7. REVIEW OF LITERATURE

Many algorithms are available for transactional database having many rows/columns. Among these, we can filter out most useful methods which we can categorize them as scalable methods for mining frequent patterns. Four key frequent pattern mining techniques are: Apriori\textsuperscript{[2]}, DHP\textsuperscript{[3]}, FP-Growth\textsuperscript{[4]}, ECLAT\textsuperscript{[5]} and these have been projected to attempt through transaction DB. Also, all are contradicted with altogether.

7.1 Apriori

The first algorithm to solve frequent itemset mining problem was proposed, later denoted as AIS. R. Agrawal and R. Srikant improved AIS in very short time and called this improved one as Apriori\textsuperscript{[24]}. Apriori is a decisive algorithm, where level by level search is followed in an iterative manner. So, to explore (k+1)-itemsets, k-itemsets are used.

Apriori algorithm is proposed in support of mining recurrent itemsets with the concept of Boolean association rules. It operates on databases having transactions to study the association rules\textsuperscript{[2, 24]}.

Apriori is a key technique anticipated for finding frequent patterns\textsuperscript{[2]}. It continues through recognizing the recurrent individual patterns in DB and enlarge these to higher and bigger patterns just like often appear in DB adequately\textsuperscript{[2]}. Apriori pursue the given steps: (i) firstly, check the DB to obtain frequent 1-pattern, (ii) creates the size of (k+1) candidate patterns from size of k frequent patterns, (iii) scan the DB for the found patterns and lastly (iv) finish while no frequent or candidate groups of items could be created\textsuperscript{[2, 28]}. Apriori utilizes a repetitive group identified as a level-wise seek, there k-patterns are utilized to discover (k+1)-patterns\textsuperscript{[2]}. Pruning on candidate itemsets, is performed to remove infrequent sub-itemsets. Frequent patterns are generated by referring candidates with a comparison of minimum support and support count of each candidate\textsuperscript{[34]}. The overall and detailed processes are described in Fig. 7.1 and Fig. 7.4 respectively.

To shrink the search space, Apriori employs “Apriori Property”: “All nonempty subsets of a frequent itemset must also be frequent”\textsuperscript{[1, 28]}.
\[ P(I) < \text{min\_sup} \Rightarrow I \quad \text{i.e. infrequent.} \]
\[ P(I+A) < \text{min\_sup} \Rightarrow I+A \quad \text{i.e. infrequent either.} \]

Anti-monotone property – “if a set cannot pass a test, all of its supersets will fail the same test as well”[26].

**Definitions:**

Itembase: “Let \( B = \{i_1, \ldots, i_m\} \) being a set of items. This set is called the item base. Items may be products, special equipment items, service options etc.”

Itemset: “Any subset \( I \subseteq B \) is called an itemset. An itemset may be any set of products that can be bought together”.

K-itemset: “an itemset which consists of \( k \) items”.

Frequent itemset: “an itemset with sufficient support”.

\( L_k \): “a set of frequent k-itemsets”.

\( C_k \): “a set of candidate k-itemsets”.

Apriori property: “if an item \( X \) is joined with item \( Y \), \( \text{Support}(X \cup Y) = \min(\text{Support}(X), \text{Support}(Y)) \)”.

Negative border: “an itemset is in the negative border if it is infrequent but all its “neighbors” in the candidate itemset are frequent.”

Interesting rules: “strong rules for which antecedent and consequent are dependent”[36].
Figure 7.1 Apriori – Frequent Itemset Generation Process\cite{1,29}
L₁ = \{find frequent large 1-itemsets\}
for (k = 2; L_{k-1} \neq \phi; k++) {
    C_k = \text{apriori-gen}(L_{k-1});
    \text{for each transaction } t \in D \{
        // Scan D for counts
        C_t = \text{subset}(C_k, t)
        // get the subsets of } t \text{ that are candidate}
        \text{for all candidates } c \in C_t \text{ do}
        \quad c.\text{count}++;
    \}
    L_k = \{ c \in C_k | c.\text{count} \geq \text{minsup}\}
}\}
return L = \bigcup_k L_k;

**Candidate Generation: Join Step**
insert into C_k
select p.item₁, p.item₂, p.item_{k-1}, q.item_{k-1}
from L_{k-1} p, L_{k-1} q
where p.item₁ = q.item₁, ..., p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}

**Candidate Generation: Prune Step**
for all itemsets } c \in C_k \text{ do}
    \text{for all } (k-1)-\text{subsets } s \text{ of } c \text{ do}
    \quad \text{if } (s \notin L_{k-1}) \text{ then}
    \quad \quad \text{delete } c \text{ from } C_k

\text{Figure 7.2 Apriori Algorithm}^{[1,2]}
Many techniques use concept of Apriori to generate frequent patterns. By referring Fig. 7.2 of the algorithm, it is found that there are 3 basic steps: (i) Generate and Test (ii) Join Step and (iii) Prune Step[^2, ^34].

**Generate and Test:** L₁ is generated from C₁ by removing infrequent itemsets. As a result, 1-itemsets will be found.

**Join Step:** Follows process like
1. Lₖ₋₁ is joined itself to generate Cₖ, number of k-itemsets.
2. Lexicographic order is followed for items to be arranged in each transaction[^23].
3. For the (k-1) itemset: l₁[1]<l₁[2]..<l₁[k-1].
   - I. The members of Lₖ₋₁ are joinable if their first (k-2) items are in common.
   - II. Members l₁, l₂ of Lₖ₋₁ are joined if (l₁[1]=l₂[1]) and (l₁[2]=l₂[2]) and (l₁[k-2]=l₂[k-2]) and (l₁[k-1]<l₂[k-1]) – no duplicates.
4. The resulting itemset formed by joining l₁ and l₂ is l₁[1], l₁[2],..., l₁[k-2], l₁[k-1], l₂[k-1].

