CHAPTER 5

PROPOSED REFINED SEARCH DIVIDE AND CONQUER
(HYBRID) ALGORITHM

5.1 INTRODUCTION

This chapter mainly deals with a new approach towards developing an algorithm to overcome the snags present in the existing algorithms prevailing across the sphere. A set of candidate patterns of length \((k+1)\) is created by Apriori from the set of frequent patterns of length \(k\) (for \(k \geq 1\)), and this in turn is checked their matching occurrence in the database. The Apriori algorithm realizes an exceptional performance gain by decreasing the size of candidate sets created. Apriori-like algorithm may still suffer from the nontrivial costs in patterns like productive frequent patterns, long patterns, or very low minimum support thresholds. It is very difficult and too expensive to handle a huge number of candidate sets as the cost incurred will be high. For example, in Apriori algorithm it is imperative to produce more candidates list and test their occurrence frequencies which cover the approach for the intrinsic cost of candidate generation and immaterial of what implementation technique is applied. Repeatedly scanning the database and check a large set of candidates for pattern matching is tedious one and it is especially important for mining long patterns.
The important task needed is to develop efficient frequent mining techniques, as frequent pattern mining is an essential and important data-mining task. Apriori algorithm has its own pros and cons. Avoiding the costly candidate-generation-and-test and repeated database scan operations; substantially improve the effectiveness of frequent pattern mining. Frequent pattern mining suffers from the lack of efficiency and the lack of effectiveness, i.e., there could be a huge number of frequent patterns generated from a database. This chapter tries to make good analysis in answering the above questions.

A. PRELUDE

Data mining is a term associated with extracting data from large data warehouses. Association rule of mining finds interesting associations or correlation among large sets of data.

Decision support System/Executive Information System (DSS/EIS) architectures have quickly evolved from the first generation of mainframe-based, proprietary application development environments to the second generation of open, distributed architectures, integrating client resident query tools with server-based relational databases.

These second-generation products, while providing more rapid development and improved Graphical User Interfaces (GUIs), are too often simply data access tools, which provide rudimentary data access and report presentation. These tools lack key features needed by both the decision support analyst and the Information System (IS) developer.
The data warehouse is the foundation of any DSS data delivery and performance, Decision Support queries, due to their broad scope and analytical intensity, typically require data models to be optimized to improved query performance. In addition to impacting query performance, the data model affects data storage requirements and data loading performance.

In real time applications regarding the aforesaid rule, the earlier researchers have developed a plethora of algorithms and methodologies. It is imperative to find a new advanced algorithm, which satisfies maximum categories of users irrespective of their fields, area of specialization and other demographics.

i. **Oracle structure**

To find a solution to the existing deficiencies, this research work would pave the way for a new advanced algorithm, by analyzing various data mining algorithms. The lack of accuracy in partitioning the databases data structures for finding larger item sets, factorization of data and data look up is the root cause of the deficiencies which are to be improved using current oracle based association rule.

In the proposed algorithm, similar data structures are mixed as a group and stored in the single database to optimize the queries and to reduce the probability of searching. The significant concept of the algorithm is to search the data dictionary and locate the data sets, and group them to minimize the area of searching and optimize memory performance. For
instance, information can be retrieved from different databases, and grouped in to a single database, so that the newly created database can be used to tap the complete information in multiple databases. The proposed algorithm is to be developed using the tablespaces which gives us the total database factions and similarly Data files provides the total number physical files used by the data warehouse in terms of size and usage.

**ii. Parallel algorithms:**

The currently available algorithms are all based on partitioning the databases and counting the longer item sets in large databases. The newly proposed algorithm exhibits a single rapids search of large volume of databases. This algorithm attempt to solve the existing deficiencies in the data mining algorithms

**iii. Automatic Query Generator:**

Of all of the tasks which the DSS engine performs, translating the analysts request in to SQL queries, the engine queries inputs from the end-user application and the metadata, and then builds the SQL query with algorithms within the DSS engine.

