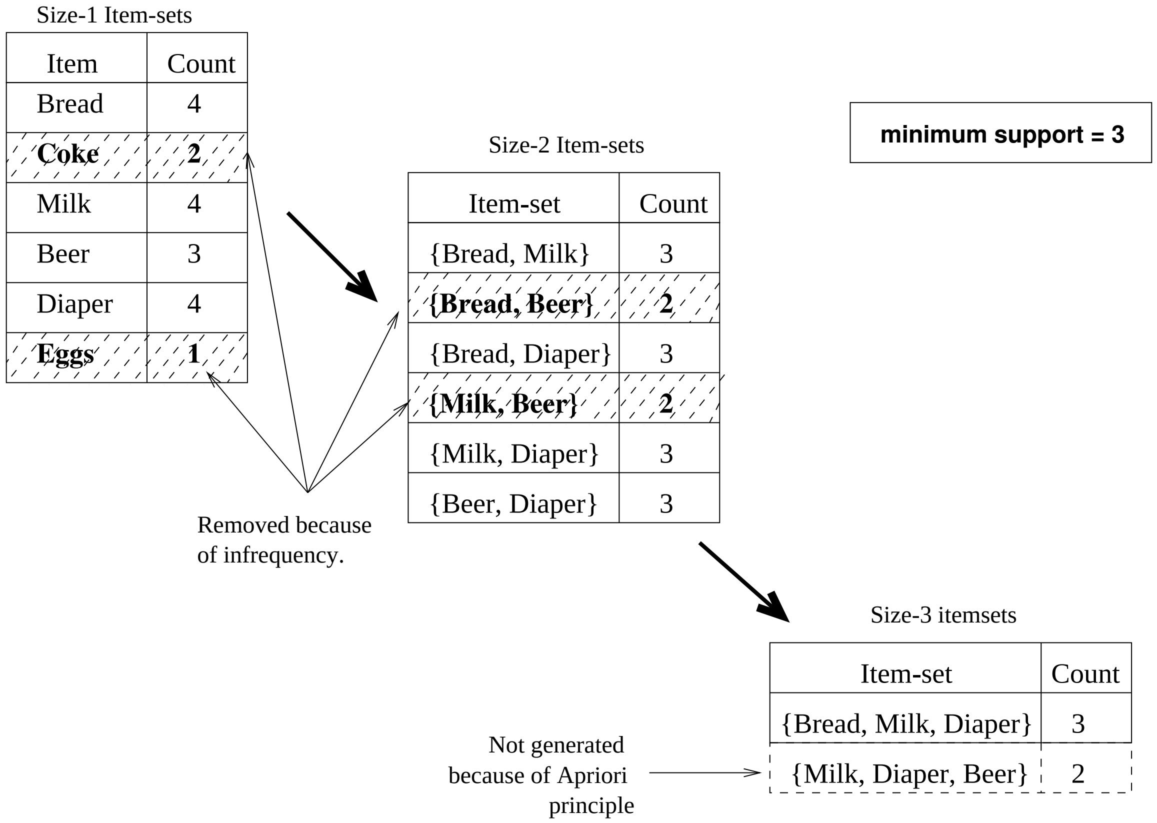

Figure 1.2: Illustration of Apriori algorithm

itemsets, $\bigcup F_k$, will be used to generate association rules.

In the following, we will discuss the two crucial operations in the Apriori algorithm: candidate generation (*apriori_gen*) and candidate counting (*subset*).

1. **Candidate Generation:**

   First, let us assume that items in each transaction are sorted according to their lexicographic ordering. For simplicity, we will represent each item by a numeric symbol, $I = \{1, 2, \cdots d\}$. An itemset of size k can be written as $L_k = \{l_1, l_2, \cdots, l_k\}$, where all the l's are in sorted order.

   The candidate generation function, apriori_gen, is composed of two steps :

   (a) **join step**

   In this step, a candidate itemset C_k will be generated by extending every frequent itemset of size $k-1$ with an item from another frequent itemset of size $k-1$. This step is implemented using a specific-form of join operation. In relational database terminology, this operation can be written in the following way :

   insert into C_k
   select $p.item_1, p.item_2, \cdots, p.item_{k-1}, q.item_{k-1}$
   from $F_{k-1}\ p$, $F_{k-1}\ q$
   where $p.item_1 = q.item_1, \cdots, p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}$;

   The above join operation will combine two frequent itemsets of size $k-1$ only if its $k-2$ prefixes are the same. This is an efficient way to generate large candidate set as shown in the next example :

```
F_1 = { frequent 1-itemsets} ;
for ( k = 2; F_{k-1} ≠ φ; k++ ) {
    C_k = apriori_gen(F_{k-1})
    for all transactions t ∈ T {
        C_t = subset(C_k, t)
        for all candidates c ∈ C_t
            c.count++
    }
    F_k = {c ∈ C_k | c.count ≥ minsup}
}
Answer = ⋃ F_k
```