**Prune Step:**
1. Cₖ is a superset of Lₖ, Lₖ contains those candidates from Cₖ, which are frequent.
2. Scanning the database to determine the count of each candidate in Cₖ—heavy computation.
3. To reduce the size of Cₖ, the Apriori property is used.

Reflect on a demonstration for joining and pruning: Let L₃={ wxy, wxz, wyz, wyt, xyz }, focusing on self-joining: L₃*L₃ wxyz from wxy and wxz, wyzt from wyz and wyt. Many people can work on pruning. For pruning: wyzt is removed because wzt is not in L₃ and C₄ will be {wxyz}[^2, ^29].
Example 1: Apriori technique, with pursuing a case of a simple DB with 9 transactions and minimum support is 2, is discussed in Fig. 7.3.

### Figure 7.3 Apriori Example

**Database TDB**

<table>
<thead>
<tr>
<th>TID</th>
<th>List of ItemIDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>T100</td>
<td>11,12,13</td>
</tr>
<tr>
<td>T200</td>
<td>12,14</td>
</tr>
<tr>
<td>T300</td>
<td>12,13</td>
</tr>
<tr>
<td>T400</td>
<td>11,12,14</td>
</tr>
<tr>
<td>T500</td>
<td>11,13</td>
</tr>
<tr>
<td>T600</td>
<td>12,13</td>
</tr>
<tr>
<td>T700</td>
<td>11,13</td>
</tr>
<tr>
<td>T800</td>
<td>11,12,13,15</td>
</tr>
<tr>
<td>T900</td>
<td>11,12,13</td>
</tr>
</tbody>
</table>

**Support Count (Sup)**

- C1: (11) 6
  - Scan D for count of each candidate

- L1: (11) 6, (12) 7, (13) 6, (14) 2
  - Compare Candidate Support Count with minimum support count
  - Generate C2 candidates from L1

**C2**

- (11,12) 4
- (11,13) 4
- (11,14) 1
- (11,15) 2
- (12,13) 4
- (12,14) 2
- (12,15) 2
- (13,14) 0
- (13,15) 1
- (14,15) 0

- Scan D for count of each candidate

**C3**

- (11,12,13) 2
- (11,12,15) 2
- (11,13,15) 1
- (12,13,14) 0
- (12,13,15) 1
- (12,14,15) 0

- Compare Candidate Support Count with minimum support count

**L2**

- (11,12,13) 2
- (11,12,15) 2
- (11,13,15) 1
- (12,13,14) 0
- (12,13,15) 1
- (12,14,15) 0

- Compare Candidate Support Count with minimum support count

**L3**

- (11,12,13) 2
- (11,12,15) 2
Start

Scan the transaction database to get support S of each item

S >= Min_Sup

Yes

Add to frequent 1-itemsets, L_1

Use L_{K-1} JOIN L_{K-1} to generate a set of candidate K-itemsets.

Scan the transaction database to get support S of each candidate K-itemset

S >= min_sup

Yes

Add to K-Frequent Itemsets, L_K

Generate Set (L_K) = NULL

No

Yes

For each frequent itemset L, generate all non-empty subsets of L

For each non-empty subset s of L, find confidence C

C >= Min_Confidence

Yes

Add to strong rules

Figure 7.4 Apriori Algorithm Role in Strong Rules Identification\textsuperscript{[1, 29]}
Fig. 7.5 illustrates the execution tree of the join-based Apriori algorithm over the transaction database mentioned in Fig. 7.3 for minimum support value 2\(^{25}\). As mentioned in the algorithm of Apriori, a candidate k-patterns are generated by joining two frequent itemset of size (k-1). Let’s consider an example, at level 3, the pattern \{I1, I2, I3\} is generated by joining \{I1, I2\} and \{I1, I3\}. After generating the candidate patterns, the support of the patterns is computed by scanning every transaction in the database and determining the
frequent ones. In Fig. 7.5, a candidate pattern is shown in a box along with its support value. A frequent candidate is shown in a solid box, and an infrequent candidate is shown in a dotted box. An edge represents the join relationship between size k candidate and size (k-1) frequent pattern. The Fig. 7.5 also illustrates the fact that a pair of frequent patterns is used to generate a candidate pattern, whereas no candidates are generated from an infrequent pattern.

**Example Explanation:**

We had \( L = \{\{I1\}, \{I2\}, \{I3\}, \{I4\}, \{I5\}, \{I1,I2\}, \{I1,I3\}, \{I1,I5\}, \{I2,I3\}, \{I2,I4\}, \{I2,I5\}, \{I1,I2,I3\}, \{I1,I2,I5\}\}.

1. Let’s take \( l = \{I1,I2,I5\} \).
2. Its all nonempty subsets are \( \{I1,I2\}, \{I1,I5\}, \{I2,I5\}, \{I1\}, \{I2\}, \{I5\} \).
3. Let minimum confidence threshold is, say 70%.
4. The resulting association rules are shown below, each listed with its confidence.

   I. \( R1: I1 \land I2 \Rightarrow I5 \)
      
      - Confidence = \( \frac{sc\{I1,I2,I5\}}{sc\{I1,I2\}} = \frac{2}{4} = 50\% \)
      - \( R1 \) is rejected.

   II. \( R2: I1 \land I5 \Rightarrow I2 \)
      
      - Confidence = \( \frac{sc\{I1,I2,I5\}}{sc\{I1,I5\}} = \frac{2}{2} = 100\% \)
      - \( R2 \) is selected.