**iv. SQL generation Algorithm:**

Once the DSS engine has received the end-user information request, it applies the Data Rotation and criteria sets to the Meta data to build the SELECT and WHERE clauses of the SQL query. The DSS engine first determines how the criteria sets should be resolved to form the foundation of
the SQL WHERE clause. The DSS engine then determines the SQL SELECT clause using the Data Rotation.

To build the FROM clause, the engine determines which tables must be accessed to retrieve the requested data by referencing the metadata. The engine must also determine which tables in the warehouse provide optimum query performance and create inter-table joins, as required.

The above SQL generation must occur dynamically at runtime, whenever the analyst creates an analysis by combining a criteria set, a data rotation and a presentation. To summarize the DSS engine must synthesize metadata and specifications from the DSS application to produce SQL queries.

v. Mathematical manipulation:

The DSS engine provides the capability to perform complex mathematical manipulations on the data returned from the SQL query. These include the ability to apply various arithmetic expressions to the data, and the ability to perform standard statistical analyses on the data before presenting the data to the end-user.

vi. Optimizing the data Warehouse for Decision Support:

The data warehouse is the foundation of any data delivery system, and the warehouse data model must address the conflicting goals of system flexibility and data delivery performance. Flexibility is required so that additional attributes, dimensions and metrics can be added to the warehouse, complementing and leveraging existing data. Quick query response is required in
all effective interactive Decision Support Systems, since it not only impacts end-user satisfaction but directly determines DSS analysts can perform given limited time.

In considering the data warehouse data model, it is useful to first review the types of tables found in a data warehouse.

The three types are:

- primary data tables
- Descriptor Tables
- Characteristics Tables

Primary Data Tables contain both metrics and attribute.

vii. Metrics:

Metrics are variables or measures typically numeric in nature, which are the focus of the decision support investigation. Examples of metrics include sales, revenues, Budgets, Inventory, and Market Share.

viii. Dimensions:

When considering a metric such as sales, it is important to consider what data is available. Does sales information exist for each of products? Does sales data exist for each of the countries, regions and states? Does sales data exist for the last five years? Metrics such as sales do not exist in isolation but rather in the context of dimensions such as product geography, and time which defines what type of sales data is available. It is natural to think of data multi-dimensionally.
ix. Attributes:

Attributes are specific metric qualifiers and are defined by columns in the data warehouse. Attributes belong to dimensions. Dimensions are conceptual metric qualifiers and are not physically represented in the data warehouse. An attribute always associated with a dimension.

If we consider the above sales example in more detail, we realize that the dimensions of product, Geography, and time only generally describe the available sales data. To perform an analysis, it may need sales data by region, state, and city, each of which is an attribute of the geography dimension. Also it may require sales data by Month, Week, and Day, each of which is an attribute of the Time dimension.

x. Performance Optimization:

DSS query performance, while a function of the performance of every component in the data delivery architecture, is strongly correlated to the physical data model. Intelligent data modeling through the use of techniques such as de-normalization, consolidation and partitioning can provide orders of magnitude performance gains compared to the use of normalized data.

xi. Oracle boundaries:

Oracle allows addition of new physical files called data files to any existing database of unlimited size. The number of physical files can be used and added up to a maximum of 1088 files. The proposed research identifies the number of physical files affecting performance of the system.
xii. **Cases:** Let us consider vast storage of information. The data, which has numerous of information and is transacted heavily like credit cards/loan application. When an application is processed the information captured is normally huge. The following list gives you an idea of the record size and its storage issues.

**Problem description:**

On an average the records size 1000 bytes or more. A table that holds this data will need more storage space. Example: - if there are 1 lakh customers we need to store

\[ \frac{100000 \text{ kb}}{1024} \text{ MB} \text{ or 100 MB of space.} \]

On an average we would have above 10 lakh customer applications, increasing it 10 times 1000mb (1 GB) for each city. As the number of customers grows the database growth is more pronounced as the number of cities increase. Actual facts show space requirements in Terabytes (1024 GB) for such databases.

- Only to hold the information. The problem is more pronounced as the number of customers grows.
- Assuming there are 10lakh customers. The hit rate for successful account holders is 50% of 20lakh applications that would have to be stored approximately.20lakh/1000 GB of space would be required. With decreasing hi rate the problem of storage also increases.
Analysis:

The average time taken to retrieve a non-indexed database tables is unimaginable.