Figure 1.3: Apriori Algorithm

Example 5 : If $L1 = \{1,2,3\}$ and $L2 = \{1,2,5\}$ are frequent 3-itemsets, i.e. $L1 \in F_3$ and $L2 \in F_3$, then the join operation on $L1$ and $L2$ produces the size-4 candidate $\{1,2,3,5\}$. However, it will not join the two itemsets, $L1 = \{1,2,3\}$ and $L3 = \{2,3,4\}$ due to their different prefixes. This is because if $\{1,2,3,4\}$ is frequent, then it can always be generated by joining $L1$ with $\{1,2,4\}$ (which has to be frequent according to the anti-monotone property). Similarly, in the example shown in Figure 1.2, assuming lexicographic ordering, the 3-itemset {Bread, Milk, Diaper} is generated from the join operation between {Bread, Milk } and {Bread, Diaper}. The 3-itemset {Milk, Diaper, Beer} is not generated at all because the 2-itemset {Milk, Beer} does not belong to F_2.

(b) **prune step**

The prune step will remove all itemsets $c \in C_k$ if any size-$(k-1)$ subset of c are not in F_k :

forall itemsets $c \in C_k$ do
  forall $(k-1)$-subsets s of c do
    if $(s \notin F_k)$ then
      delete c from C_k

Example 6 : Let $F_3 = \{\{1,2,3\},\{1,2,4\},\{1,2,5\},\{1,3,5\},\{1,4,5\},\{2,4,5\}\}$. The join step will produce the candidate itemsets, $C_4 = \{\{1,2,3,4\},\{1,2,3,5\},\{1,2,4,5\}$. However, after the pruning step, $\{1,2,4,5\}$ is the only remaining candidate itemset in C_4. The candidate $\{1,2,3,4\}$ is pruned since $\{2,3,4\} \notin F_3$ while $\{1,2,3,5\}$ is removed because $\{2,3,5\} \notin F_3$.

2. **Candidate Counting:**

Once the candidate itemsets have been generated, the next step is to efficiently compute the support count of each candidate. One naive way to do this is by string-matching each transaction against each candidate itemset. A faster approach is to use a hash tree to store the candidate itemsets [2]. Figure 1.4 illustrates an example of a hash tree with candidates of size 3. The internal nodes of this tree contain hash tables with pointers to child nodes. The leaf nodes of this tree contain all the candidate 3-itemset. The root node is defined to be at depth 1.

In this section, we will describe (1) how the hash tree is constructed, and (2) how the tree is eventually used to count the support of candidate itemsets (the *subset* function of Figure 1.3). We will discuss The latter step first.

The *subset* function works in the following way. It traverses the hash tree several times from the root down to the leaf nodes by hashing on every item appearing in the transaction. Unlike the naive

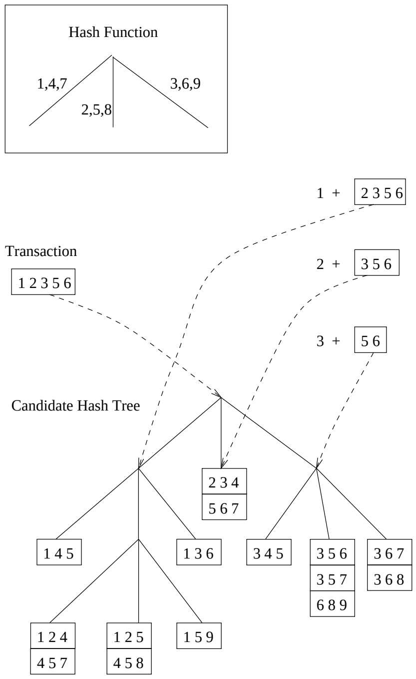

Figure 1.4: Subset operation on the root of a candidate hash tree.

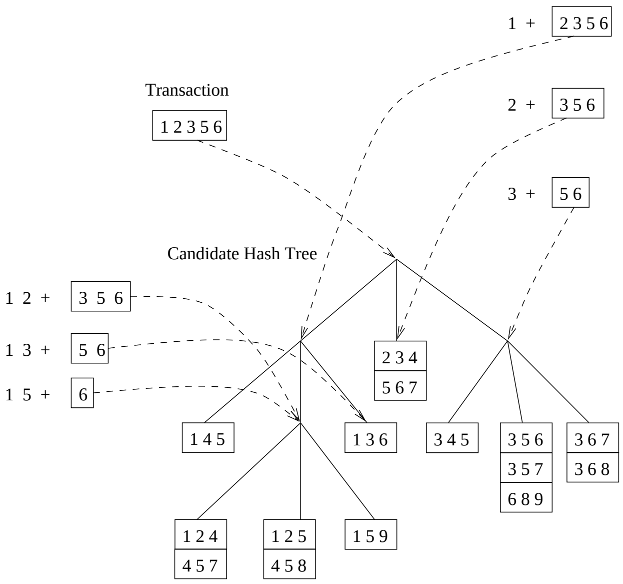

Figure 1.5: Subset operation on the left most subtree of the root of a candidate hash tree.

approach where all candidates are compared against every transaction, only the subset of leaf nodes reachable from the root via hashing will be checked for candidate itemsets. Whenever a leaf node is reached, all candidate itemsets stored at the node are compared against the transaction and their support counts updated accordingly. On the other hand, if we reach an interior node after hashing on item i of a transaction t, we will apply the hash function to every item that comes after i in t and repeat the procedure recursively on the corresponding child nodes.