   III. \( R3: I2 \land I5 \Rightarrow I1 \)
      
      - Confidence = \( \frac{sc\{I1,I2,I5\}}{sc\{I2,I5\}} = \frac{2}{2} = 100\% \)
      - \( R3 \) is selected.

   IV. \( R4: I1 \Rightarrow I2 \land I5 \)
Confidence = \frac{\text{sc}\{I1,I2,I5\}}{\text{sc}\{I1\}} = \frac{2}{6} = 33\%

R4 is rejected.

V. R5: I2 \rightarrow I1 \land I5

Confidence = \frac{\text{sc}\{I1,I2,I5\}}{\text{I2}} = \frac{2}{7} = 29\%

R5 is rejected.

VI. R6: I5 \rightarrow I1 \land I2

Confidence = \frac{\text{sc}\{I1,I2,I5\}}{\text{I5}} = \frac{2}{2} = 100\%

R6 is selected.

In this way, three strong association rules have been found.

Apriori algorithm can be useful as the base technique\textsuperscript{[2]}, on which many researches are done, and improvements are suggested in the general case as well as a specific subset of the applicable data. Due to the huge amount of data that is mined in the present applications, even a small performance gain on the algorithm will result in a considerable amount of throughput gain. Some enhancements to Apriori algorithm sacrifice the accuracy for a better response time. Apriori algorithm has some causes\textsuperscript{[1, 6, 8, 11, 19, 26]} like

1. Assumes transaction database is memory resident.
2. Huge number of database scans for generating large itemset and doing support count,
3. Large number of candidate generation.

Let’s consider F as sales of item “a” and N as sales of item “b”.

(i) Rule F-> … : “tells you what products will be affected if F is affected.”
(ii) Rule … -> N : “tells you what needs to be done so that N is affected.”
(iii) Rule F … ->N : “tells you what to combine with F to affect N.”

Considering sequential pattern finding (“association rules in time”), here a case: Keyboard -> Mouse
Assume min_sup = 40%, min_confidence = 60%. Possibility stands for supermarket is shown in Table 7.1.

**Table 7.1 Association Rule mining – Basic Data Conceptualization**

<table>
<thead>
<tr>
<th></th>
<th>Buy Mouse</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Buy Keyboard</td>
<td>2000</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>1750</td>
<td>250</td>
</tr>
<tr>
<td>Total</td>
<td>3750</td>
<td>1250</td>
</tr>
</tbody>
</table>

The strong rule is “Buy Keyboard -> Buy Mouse”. Since support(Buy Keyboard & Buy Mouse) = 2000/5000 = 2/5 (>=min_sup)\[^{36}\].

support(Buy Keyboard) = 3000/5000.

Confidence(RULE) = 2/3 > min_confidence.

Apriori algorithm is used as a recommendation engine in an E-commerce system. Based on each visitor's purchase history, the system recommends related and potentially interesting products. It is also used as the basis for a CRM system as it allows the company itself to follow-up on customer's purchases and to recommend other products by e-mail\[^{1, 19}\].

A government application is proposed for Apriori\[^{21}\]. The problem is connected to the management of the risk associated with social security clients in Australia. The problem is confirmed as a sequence mining task. The action ability of the model obtained is an essential concern of the authors. They concentrate on the difficult issue of performing an evaluation taking both technical and business interestingness into account\[^{21}\].

We assured that Apriori technique effectively searches all frequents patterns from DB. However, increase in DB size with larger number of items then (i) consumes large space as well as I/O time would increase, (ii) DB access/scan is enlarged. This results in large computational cost\[^{22}\].

Many techniques have been emerged to overcome the computational cost of Apriori. Apriori algorithm can be improved by: (i) reducing passes of transaction database scans, (ii) shrinking the number of candidates and (iii)
facilitating support counting of candidates\cite{20}.

**Implementation:**

Apriori algorithm implementation summary using Java sample code is given here.

```java
find_frequent_itemsets(long m_sup1)
{
    //C: Candidate itemset of size k
    //L : frequent large itemset of size k
    int k=1;   // Initially k=1
    min_sup = m_sup;
    LinkedList<FrequentItem> L;
    LinkedList<FrequentItem> C=null;
    L = Find_frequent_1_itemsets();    // Find frequent 1- itemsets
    PrintTable("L" + k++, L);
    while(L.size() >= 1)
    {
        C = apriori_gen(L,k);  // Generate new k-itemsets candidates
        C=CandidateSupportCount(C,k); // Find the support of all the candidates
        if(C.size()<=0)
            break;
        PrintTable("C" + k, C);
        L=FindLargeItemsets(C); // Take only those with support over minsup
        if(L.size()<=0)
            break;
        PrintTable("L" + k, L);
        k++;
    }
}
```
//Candidate Generation:
apriori_gen(L, k)
{
    LinkedList<FrequentItem> C = new LinkedList<FrequentItem>();
    for(int i=0; i<L.size(); i++)
    {
        for(int j=i+1; j<L.size(); j++)
        {
            if(IsPossibleInter(L.get(i).itemset, L.get(j).itemset, k-1))
            {
                int temp = L.get(j).itemset.lastIndexOf("",)+1;
                FrequentItem c = new FrequentItem(L.get(i).itemset+"," + L.get(j).itemset.substring(temp), 0);
                if(!Has_Infrequent_Subset(c, L, k))
                    C.addLast(c);
            }
        }
    }
    return C;
}

Limitations:

Apriori, while historically significant, suffers from a number of inefficiencies or trade-offs, which have produced other algorithms. Apriori faces some challenges like candidate generation, i.e. generates large numbers of subsets (the algorithm attempts to load up the candidate set with as many as possible before each scan), multiple scans of transaction database and heavy job of support calculation for candidates[18, 26, 29].
Improvements:

Also Apriori algorithm can be modified to improve its efficiency (computational complexity) by hashing, removal of transactions that do not contain frequent itemsets, sampling of the data, partitioning of the data, and mining frequent itemsets without generation of candidate itemsets.