The index takes 100 to 125% of the database size is effectively doubling the table/database size. The information retrieval again is grossly direct proportional to database size.

Estimation- Data files query

Select count (*) from v_$datafiles

Select count (*) from DBA_tablespaces

The Number of datafiles for a table space once the users tablespace is identified (other than, index, temporary, system, Rollback) the number of datafiles corresponding to the tablespaces can be identified.

Once the datafiles are identified the query will give the largest tables and size and the number of datafiles it is split into execution plan identify primary key, the record length, no of rows, no of data files, then it can be portioned based on the number of values
Select the number of data files based on row id

Query – to indicate fragmentation or partition

Step 2 – Decide on partition/fragmentation execution plan for indexes

*  Pick a query for V$sql_Area to find out the most frequently and query work on its cost
*  query partition the tables and rewrite procedure to trace partition ranged maintain one table to access partition rows and values minimizing the search time and access directly by row id, index an rowed only for the target tables.

5.2 CONTRIBUTION

This chapter describes the following contributions. This work develops an association rule based pattern mining. A new algorithm RSDCA is proposed for efficiently mining frequent patterns from voluminous datasets. Constraint-based data mining is an important approach to solve the problem of data mining. This research finds the problem of constraint-based fast mining using pattern-growth methods and Divide and Conquer method. This research shows the pattern growth methods and drives the constraints into the mining process towards “Divide and conquers method” and extracts the frequent patterns quickly and easily. This work extends the fast pattern growth method to allow the mining of sequential patterns. This approach shows that fast pattern mining methods are more efficient in mining voluminous sequence
databases and new techniques are developed to solve the sequential fast search mining problem effectively.

5.3 PROBLEM DESCRIPTION

This chapter uses the standard definition of association rules from sections 1.5 and 2.2. In this chapter, the Association Rule Mining problem is decomposed into two phases.

**Phase 1:** Find all frequent itemsets \( X \) that have support above the user-specified minimum support \( \alpha \).

**Phase 2:** For each frequent itemset found in **Step 1**, derive all rules that have more than user-specified minimum confidence \( \beta \) and minimum lift \( \gamma \) as follows: for itemset \( x \) and \( y \) \( \text{Conf}(x \Rightarrow y) \geq \text{MinConf} \), then association rule \( x \Rightarrow y \) is derived.

An association rule \( x \Rightarrow y \) such that \( \text{Supp}(X) \geq \alpha \) and \( \text{Conf}(x \Rightarrow y) \geq \beta \) is called strong valid and interesting association rule. The support and confidence measures are discussed at section 2.6.

5.4 RELATED WORKS

5.4.1 AprioriTID

The AprioriTID [81] is another variant of Apriori which reduces the time needed for the frequency counting procedure by replacing every transaction in the database by set of candidate sets that occurs in that
transaction. This is done repeatedly at every iteration $k$. The adapted transaction database is denoted by $C_k$. AprioriTID algorithm is a very quicker in next iterations part; it carry out slower than Apriori in the previous iterations. It is extra overhead that is created when $C_k$ does not fit into the memory and has to be written to disk [9] [10]. If consider a specific transaction does not containing any candidate $k$-sets, then $C_k$ should not have an access for this particular transaction. Hence, the amount of entries in $C_k$ may be lesser than number of transactions in database, especially at later iterations of the algorithm.

The AprioriTID algorithm also used Apriori_gen function to determine candidate itemsets before pass begins. The transaction in $C_k$ is the form $\langle TID, \{X_k\} \rangle$, where each $X_k$ is a potentially a large $k$-itemset present in the transaction with identifier $TID$.