Figure 1.4 shows an example of how the subset operation can be used to find candidates for transaction $\{1\ 2\ 3\ 5\ 6\}$. Also shown in this figure is the hash function used by the internal nodes of the tree: e.g. items 1, 4 and 7 are always hashed to the left child of the current node; items 2, 5, 8 to the middle child; and so on. At the root node, the hash function will be applied to all items in the transaction. For instance, item 1 is hashed to the left child node of the root, items 2 and 5 are hashed to the middle child of the root whereas items 3 and 6 are hashed to the right child. After hashing on item 1, the remaining items of the transaction, $\{2\ 3\ 5\ 6\}$, is applied recursively to the corresponding child node at depth 2. In doing so, items 2 and 5 are both hashed to the middle child node at depth 3 with their remaining items being $\{3\ 5\ 6\}$ and $\{6\}$ respectively (Figure 1.5). If we continue hashing from the middle child node on items 3 and 6, we would end up at the leaf node containing the candidate 3-itemset $\{1\ 5\ 9\}$. Since the candidate is not a subset of the original transaction, its support count is not updated. However, if we hash on item 5, we would reach the leaf node containing two 3-itemset $\{\{1\ 2\ 5\},\ \{4\ 5\ 8\}\}$.Only the support count for $\{1\ 2\ 5\}$ is incremented because it is a subset of the transaction.

To verify the correctness of this function, note that if an itemset c is contained in a transaction t, the first item of c must also be in t. Hence, by hashing on every item in t at the root node, we can ensure that the only child nodes we ignore will lead to itemsets that start with an item not in t. A similar argument applies to all internal nodes. Since the items are ordered, if we reach the internal node by hashing on item i, we only need to continue hashing on all items that occur after i in t.

So far, we have described how the hash tree can be used in the subset function. Now, let us see how it is constructed. It is important to note that a new hash tree is constructed at every pass of the algorithm (when new candidates are generated). Initially, the hash tree contains only a root node, which is a leaf with no candidate itemset. When a candidate itemset is generated, the items in the set are stored in sorted order. New candidate itemsets can be inserted into the leaf node as long as the total number of candidates at the node is less than the maximum allowable threshold. If the threshold is exceeded, the leaf node is converted to an interior node and child nodes are created for the new internal node. The candidate itemsets are distributed to the child nodes according to the hash values of the items. Following this, any new candidate itemset is inserted into the hash tree by hashing on each successive item at the internal nodes until a leaf node is reached. Again, if the maximum threshold of the leaf is exceeded, new internal nodes will be created. For example, the candidate item set $\{1\ 2\ 4\}$ is inserted by hashing item 1 at the root to reach the left child node of the root, hashing item 2 at that node to reach the middle child node, hashing item 4 to reach the left child node which is a leaf node.

The runtime for the entire algorithm is dominated by the counting process of the *subset* function. More precisely, according to the analysis presented in [13], at level k of the algorithm, the computation time required per transaction for visiting the hash tree is proportional to the number of candidate k-itemsets present in a transaction (N_{C_k}), and the expected number of distinct leaf nodes visited by the transaction. It can be shown that as the number of leaf nodes in hash tree grows larger, the runtime gets dominated more by N_{C_k}.

1.3.2 Other Frequent Itemset Generation Algorithms

Aside from *Apriori*, there have been many other algorithms proposed to efficiently generate frequent itemsets. In order to put the various algorithms in perspective, we need to take a look at the following issues.

1. **Traversal of Itemset Lattice:**

Earlier, it was shown that the generation of frequent itemsets conceptually represents a traversal on the itemset lattice structure (Figure 1.1). The goal of most association rule discovery algorithm is to generate frequent itemsets in the least amount of time by traversing the lattice in an effective manner. *Apriori* uses a level-wise traversal scheme, i.e., it first discovers all the size-1 frequent itemsets, followed by all the size-2 frequent itemsets, and so on. However, other traversal schemes do exist. For example, if we start our search at the itemset $\{ABCD\}$, gather its count, and find it to be frequent, then we can conclude that all the subsets of it must be frequent. We can stop our search at the part of lattice which comprises of all the subsets of $\{ABCD\}$. Next, we move on to $\{BCDE\}$, count it, and discover that it is infrequent. Then we might backtrack our search to all its subsets, and so on. A third possibility is to do a depth-first search. Here, we may choose to start at say itemset $\{A\}$, count it, and determine whether it is frequent. If it is, then we can keep going down the branch to $\{AB\}$, $\{ABC\}$, until we find an infrequent itemset, say $\{ABCD\}$. If that is the case, we can skip to the $\{ABCE\}$ branch, and continue our search from there.