Sampling is the simplest example, where accuracy is lost in favor of performance gain\textsuperscript{[4, 13]}. Hash-based technique\textsuperscript{[36]}, Transaction reduction, Partitioning, Dynamic itemset counting\textsuperscript{[36]}, and multilevel and multidimensional association rules are some of the other common enhancements proposed to improve the efficiency of Apriori algorithm\textsuperscript{[1, 3]}. 
7.2 DHP – Direct Hashing and Pruning

Improvement into Apriori has been anticipated by various algorithms which holds efficiency enhancement\cite{3}. Hash-based technique is useful to compact the quantity of the candidates for each k-level. Database scanning for every transaction is done for candidate generation\cite{1, 3}. Like for $L_1$ we find $C_1$, same for $L_2$, find $C_2$. Hashing can be functioned as\cite{1, 3}:

a. “$H(x,y)=((\text{order of } x) \times 10 + \text{order of } y) \mod 7$”

b. “A 2-itemset whose corresponding bucket count in the hash table is below the threshold cannot be frequent and thus should be removed from the candidate set”.

DHP makes lesser size $C_k$ than Apriori findings. Consequently, it is quicker in performing $C_k$ from DB to establish $L_k$. The size of $L_k$ shrinks hastily as $k$ enlarges\cite{3, 39}.

DHP can be used for proficient large itemset generation. It has two major features: efficient candidate generation for large itemsets and effective reduction on transaction database\cite{3, 39}. It uses a hashing technique. In particular, for the large 2-itemsets where the number of candidate itemsets generated by DHP, is smaller in size compare to Apriori method\cite{3, 18}. Thus, it improves the performance bottleneck of the whole process\cite{19, 39}. It uses a pruning technique to reduce the size of the database progressively\cite{6}.

How does it look like?

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{apriori_dhp.png}
\caption{Comparison of Apriori and DHP\cite{1}}
\end{figure}
Hash Table Construction:

We are having Items like “A, B, C, D, E”, Sequence for these items will be “1, 2, 3, 4, 5” respectively, then “H({C, E})= (3*10 + 5)% 7 = 0”. Therefore, {C, E} fits into bucket 0.

![Figure 7.7.1 DHP Example (Part 1)](image-url)
**7. Review of Literature**

### Generating $C_2$

<table>
<thead>
<tr>
<th>Itemset</th>
<th># in bucket with itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>${A, B}$</td>
<td>1</td>
</tr>
<tr>
<td>${A, C}$</td>
<td>3</td>
</tr>
<tr>
<td>${A, E}$</td>
<td>1</td>
</tr>
<tr>
<td>${B, C}$</td>
<td>2</td>
</tr>
<tr>
<td>${B, E}$</td>
<td>3</td>
</tr>
<tr>
<td>${C, E}$</td>
<td>3</td>
</tr>
</tbody>
</table>

### Database TDB

<table>
<thead>
<tr>
<th>Tid</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>A, C, D</td>
</tr>
<tr>
<td>20</td>
<td>B, C, E</td>
</tr>
<tr>
<td>30</td>
<td>A, B, C, E</td>
</tr>
<tr>
<td>40</td>
<td>B, E</td>
</tr>
</tbody>
</table>

### Counting Support in a hash tree

- **Part 2:**

  Generates the set of candidate itemsets $C_2$ based on the hash table $(H_2)$, determines the set of large 2-itemsets $L_2$. Also reduces the size of database for the next large itemsets & makes $h_3$ for next $C_3$ candidate large itemsets.

### Database TDB

<table>
<thead>
<tr>
<th>Tid</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>B, C, E</td>
</tr>
<tr>
<td>30</td>
<td>B, C, E</td>
</tr>
</tbody>
</table>

### $C_2$

<table>
<thead>
<tr>
<th>Itemset</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>${A, C}$</td>
<td>2</td>
</tr>
<tr>
<td>${B, C}$</td>
<td>2</td>
</tr>
<tr>
<td>${B, E}$</td>
<td>3</td>
</tr>
<tr>
<td>${C, E}$</td>
<td>2</td>
</tr>
</tbody>
</table>

### $L_2$

<table>
<thead>
<tr>
<th>Itemset</th>
<th>sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>${A, C}$</td>
<td>2</td>
</tr>
<tr>
<td>${B, C}$</td>
<td>2</td>
</tr>
<tr>
<td>${B, E}$</td>
<td>3</td>
</tr>
<tr>
<td>${C, E}$</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 7.7.2 DHP Example (Part 2)**
Candidate Itemsets Trimming Process:

Bucket value is compared with support, if it is larger, then this will be forwarded to the next k+1 level process.

(DHP Example)

(1) In transaction (A, C, D), a single candidate AC is found in \( C_2 \). Values are: \( a[0] = 1, a[1] = 1, a[2] = 0 \). Due to the lesser value of each candidate than \( k=2 \), these itemsets will be discarded. (II) In transaction (A, B, C, E), has four candidate 2-items (AC, BC, BE, CE) found in \( C_2 \). Values are: \( a[0] = 1, a[1] = 2, a[2] = 2, a[3] = 2 \). Since the value of \( a[0] \) is less than \( k \) (i.e. now \( k=2 \)), and the remaining are \( \geq k \), this transaction will be reduced to (B, C, E) and A is thus discarded.