This member of $C_k$ corresponding to the transaction $t$ is defined as

$$C_k = \langle t.TID, \{c \in C_k \mid c \text{ contained } t \} \rangle$$

The main drawback of AprioriTID is that the modified data structure can be much larger than initial database and only faster in the later stage of passes.
5.4.2 Other Algorithms

This proposed work is compared with FP-Growth, AprioriTid, Apriori algorithm, and éclat and used to search the itemsets using rapid search methods. The proposed work is compared, analyzed and summarized with the following algorithms: Apriori, Fp-Growth and Eclat. These algorithms are discussed in detail in the previous chapters. The proposed algorithm RSDCA is compared with above said algorithms to find the efficiency rate and the execution time.

5.5 REFINED SEARCH DIVIDE AND CONQUER (HYBRID) ALGORITHM PROCEDURE

A huge number of candidates generation and the repeated scanning of large transaction databases to test those candidates are major costs in Apriori-like methods. It is a bottleneck for Apriori-like methods. Rapid Search Data mining Algorithm (RSDCA) is designed and developed to avoid unnecessary candidate-generation. This chapter proposes an efficient algorithm for mining frequent patterns from Frequent Pattern (FP).

For mining frequent patterns, information from transaction databases is essential and compulsory. Therefore, the proposed method extracts the brief information for frequent pattern mining and stores it into one format, and then it is used for frequent pattern mining. This chapter develops a compact data structure, called RSDCA, to store the information with no redundant information for frequent pattern mining.
To develop a well-structured data structure for well-organized frequent-pattern mining, first inspect the example. Based on the following explanation, the well-defined data structure is designed.

1. While the frequent items will act as an essential role in the frequent-pattern mining, the one scan transaction is important to find the set of frequent items.

2. To avoid the repeated scanning of the main transaction database, it is very important to store the set of frequent items of every transaction in a well-defined structure.

3. When more than one transactions share a set of frequent items, it is easy to combine the shared sets with the number of event recorded as count. If the frequent items in all transactions are scheduled in a predetermined order then it is easy to check whether two sets are same or not.

4. While two transactions uses the same prefix, the shared parts can be combined by using one prefix structure and the count is recorded correctly.
5.5.1 Algorithm

The proposed RSDCA is a minimal search data mining algorithm to search the itemset from the voluminous databases. FP-Growth approach is based on divide and conquers strategy for producing the frequent item sets. FP-growth is mainly used for mining frequent item sets without candidate generation an unnecessary burden in frequent item sets.

Major steps in FP-growth is:

Step1: Firstly compresses the database showing frequent item set in to FP-tree. FP-tree is built using 2 passes over the dataset.

Step2: Divides the FP-tree in to a set of conditional database and mines each database separately, thus extract frequent item sets from FP-tree directly.

It consist of one root labeled as null, a set of item prefix sub trees as the children of the root, and a frequent item header table. Each node in the item prefix sub tree consists of three fields: item-name, count and node link where item-name registers which item the node represents; count registers the number of transactions represented by the portion of path reaching this node, node link links to the next node in the FP-tree.

Each item in the header table consists of two fields’ item name and head of node link, which points to the first node in the FP-tree carrying the item name. The pseudo code of mining on FP-tree is depicted in Figure.5.1.
Input: constructed FP-tree
Output: complete set of frequent patterns
Method: Call FP-growth (FP-tree, null).

Procedure FP-growth (Tree, α) {
  1) if Tree contains a single path P then
  2) for each combination do generate pattern $\beta \sqcup \alpha$ with support = minimum support of nodes in $\beta$.
  3) Else For each header $a_i$ in the header of Tree do {
    4) Generate pattern $\beta = a_i \cap \alpha$ with support = $a_i$.support;
    5) Construct $\beta$.s conditional pattern base and then $\beta$.s conditional FP-tree Tree $\beta$
    6) If Tree $\beta$ = null
    7) Then call FP-growth (Tree $\beta$, $\beta$) } }

Figure 5.1: Pseudo code of FP-Growth algorithm

<table>
<thead>
<tr>
<th>TID</th>
<th>ITEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₁</td>
<td>{a,b}</td>
</tr>
<tr>
<td>T₂</td>
<td>{b,c,d}</td>
</tr>
<tr>
<td>T₃</td>
<td>{a,c,d,e}</td>
</tr>
<tr>
<td>T₄</td>
<td>{a,d,e}</td>
</tr>
<tr>
<td>T₅</td>
<td>{a,b,c}</td>
</tr>
</tbody>
</table>

129
FP-Tree construction

Pass 1:

- First scan the database and find the support count for each item. Then discard the infrequent items whose support count is less than the minimum support count.