2. **Representation and Traversal of Transaction Database:** Besides the various lattice traversal schemes, there also exist different ways in which the database can be scanned. One way to characterize the database traversal is by looking at the total number of passes the algorithm makes over the entire database. Depending on the database representation, making multiple passes over the database may incur more I/O costs.For example, consider the representation shown in Table 1.1, in which the set of items is stored for each transaction. This representation is called *horizontal* data layout. Alternatively, one can store the list of transaction ids (tid) for each of the items. This is refered to as the *vertical* data layout. The *Apriori* algorithm assumes a horizontal database layout and makes as many passes over the entire database as the length of the longest frequent itemset present in the transaction database. There may be other ways to reduce the I/O cost. For example, assume the depth-first search strategy is adopted, and the database is represented using the vertical layout. Then, as we traverse deeper down the branches of the lattice, say from $\{A\}$ to $\{A, B\}$, $\{A, B, C\}$, and so on, we can progressively shrink the tid-list for each itemset. After a while, the required tid-lists will start fitting into the main memory, and we may not need to go the disk every time we visit a child node. Thus, a proper algorithm design would require a careful combination of database representation, lattice search strategy, and database traversal techniques.

3. **Generate-and-Count Paradigm:** In principle, the strategy employed by *Apriori* was to first generate the *candidate* itemsets, before counting the actual support of these candidates. Several other algorithms also rely on the generate-and-count strategy. But, this paradigm may not be the only efficient way to discover the frequent itemsets. What if we can gather all the counts at the beginning and then use or modify them to guide our search through the itemset lattice? These precise issues have given rise to a recent class of algorithms that represent the transaction database in a compact fashion and directly infer the frequent itemsets from it.

Over the years, various algorithms have been proposed to address some of the issues described above. These algorithms include the DHP algorithm [3],, Tree Projection algorithms [10, 11], PARTITION [4], the sampling-based algorithms [6], a a set of algorithms proposed in [5], a family of algorithms proposed in [8], the DIC algorithm [9], the FP-tree based algorithm [12], and a trie-based algorithm [7]. A review of these algorithms is given in the Appendix.

1.4 Association Rule Generation.

A straightforward way to generate association rules from a frequent itemset L is to find all non-empty subsets, $l \subset L$, such that $l \Longrightarrow L - l$ satisfies the minimum confidence requirement.

Example 7 : Suppose $L = \{1, 2, 3\}$ is a frequent itemset. The set of all possible rules that can be generated from this frequent itemset are $\{1, 2\} \Longrightarrow \{3\}$, $\{1, 3\} \Longrightarrow \{2\}$, $\{2, 3\} \Longrightarrow \{1\}$, $\{1\} \Longrightarrow \{2, 3\}$, $\{2\} \Longrightarrow \{1, 3\}$ and $\{3\} \Longrightarrow \{1, 2\}$.

The question is, can we find a similar pruning strategy based on some monotonicity property of the confidence measure? Or, does such property even exist?

1.4.1 Rule Generation in Apriori

It is important to know some of the basic principles underlying the generation of rules from a given frequent itemset :

1. All rules must include every item from the frequent itemset. In the above example, rules such as $\{1\} \Longrightarrow \{2\}$ are not generated from L.
2. All rules generated from the same frequent itemset have the same support.
3. Confidence of a highly supported rule can always be computed using the support of frequent itemsets alone. For example, if the rule $\{1,2\} \Longrightarrow \{3\}$ is generated from the frequent itemset $\{1,2,3\}$, then its confidence can be computed using the support for $\{1,2,3\}$ and $\{1,2\}$. The itemset $\{1,2\}$ must also be frequent due to the anti-monotone property of support. Since the support of all frequent itemsets are known, the confidence of a rule can always be determined without scanning the entire database again. This is why rule generation is not as expensive as frequent itemset generation.
4. If a rule $l \Longrightarrow L-l$ does not satisfy the minimum confidence threshold, then having any itemset $a \subset l$ as the premise will not produce rules with sufficient confidence. For instance, if $\{1,2\} \Longrightarrow \{3\}$ is below the confidence threshold, so will $\{1\} \Longrightarrow \{2,3\}$. This is because the numerator term of confidence measure is the same for both $a \Longrightarrow L-a$ and $l \Longrightarrow L-l$; but the support for a is greater than or equal to l. Therefore, the confidence for the former rule will always be less than or equal to the confidence of the latter. Based on this property, it follows that for a rule $L-l \Longrightarrow l$ to hold, all rules of the form $L-a \Longrightarrow a$ must have at least $minconf$, where a is a non-empty subset of l.

The last principle exhibits the anti-monotonicity property of confidence on the number of items in the rule consequent. These principles have been used extensively by the Apriori rule generation algorithm in [2]. Basically, for each frequent itemset, the algorithm would first find all highly confident rules having k-item consequents. These k-item consequents are then used to determine the candidate $k+1$-item consequents using the same apriori-gen function as frequent itemset generation. The rest of the algorithm is summarized in Table 1.6.