DHP uses a hashing technique due to which, it is more powerful than Apriori i.e. candidate 2-itemsets are huge for Apriori, while DHP trims them using hashing. For Apriori, transaction database is huge, so that one scan per iteration is costly. While, DHP is less costly as it prunes both number of transactions and number of items in each transaction after each iteration\(^{[6,39]}\).
Figure 7.9 DHP Algorithm\cite{3}

Part : 1 :

Gets a set of large 1-itemsets and makes a hash table (i.e. $H_2$) for 2-itemset:

\[
\begin{align*}
\text{/* Part 1 */} \\
s &= \text{a minimum support; } \\
\text{set all the buckets of } H_2 \text{ to zero; } /* \text{hash table */} \\
\text{forall transaction } t \in D \text{ do begin } \\
\text{insert and count 1-items occurrences in a hash tree; } \\
\text{forall 2-subsets } x \text{ of } t \text{ do } \\
H_2[h_2(x)] &\Rightarrow +; \phantom{000} \\
\text{end} \\
L_1 &= \{c|c.\text{count} \geq s, c \text{ is in a leaf node of the hash tree}\}; \\
\end{align*}
\]

Part : 2 :

Generates the set of candidate itemsets $C_2$ based on the hash table ($H_2$), determines the set of large 2-itemsets $L_2$. Also reduces the size of database for the next large itemsets & makes $h_3$ for next $C_3$ candidate large itemsets

\[
\begin{align*}
\text{/* Part 2 */} \\
k &= 2; \\
D_k &= D; \quad /* \text{database for large k-itemsets */} \\
\text{while } \{|\{x|H_k[x] \geq s\}| \geq \text{LARGE} \} \{ \\
\text{/* make a hash table */} \\
\text{gen_candidate}(L_{k-1}, H_k, C_k); \\
\text{set all the buckets of } H_{k+1} \text{ to zero;} \\
D_{k+1} &= \phi; \\
\text{forall transactions } t \in D_k \text{ do begin } \\
\text{count_support}(t, C_k, k); \quad /* t \subseteq t */ \\
\text{if } (|t| > k) \text{ then do begin } \\
\text{make_hash}(\hat{t}, H_k, k, H_{k+1}, \hat{t}); \\
\text{if } (|t| > k) \text{ then } D_{k+1} = D_{k+1} \cup \{\hat{t}\}; \\
\text{end} \\
L_k &= \{c \in C_k|c.\text{count} \geq s\}; \\
k &= k+1; \\
\} \\
\end{align*}
\]
Procedure count_support\( t, C_k, k, i \) 
\( \text{/* explained in Section 3.2 */} \)
forall \( c \) such that \( c \in C_k \) and \( c = t_{i_1} \cdots t_{i_k} \in t \) do
begin
\( c.\text{count} + +; \)
for \( (j = 1; j \leq k; j++) \) \( a[i_j] + +; \)
end
for \( (i = 0; j = 0; i < \mid t \mid; i++) \)
if \( a[i] \geq k \) then do begin \( t_j = t_i; j++; \) end
end Procedure

Procedure gen_candidate\( L_{k-1}, H_k, C_k \)
\( C_k = \emptyset; \)
forall \( c = c_p[1] \cdots c_p[k - 2] \cdot c_p[k - 1] \cdot c_q[k - 1], \)
\( c_p, c_q \in L_{k-1}, \mid c_p \cap c_q \mid = k - 2 \) do
if \( H_k[h_k(c)] \geq s \) then
\( C_k = C_k \cup \{c\}; \) /* insert \( c \) into a hash tree */
end Procedure

Procedure make_hasht\( \hat{t}, H_k, k, H_{k+1}, \hat{t} \)
forall \((k + 1)\)-subsets \( x = t_{i_1} \cdots t_{i_k} \) of \( \hat{t} \) do
if (for all \( k\)-subsets \( y \) of \( x \), \( H_k[h_k(y)] \geq s \)) then do begin
\( H_{k+1}[h_{k+1}(x)] + +; \)
for \( (j = 1; j \leq k + 1; j++) \) \( a[i_j] + +; \)
end
for \( (i = 0; j = 0; i < \mid \hat{t} \mid; i++) \)
if \( a[i] > 0 \) then do begin \( \hat{t}_j = \hat{t}_i; j++; \) end
end Procedure
Part 3:
Further process same as Apriori method. But it provides database reduction in each pass.

```c
/* Part 3 */
gen_candidate(L_{k-1}, H_{k}, C_{k});
while (|C_{k}| > 0) {
    D_{k+1} = \phi;
    forall transactions t \in D_{k} do begin
        count_support(t, C_{k}, k, \hat{t}); /* \hat{t} \subseteq t */
        if (|\hat{t}| > k) then D_{k+1} = D_{k+1} \cup \{\hat{t}\};
    end
    L_{k} = \{c \in C_{k} | c.count \geq s\};
    if (|D_{k+1}| = 0) then break;
    C_{k+1} = apriori_gen(L_{k}); /* refer to [5] */
    k++;}
```
### Apriori Result

Table Name: transactions1

<table>
<thead>
<tr>
<th>Transaction Table</th>
<th>Analysis</th>
<th>No. of Rows in Database Table (After Ck)</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C, D</td>
<td>4 - transactions considered for study</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>A, B, C, E</td>
<td>5 - itemsets for C1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>B, C, E</td>
<td>4 - itemsets selected for L1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>A, B, C, E</td>
<td>6 - itemsets for C2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>A, C, E</td>
<td>4 - itemsets selected for L2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>B, C, E</td>
<td>1 - itemset for C3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>B, C, E</td>
<td>1 - itemset selected for L3</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 7.10 Apriori Result for Candidate Itemset Generation
### DHP Result

**Table Name:** transactions1  

<table>
<thead>
<tr>
<th>Transaction Table</th>
<th>Analysis</th>
<th>No. of Rows in Database Table (After Ck)</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, C, D</td>
<td>4 - transactions considered for study</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>B, C, E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, B, C, E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, C</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>B, C</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, E</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C, E</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>L2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A, C</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>B, C</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, E</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C, E</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C3</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B, C, E</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td><strong>L3</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.11 DHP Result for Candidate Itemset Generation**
By following Fig. 7.10 and 7.11 for candidate itemset $C_2$, Apriori generates 6 itemsets while DHP generates only 4 itemsets which is lesser than Apriori and proves earlier discussion.