1.4.2 Variations and Generalizations of Association Rule Mining

In this section, we will see that association rules are not the only kind of inter-item relationship that can be extracted from a transaction database. Under certain circumstances, other kinds of relationships can be more useful or complementary to association rules. We will briefly examine three variations of these patterns: dependence rules, negative association rules and indirect association. We will also look at the different ways to generalize the original concept of association rules in order to incorporate non-binary data and multi-level product hierarchies.

A. Variation of Association Patterns

1. **Dependence Rules:**

   So far, we have learnt that confidence is one of the measures used to filter uninteresting association rules. The confidence of a rule $X \Longrightarrow y$ can be simply looked at as a *conditional probability for the presence of y given X*. However, there are situations in which the confidence measure may be misleading. We will illustrate this with the following example.

   At a computer store, out of 200 buyers of operating systems, suppose 160 of them bought Windows operating system and there were 100 customers who bought Linux operating system. Of the Linux buyers, 60 of them also bought the Windows operating system. Now a rule $Linux \longrightarrow Windows$ has

```
forall large k-itemsets l_k, k ≥ 2 do
    H_1 = { consequents of rules derived from I_k with one item in the consequent };
    call ap-genrules(I_k, H_1);


procedure ap-genrules(l_k: large k-itemset, H_m: set of m-item consequents)
if (k > m + 1) then {
    H_{m+1} = apriori_gen(H_m);
    forall h_{m+1} ∈ H_{m+1} {
        conf = σ(l_k)/σ(l_k − h_{m+1});
        if (conf ≥ minconf) then
            output the rule (l_k − h_{m+1}) ⟹ h_{m+1};
        else
            delete h_{m+1} from H_{m+1};
    }
    call ap-genrules(l_k, H_{m+1});
}
```

Figure 1.6: Algorithm for generating rules from frequent itemsets.

sufficiently large support (30%) and also sufficiently high confidence (60%). So, looking at this rule, one might conclude that whenever Linux is sold, Windows is also sold with 60% chance. Let us examine the situation from a different angle. If we didn't know anything about a customer, the probability that he or she will buy Windows is 160/200 = 80%. However, the moment we know that a customer has bought Linux, her probability of purchasing Windows has gone *down* from 80% to 60%. This implies that although the rule $Linux \longrightarrow Windows$ has sufficient confidence, it is misleading. Here is the reason. Let us assume that the store-owner decides to the push the sales for Windows up by 200 by sending ads to customers. If the owner was to follow the prior probability of Windows, then he or she would need to mail ads to 200/0.8 = 250 customers. Whereas, if the decision was based on the $Linux \longrightarrow Windows$ rule, then the owner would need to increase the Linux sale by 200/0.6 = 333, by identifying and attracting 333 more Linux customers. Moreover, following the first decision option, the store-owner needs to identify the profile of the store's generic customers, whereas for second option, he or she has to identify profiles of Linux customers, which may be more difficult. On the other hand, if the confidence of the rule $Linux \longrightarrow Windows$ was more than the prior probability of Windows, then following this rule would have made more sense, rather than merely relying on the prior probability.

This example stresses the need to get more information than mere conditional probability (confidence) of a rule. The impact of conditioning a presence of some item on presence of others can be judged by looking at whether the conditional probability is higher than or lower than or same as the prior probability of that item. This notion is precisely captured by *statistical dependence*. In statistics, a set of *events* {x,y} is said to be independent, if their joint probability is the product of their individual probabilities; i.e., $P(x,y) = P(x)P(y)$. In the example above, the events were Product_Sold = Linux or Product_sold = Windows. If the events are not independent, we can look at the ratio $P(x,y)/P(x)P(y)$ If the ratio is more than 1, then events are said to have *positive* dependence, which means that conditioning the presence of any of the events (x or y), on the presence of other, causes increase in the prior probability of occurrence of that event. This can be seen by rewriting the equation $P(x,y)/P(x)P(y) > 1$ as $P(x,y)/P(x) > P(y)$. The left hand side of the last equation is the confidence $(P(y\|x))$ of the rule $x \rightarrow y$ rule, which conditions the presence of y on presence of x. The right hand side is the prior probabiity of y. Similarly, *negative* dependence means that $P(x,y)/P(x)P(y) < 1$, and can be interpreted as decrease in prior probability of either of the events after conditioning its presence on the other's presence.

Thus, statistical dependence yields a possibly more useful information between two events, rather than merely looking at the confidence of a rule. Essentially, statistical dependence combines the prior probabilities and conditional probabilities into a single measure.

2. **Negative Association Rules:**

   So far, the notion of association rules introduced in this chapter assumes that presence of items in a transaction is more important than its absence. However, there could be situations in which it would be advantageous to derive rules involving absence of certain items (or features) from a transaction. An example of such a rule might be "70who buy televisions do not purchase a DVD player." Such information could be useful for marketing managers when devising new marketing strategies.

   The biggest challenge in mining this type of pattern is there could be a huge number of uninteresting negative associations in the data. This is because for large dimensional data sets with sparse transactions, it is more likely that two items are negatively associated rather than positively associated with each other. As a result, by looking for lowly supported itemsets or rules, one may end up generating a very large number of item combinations.

   Savasere et al.[14] have proposed a method to prune out spurious negative assocation rules. Basically, they have suggested that a negative association rule is interesting if it deviates significantly from its expectation based on prior knowledge. For example, suppose the support for Ruffles chips and Coke is high, then they expect the support for Ruffles chips and Pepsi would be high too. This argument is based on the assumption that items belonging to the same product category are expected to have similar types of associations with other items. If instead, we found the actual support between Ruffles chips and Pepsi to be significantly lower than expected, then we believe that Ruffles chips and Pepsi forms an interesting negative association. This approach requires a specification of item taxonomy along with algorithms for mining frequent itemsets from multiple levels of the taxonomy (which will be described below). By generating the higher level frequent itemsets, one can compute the expected support for their descendent itemsets. Any descendents whose actual support is significantly different from its expected value are considered to be negative candidate itemsets.

3. **Indirect Association:**

   Association rule mining would discover rules involving items that occur frequently together in the transaction database. Any itemsets that fail the support threshold criteria will be automatically removed. This action is justifiable since we are only interested in finding items that are directly associated with each other. However, under certain situations, it is possible that some of the lowly-supported itemsets can be interesting due to the presence of higher-order dependencies[2] among the items. Consider a pair of items, (a, b), having a low joint support value (Figure 1.7). Let Y be a set that contains items that are highly dependent on both a and b. This figure illustrates an indirect association between a and b via a mediator set, Y. Unlike direct association, an indirect relationship is characterized not only by the two participants, a and b, but also by the items mediating this interaction.

   One can think of many potential applications for indirect association. In the market basket scenario, a and b may represent competing products. Analysts may be interested in finding what are the common products (Y) bought together by customers who purchased the two competing products. With this information, they can then performed an in-depth analysis on why customers prefer one brand over the other based on the strength of dependencies between Y with a and b. Another potential application for indirect association is in textual data mining. For text documents, indirectly associated words may represent synonyms and antonyms or terms that appear in different contexts of another word. For example, in a collection of news articles, the words growth and decline may be indirectly associated via the word economy. Another example is the pair soviet and worker may refer to the different contexts in which the word union is being used. For stock market data, a and b may refer to disjoint events that will partition event Y. For example, the event $Y = Microsoft - Up$ can be partitioned into

[2]direct association between items are considered to be first-order dependencies.

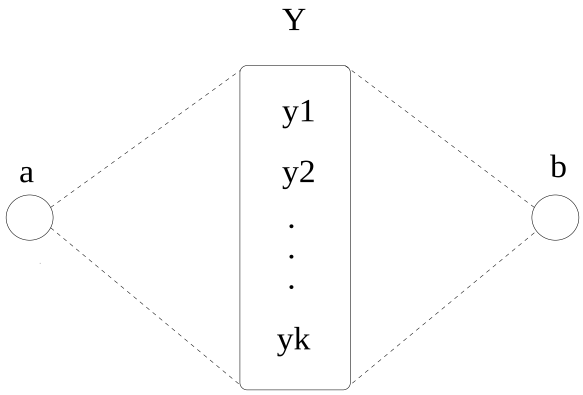

Figure 1.7: Indirect association between a pair of items.

$a = Redhat - Down$ and $b = Intel - Up$ where a denotes the event where the competitor's stock value is down whereas b may represent the event in which most of the technology stocks for large corporations are up.

[15] proposed a general framework for defining indirect association in various application domains. Their formal definition of indirect association is :

Definition 3 : The pair a, b is indirectly associated via a mediator Y if the following conditions hold :

(a) $\sigma(a, b) < t_s$ (Itempair Support condition).

(b) $\exists Y \neq \emptyset$ such that $\forall y_i \in Y$:

 i. $sup(a, y_i) \geq t_f, sup(b, y_i) \geq t_f$ (Mediator Support condition).

 ii. $d(a, y_i) \geq t_d, d(b, y_i) \geq t_d$ where $d(p, q)$ is a measure of the dependence between p and q (Dependence condition).

In order to discover indirectly associated items, one must address the following questions :

(a) What is a good measure of dependence between items?

(b) What constitutes the elements of the mediator?

(c) How to define the overall indirect association measure between a and b?

The first question is related to the issue of finding good interestingness measures for itemsets (or association rules) which will be described in Section 1.5. The second question deals with the problem of defining mediating elements either as single items, single itemsets or multiple itemsets. The last question allows one to rank the discovered itempairs according to the strength of their indirect association. [15] described several ways to tackle these issues and presented a simple algorithm for mining indirect itempairs.

B. Generalization of Association Rules.

1. **Association Rules with Item Taxonomy:**

 There are several reasons why it is useful to incorporate item taxonomy into the association rule discovery problem :

 (a) Users may be interested in finding rules that span over multiple levels of the taxonomy.

 (b) Rules at lower levels of the item taxonomy may not have sufficient support.