As described in Fig. 7.6, Apriori behaves in sequence like - generate candidate set and perform support count from the database. While DHP behaves in a sequence like – generate candidate sets, perform support count from the database and make a new hash table using database for the next stage.

It is difficult to handle hash table and candidate set as the level increases along with bucket size grown up. This makes DHP more difficult to manage[39].
7.3 ECLAT

Equivalence CLASS Transformation (ECLAT) is used to mine frequent patterns.

Eclat algorithm finds the elements from the bottom like depth first search\textsuperscript{[28, 36]}. Eclat algorithm is very simple algorithm to find the frequent itemsets. This algorithm uses vertical database rather than horizontal database\textsuperscript{[34]}. If there is any horizontal database, then we need to convert it into vertical database. There is no need to scan the database repetitively. Eclat algorithm scans the database only once. Support is counted in this algorithm while confidence is not calculated in this algorithm\textsuperscript{[36]}.

ECLAT does set intersections to generate frequent patterns. Apriori as well as FP-growth techniques, generate frequent itemsets from a group of entry sets in horizontal data layout (i.e.,\{TID: pattern\}), where TID is an entry-id and pattern is the group of objects bought in entry TID\textsuperscript{[5]}. Instead, mining can also be carried out with data offered in vertical data layout (i.e., \{Object: TID\_entryset\}). Eclat technique is projected by discovering the vertical data layout\textsuperscript{[5, 28]}. First scanning of DB constructs the unique TID\_entryset of the individual item appeared in the transaction. Starting with a one item (k = 1), the frequent (k +1)-patterns developed from a preceding k-pattern, can be produced as per Apriori property and with a depth-first calculation arrangement related to FP-growth\textsuperscript{[4, 5]}. The calculation is completed by the junction of the TID\_entrysets of frequent k-patterns to calculate the TID\_entrysets of the consequent (k+1)-patterns. This procedure continues, until no frequent patterns or no candidate patterns can be generated\textsuperscript{[5]}.

Attributes of ECLAT are\textsuperscript{[5]}: locality enhancing approach, easy and efficient to parallelize, a few scans of the database (best case 2).
Figure 7.12 ECLAT Design Process

In Fig. 7.12, design view of ECLAT shows that it employs depth-first search approach and it does set intersections to generate frequent patterns. Some properties related to ECLAT are:

1. Lexicographic order is followed to check the generated itemsets.
2. Search format is followed with prefix property and generating canonical extensions.
3. Eclat generates more candidate itemsets than Apriori, because it (usually) does not store the support of all visited itemsets. As a consequence it cannot fully exploit the Apriori property for pruning[49].
4. Individual test for each subset is not performed in its creation as well as to do support count. Rather support count is performed with intersection results generated from transactions.
Input: $D, \sigma, I \in T$
Output: $F[I](D, \sigma)$

-----------------------------

$F[I] := \emptyset$

for all $i \in T$ occurring in $D$ do

$F[I] := F[I] \cup \{I \cup \{i\}\}$

// Create $D^i$

$D^i := \emptyset$

for all $j \in T$ occurring in $D$ such that $j > i$ do

$C := \text{cover} \{\{i\}\} \cap \text{cover} \{\{j\}\}$

if $|C| \geq \sigma$ then

$D^i = D^i \cup \{j, C\}$

end if

end for

// Depth-first recursion

Compute $F[I \cup \{i\}](D^i, \sigma)$

$F[I] := F[I] \cup F[I \cup \{i\}]$

end for

Figure 7.13 ECLAT Algorithm\textsuperscript{[5, 40]}
Database

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, b, c, d</td>
</tr>
<tr>
<td>2</td>
<td>a, b, c</td>
</tr>
<tr>
<td>3</td>
<td>a, b, d, e</td>
</tr>
<tr>
<td>4</td>
<td>c, e</td>
</tr>
<tr>
<td>5</td>
<td>b, d, e</td>
</tr>
<tr>
<td>6</td>
<td>a, b, e</td>
</tr>
<tr>
<td>7</td>
<td>a, c, e</td>
</tr>
<tr>
<td>8</td>
<td>a, d, e</td>
</tr>
<tr>
<td>9</td>
<td>b, c, e</td>
</tr>
<tr>
<td>10</td>
<td>b, d, e</td>
</tr>
</tbody>
</table>

min_sup=2

**Figure 7.14 ECLAT Algorithm Example**
Figure 7.15 ECLAT Algorithm Example with Final Result

Implementation:

Implementation of this technique uses groups of transactions by the bit matrix. It intersects rows to find support and candidate itemsets.

Applications:

Eclat is also used in the mapreduce framework. Also, it is used in network
service. Eclat algorithm helps to find which product is purchased frequently by customer in their online stores. We know that people who live in the same city make the same demand in the market. So by Eclat algorithm, shop keeper can know the interest of people\textsuperscript{[19]}.