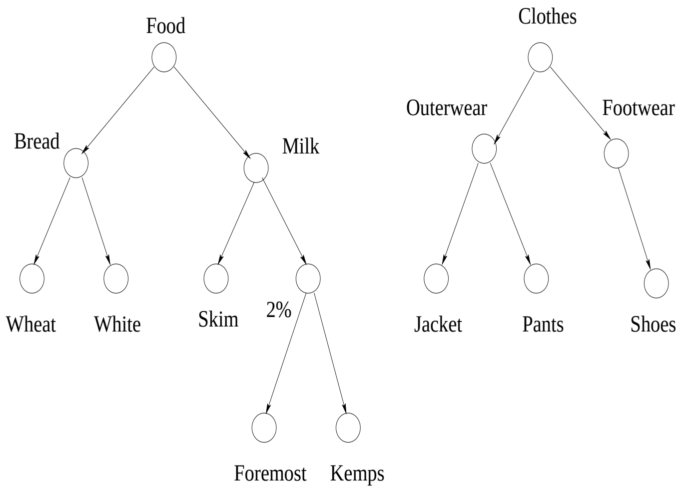

Figure 1.8: Example of an item taxonomy..

(c) Item taxonomy can be used to summarize the discovered rules.

Note that one cannot directly infer rules involving items at higher levels of the taxonomy based on the rules discovered for lower level items. Let us investigate how the support and confidence measures change as we traverse the product hierarchy level-wise. Let G denotes a directed acyclic graph representing the set of item taxonomies (Figure 1.8). If there is an edge in G from p to c, we call p a *parent* of c and c a *child* of p. For example, if "Milk" is a parent of "Skim Milk", then there is a directed edge from the node Milk to Skim Milk in G. Furthermore, we say $\hat{X}$ is an ancestor of X if there is a path from $\hat{X}$ to X in G. The properties of support and confidence at different levels of the taxonomy are as follows :

- if $\hat{X}$ is the parent for both X_1 and X_2, then $\sigma(X)$ may not be equal to $\sigma(X_1) + \sigma(X_2)$.
- if $\sigma(X \cup Y) \geq s$, then $\sigma(\hat{X} \cup Y) \geq s$, $\sigma(X \cup \hat{Y}) \geq s$ and $\sigma(\hat{X} \cup \hat{Y}) \geq s$.
- if $conf(X \Longrightarrow Y) \geq \alpha$, then only $conf(X \Longrightarrow \hat{Y}) \geq \alpha$ but not $conf(\hat{X} \Longrightarrow Y)$ and $conf(\hat{X} \Longrightarrow \hat{Y})$.

There are several difficulties in applying the standard Apriori-like technique directly to this problem :

(a) Items at high levels of the taxonomy tend to have significantly larger support compare to those at lower levels. Thus, a single minimum support threshold may not be appropriate for this situation.

(b) By introducing item taxonomy, we have increased the dimensionality of the feature space. This will increase the complexity of the problem as well as the number of rules that are being generated. The latter problem must be addressed for efficiency reasons since many of these rules are often redundant (eg. 2% $Lowfat\ Milk \Longrightarrow Milk$).

[16] and [17] are two of the earliest work that have attempted to deal with these issues. In [16], an algorithm was proposed to progressively discover itemsets at multiple levels of the item taxonomy using a top-down approach. Initially, all large itemsets of size-1 at the top level of the taxonomy are generated. These itemsets will be denoted as $L(1,1)$. Next, using $L(1,1)$, we derive all the large itemsets of size 2 at level 1 ($L(1,2)$). The algorithm for generating these itemsets is basically the same as the Apriori algorithm given in Figure 1.3. This step will be repeated until all level-1 large itemsets, $L(1,k)$ ($k > 1$), are found. The algorithm then moved on to the next level of the taxonomy. Only the descendents of large itemsets in $L(1,1)$ will be considered as candidates for $L(2,1)$. The algorithm then proceeds to generate $L(2,2), L(2,3) \cdots$ sequentially until all level-2 large itemsets are discovered. Subsequently,

it will progress to the next level and so on, until it terminates at the deepest level requested by the user. [16] used a special encoding of the transaction database to improve the performance of their algorithm. Basically, each item in the database is represented by a unique identifier, based on the item taxonomy. For example, the item 2% *Foremost milk* can be encoded as 1322 where the leftmost digit '1' represents the food category; followed by the digit '3' representing the product type, milk; and the digit '2' representing the product content i.e. '2% *milk*' and finally, the digit '2' for the product brand, 'Foremost'. With this method of encoding, all level-1 items are represented as 'd***', level-2 items represented as 'dd**', and so on. Furthermore, each time $L(k, 1)$ is generated, the transaction database is filtered to remove all level-k items that are infrequent and all transactions that contain only infrequent items. This is possible because the support for lower-level items is always smaller than their ancestors. Nevertheless. the algorithm proposed in [16] would discover only rules that reside at the same level of the hierarchy.