Limitations:

TID\_sets can be quite long, taking substantial memory space as well as computation time for intersecting the large sets\textsuperscript{[19]} but this method approaches very fast support counting.
7.4 FP-Growth

FP-growth creates frequent patterns without candidate generation\cite{4}. The shortfalls of Apriori, DHP and ECLAT are considered and FP-growth had tried to improve on these techniques\cite{19}. This is done with the use of FP-tree. FP-tree is a compressed and compact version of DB\cite{4}. FP-growth avoids costly repeated database scans. FP-growth follows depth-first search approach\cite{1, 4, 19}. FP-tree also uses conditional tree meanwhile by satisfying min\_sup criteria\cite{1, 4, 19}.

FP-growth taken-place with a divide-and-conquer method\cite{4, 28, 29}. The first check of the DB originates a listing of frequent patterns. This listing of patterns is prearranged with occurrence downward sequence\cite{4, 40}. As per occurrence-downward listing, DB is compacted into frequent-itemsets hierarchy, i.e. FP-tree. FP-tree also holds the pattern relationship information. FP-tree is mined by beginning from every frequent size-1 itemset (like a preliminary suffix pattern)\cite{4}. Then building its conditional itemset base (“substitute DB”, that holds of group of prefix path in FP-tree found together with the suffix pattern)\cite{4, 29}. Last building their conditional FP-tree, as well as performing mining iteratively in recursion mode with conditional FP-tree. Itemsets development can be accomplished with the joining of suffix itemset with frequent itemsets created from conditional FP-tree\cite{4, 28}. The FP-growth technique uses a recursive approach to discover long pattern by following smaller patterns. FP-growth considerably diminishes multiple database scanning costs.

Apriori algorithm generates candidate sets and tests them to discover recurrent itemsets, significantly reducing the amount of candidate sets. While FP-growth mine recurrent itemsets without candidate generation. Both algorithms have their own advantages and disadvantages. Many hybrid algorithms have been proposed and still researched to suit the general case, or mostly a particular case specialized for a given dataset. “Both the Apriori and FP-growth methods mine frequent patterns from a set of transactions in TID- itemset format (that is, \{TID : itemset\}), where TID is a transaction-id and itemset is the set of items bought in transaction TID".
Construct FP-tree:

This task follows two steps\textsuperscript{[29, 40]}:

I. First time database scan is followed to find frequent 1-itemsets. Arrange them in descending order by support count into a list L.

\[ L = \{f:4, c:4, a:3, b:3, m:3, p:3\} \]

Consider the format of (item-name: support).

II. Utilize descending sequence in L and arrange each transaction’s items by following L. Perform another database scan and build FP-tree.
Input: A database DB, represented by FP-tree.

Output: The complete set of frequent patterns.

Method: call FP-growth(FP-tree, null).

Procedure FP-growth(Tree, α)

1. if Tree contains a single prefix path  // Mining single prefix-path FP-tree
2. then {
3. let P be the single prefix-path part of Tree;
4. let Q be the multipath part with the top branching node replaced by a null root;
5. for each combination (denoted as β) of the nodes in the path P do
6. generate pattern β ∪ α with support = minimum support of nodes in β;
7. let freq_pattern_set(P) be the set of patterns so generated;  }
8. else let Q be Tree;
9. for each item αi in Q do { warmed multi-path FP-tree
10. generate pattern β = αi ∪ α with support = αi.support;
11. construct β’s conditional pattern-base and then β’s conditional FP-tree Treeβ;
12. if Treeβ ≠ ∅
13. then call FP-growth(Treeβ, β);
14. let freq_pattern_set(Q) be the set of patterns so generated;  }
15. return(freq_pattern_set(P) ∪ freq_pattern_set(Q) ∪ (freq_pattern_set(P)
× freq_pattern_set(Q)))
}

Figure 7.16 FP-Growth Algorithm\textsuperscript{[4]}

FP-Tree Example\textsuperscript{[4]}:

Step 1: Scan DB for the first time to generate L\textsuperscript{[40]}.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>{f, a, c, d, g, i, m, p}</td>
</tr>
<tr>
<td>200</td>
<td>{a, b, c, f, l, m, o}</td>
</tr>
<tr>
<td>300</td>
<td>{b, f, h, j, o}</td>
</tr>
<tr>
<td>400</td>
<td>{b, c, k, s, p}</td>
</tr>
<tr>
<td>500</td>
<td>{a, f, c, e, l, p, m, n}</td>
</tr>
</tbody>
</table>

L

<table>
<thead>
<tr>
<th>Item</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>m</td>
<td>3</td>
</tr>
<tr>
<td>p</td>
<td>3</td>
</tr>
</tbody>
</table>

By-Product of First Scan of Database

Figure 7.17 First Time DB Scanning for FP-Tree Generation
Step 2: Scan DB for another time, order frequent items in every transaction and construct a FP-Tree\(^{[40]}\).

<table>
<thead>
<tr>
<th>TID</th>
<th>Items bought</th>
<th>(ordered) frequent items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>{f, a, c, d, g, i, m, p}</td>
<td>{f, c, a, m, p}</td>
</tr>
<tr>
<td>200</td>
<td>{a, b, c, f, l, m, o}</td>
<td>{f, c, a, b, m}</td>
</tr>
<tr>
<td>300</td>
<td>{b, f, h, j, o}</td>
<td>{f, b}</td>
</tr>
<tr>
<td>400</td>
<td>{b, c, k, s, p}</td>
<td>{c, b, p}</td>
</tr>
<tr>
<td>500</td>
<td>{a, f, c, e, l, p, m, n}</td>
<td>{f, c, a, m, p}</td>
</tr>
</tbody>
</table>

Figure 7.18 Second Time DB Scanning for FP-Tree Generation

Figure 7.19 FP-Tree Construction
Formally, FP-tree is a tree structure defined as below:\(^4\), \(^40\):

1. “One root labeled as ‘null’, a set of item prefix sub-trees as the children of the root, and a frequent-item header table”.