[17] introduced a more general framework for mining association rules with item taxonomy. In this framework, association rules may involve items from any level of the taxonomy. Their algorithm works in the following way : initially, each transaction t is replaced by an *extended transaction* t' which contains all the items in t as well as the ancestors of the items in t. For example, the transaction t = {jacket, wheat bread} will be extended to t' = {jacket, wheat bread, outerwear, clothes, bread, food} where outerwear and clothes are ancestors of jacket while bread and food are ancestors of wheat bread. We then run the Apriori algorithm on this extended transaction database to obtain rules that span over multiple levels of the taxonomy. However, the execution of this algorithm can be rather slow due to the additional number of features. [17] have introduced several optimization methods in order to improve the performance of this basic algorithm. For example, one can prune any itemsets containing an item, x, and its ancestor, $\hat{X}$. This is because the support of the itemset remains unchanged even if we remove $\hat{X}$ from the set. Moreover, if L_k, the set of frequent k-itemsets, does not include any itemset that contain both x and $\hat{X}$, then any candidates C_{k+1} generated cannot include these items as well.

2. **Association Rules for Numerical Variables:**

So far, we have described the association rule problem in the context of boolean attributes. We now turn to the problem of mining association rules for numeric (quantitative) attributes. An example of a quantitative association rule is :

Example 8 : $(Salary \in [30K, 45K])$ and $(Married = No) \Longrightarrow (NumHouses = 1)$

Example 9 : $(Temperature > 90F)$ and $(Pressure > 2.0\ atm) \Longrightarrow (Valve = Closed)$

Note that both examples above contain a mixture of numeric, categorical and boolean attributes. [18] presented an approach to generalize the notion of association rules to include both quantitative and categorical attributes. Categorical attributes can be handled by replacing each such attribute with as many fields as the number of attribute values. For example, if the $Valve$ attribute has three possible values, say *Closed*, *Released* and *Broken*, it will be replaced by 3 new fields, $Valve = Closed$, $Valve = Released$ and $Valve = Broken$ in the transaction database. In fact, the attribute values themselves are mapped into unique numeric identifiers (say, Valve=1 instead of Valve=Closed). This allows them to handle both categorical and numeric attributes in the same manner. For numeric attributes, a novel technique is employed to partition the range of the numeric attribute into discrete intervals. The initial attribute set I is generalized to $I_R = \{< x, l, u > | x \in I, l \in R^1, u \in R^1\}$ where each attribute x is represented by a triple, $< x, l, u >$, R^1 is the set of real numbers and $[l, u]$ represents the interval of possible values for x. Categorical (and boolean) attributes are also represented in the same way, with $l = u$.

Definition 4 : A *quantitative* association rule is an implication of the form $X \Longrightarrow Y$, where $X \subset I_R$, $Y \subset I_R$ and $X \cap Y = \emptyset$. The problem of mining quantitative association rules is to discover all association rules that have support and confidence greater than the user-specified minimum support and minimum confidence.

[18] mentioned two difficulties in discretizing the quantitative attribute :

(a) The support for a single interval of an attribute can be low if the number of intervals are large. In other words, if the size of the interval is too small, we may not find any rules involving this attribute.

(b) If the granularity of the interval is too coarse, some rules may fail the confidence threshold. [18] called this the information loss problem. For example, the rule $Salary \in [30K, 45k] \Longrightarrow OwnCar = Yes$ may have high confidence. But, the rule $Salary \in [0K, 45K] \Longrightarrow OwnCar = Yes$ may fail the confidence threshold.

To prevent the first problem, adjacent intervals are allowed to be combined together. Nevertheless, this approach still suffers from the information loss problem. As a result, a measure was introduced in [18] to quantify the amount of information loss due to discretization. This measure was then used them to determine the appropriate size and number of partitioned intervals. Once the partitioning was done, the algorithm for discovering frequent itemsets have similar structure to Apriori. However, the procedure for generating candidate sets must deal with the issue of handling and combining quantitative intervals, pruning redundant candidate itemsets and efficient counting of the support for candidate sets. Other discretization methods have also been proposed. For example, [19] suggested the use of clustering techniques to improve the quality of partitioning quantitative attributes.

[20] gave an example in which Definition 4 may be incomplete at characterizing meaningful quantitative rules. For instance, the rule $(Height \in [100cm, 150cm]) \Longrightarrow (age \in [0, 14])$ may have sufficient support and confidence even though very few children, under the age of one are 100 cm tall. One way to deal with this problem is by using a maximum support threshold for a rule. Although this may partially solve this problem, it still does not address the problem due to uneven data distribution. Instead, [20] proposed a new way to define association rules, taking into account the distribution of values for quantitative attributes. An example of such a rule is :

Example 10 : $Region = Midwest \Longrightarrow Salary : mean = 9.50/hr$ (Overall mean Salary$= 10.50/hr$).

which says that the average salary for an employee living in the Midwest region is lower than the average salary for the whole nation. Essentially, [20] proposed the following structure for an association rule :

$$\text{population subset} \Longrightarrow \text{statistical measures of subset}$$

where the statistical measure can be in the form of mean, variance, median, etc. A rule is interesting only if the statistical measure of the subset is significantly different from the rest of the population.