2. Each node in the item prefix sub-trees, has three fields:
   
   I. Item-name: register which item this node represents,
   
   II. Count, the number of transactions represented by the portion of the path reaching this node,
   
   III. Node-link that links to the next node in the FP-tree carrying the same item-name, or null if there is none\(^{34}\).

3. Each entry in the frequent-item header table has two fields,
   
   I. Item-name, and
   
   II. Head of node-link that points to the first node in the FP-tree carrying the item-name.
**FP-Tree Properties**\(^4\):

1. Node-link property: “For any frequent item \(a_i\), all the possible frequent patterns that contain \(a_i\) can be obtained by following \(a_i\)'s node-links, starting from \(a_i\)'s head in the FP-tree header.”

2. Prefix path property: “To calculate the frequent patterns for a node \(a_i\) in a path \(P\), only the prefix sub-path of \(a_i\) in \(P\) needs to be accumulated, and its frequency count should carry the same count as node \(a_i\).”

**Advantages of the FP-Tree Structure:**

1. FP-Tree scans the database two times only\(^{29}\).

2. Completeness: FP-tree includes information like item-name, count, and node-link, which helps to perform FPM by referring min_sup count\(^{29}\).

3. Compactness: Frequent itemsets decides the size of the structured tree and height of it depends on the possible count of the total items in the transaction\(^{29}\).

**Justification for Descending Order Items Arrangement**\(^4\):

![Figure 7.21 Unordered Frequent Items](image)

**Figure 7.21 Unordered Frequent Items**
Figure 7.22 Ascending Ordered Frequent Items

Mining Frequent Patterns Using FP-Tree\[^4\]:

There are three major steps for frequent pattern mining from FP-Tree. Mining process begins with the processing of list L from the end.

Step 1: Build conditional pattern base\[^{40}\].

1. This begins with header table bottom node.
2. Next, perform traversal into FP-Tree by following node link.
3. Then building each item’s conditional itemset base (“substitute DB”, that holds of group of prefix path in FP-tree found together with the suffix pattern) from transformed prefix paths.

Figure 7.23 Conditional Pattern Base (CPB) Construction
Step 2: Build conditional FP-Tree\[^{[40]}\].

For each CP base,

1. First, perform accumulation of every item count in the base.
2. Next, conditional FP-Tree construction is performed for each frequent item in the pattern base.

![Figure 7.24 Conditional FP-Tree(CFP-Tree) Construction](image1)

Step 3: CFP-Tree mining being done recursively

![Figure 7.25 Mining of Conditional FP-Tree Recursively\[^{[40]}\]](image2)
Pattern growth property: “Let $\alpha$ be a frequent itemset in DB, $B$ be $\alpha$’s conditional pattern base, and $\beta$ be an itemset in $B$. Then $\alpha \cup \beta$ is a frequent itemset in DB if $\beta$ is frequent in $B$.”

Let’s think about “fcabm”. Is it frequent pattern?

1. m's conditional pattern base is concerned for “fcab”.
2. For “fcab”, b is not frequent. Also, “bm” is not frequent pattern.

So, the answer is, “fcabm” is not frequent itemset.

<table>
<thead>
<tr>
<th>Item</th>
<th>Conditional pattern base</th>
<th>Conditional FP-tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>{{fcam:2}, (cb:1)}</td>
<td>{{c:3}}</td>
</tr>
<tr>
<td>m</td>
<td>{{fca:2}, (fcab:1)}</td>
<td>{{f:3, c:3, a:3}}</td>
</tr>
<tr>
<td>b</td>
<td>{{fca:1}, (f:1), (c:1)}</td>
<td>Empty</td>
</tr>
<tr>
<td>a</td>
<td>{{fc:3}}</td>
<td>{{f:3, c:3}}</td>
</tr>
<tr>
<td>c</td>
<td>{{f:3}}</td>
<td>{{f:3}}</td>
</tr>
<tr>
<td>f</td>
<td>Empty</td>
<td>Empty</td>
</tr>
</tbody>
</table>

Figure 7.26 Conditional Pattern Bases and Conditional FP-Tree

Suppose an FP-tree $T$ has a single path $P$. The complete set of frequent pattern of $T$ can be generated by enumeration of all the combinations of the sub-paths of $P$.

Figure 7.27 Single FP-Tree Path Generation
FP-Growth suffers with convincing shortcomings. Like, FP-tree requires higher memory resident which is a shortfall in main memory\textsuperscript{[37]}. Diverse storage configuration makes much higher execution time for this method\textsuperscript{[6, 37]}.

**Other Algorithms:**

Dong and Han\textsuperscript{(2007)} presented a BitTableFi algorithm\textsuperscript{[10]}. In this algorithm, data structure BitTable is used horizontally and vertically to compress database for quick candidate itemsets generation and support counting, respectively. As reported by Dong and Han, BitTableFi outperforms other algorithms like CBAR and Apriori\textsuperscript{[10]}.

In Index-BitTableFi algorithm proposed by W.Song, B.R.Yang and Z.Y.Xu\textsuperscript{(2008)}, index array is used to reduce the redundant operations on intersection of tidsets and frequency checking significantly\textsuperscript{[17]}. In this algorithm, frequent itemsets can be directly identified by connecting the representative item with all the combinations of items in its index array values.