- Sort the frequent items in decreasing order based on their support count. In our example a, b, c, d, e. Use this order when building the FP-Tree, so common prefixes can be shared.

Pass 2: Construct the FP-Tree

- Read transaction 1: \{a, b\}. Create 2 nodes a, b and the path null $\rightarrow$ a $\rightarrow$ b. Set counts of a, b to 1.

- Read transaction 2: \{b, c, d\}. Create 3 nodes for b, c and the path null $\rightarrow$ b $\rightarrow$ c $\rightarrow$ d$\rightarrow$. Set counts to 1. Note that although transaction 1 and 2 share b, the paths are disjoint as they don’t share a common prefix. Add the link between the b’s.
- Read transaction 3: \{a, c, d, e\}. It shares common prefix item with transaction 1 so the path for transaction 1 and 3 will overlap and the frequency count for node a will be incremented by 1. Add links between the c’s and d’s.

- Continue until all transactions are mapped to a path in the FP-Tree.

![Figure 5.2 Complete FP-Trees](image)

**Disadvantages of FP-Growth**

- FP-Tree may not fit in memory

- FP-Tree is too expensive to build. It takes time to build, but once it is built, frequent item sets are read off easily. Time is wasted especially if support threshold is high, as the only pruning that can be done is on
single items. Support can only be calculated once the entire data-set is added to the FP-Tree.

- If user wants to change the minimum count, the entire FP-Tree must be reconstructed. It is difficult one.

5.5.1.1 Proposed Refined Search Divide and Conquer (Hybrid) Algorithm (RSDCA)

Proposed RSDCA extracts frequent item sets from sub tree only, instead of constructing the full FP-Tree. The FP-Tree may not fit in memory and expensive to built. In proposed Refined Search Divide and Conquer Algorithm extraction of frequent item sets starts from its conditional sub tree only. First it traverses the transaction database and scan the supports of the items is calculated and manipulated.

Then items are sorted in ascending order with respect to their support. So, no need to create a complete FP-Tree for the database. The conditional sub-tree is enough for extract the frequent item sets from the transactional database. And also the sub-tree needs minimum memory space only. So it can fit into the system provided memory space itself.

If any items support count is less than the minimum support then that item is specified as infrequent and that item is discarded from the support table not from the main table. It processes the transactions directly from its conditional sub tree, so its main strength is its simplicity. The step by step processing of RSDCA as follows:
**Step 1:** In the initial scan, the supports of the items are calculated. Then the items in the database are sorted in ascending order with respect to their support. Let us consider a transactional database $D$ (Table 3.5.1), suppose the minimum support count is 3. The items whose support count is less than minimum support are specified as infrequent items and discarded.

In transaction database, item support, transaction database with items in transactions sorted in ascending order with respect to their support. In our example: $e,d,c,b,a$. Use this order when building the conditional sub tree, so common prefixes can be shared.

**Step 2:** Construct the conditional sub tree to extract the frequent item sets. Bottom-up algorithm is used from the leaves towards the root to extract the frequent item sets in normal FP-Tree. In the proposed RSDCA, first consider the item which support count is least. In our example the item $e$ least support count, and look for frequent item sets ending in $e$, then $de$, etc… then $d$, then $cd$, etc… Here $e$ has the least support count. So, one can consider a leaf node of the tree. First construct the sub trees ending in an item $e$. Each sub tree is processed recursively to extract the frequent item sets with the help of divide and conquer method. Solutions are then merged.
Example: The sub tree for e will be used to extract frequent item sets ending in e, de, ce, be, and ae, then in cde, bde, cde, etc. Divide and conquer approach is used recursively to find the frequent item sets.

Above shown table (Table 5.1: Transaction data set) item e is not in the transaction $T_1, T_2, T_5, T_6, T_7, T_8$, and $T_9$. So leave that transaction. But item e is in $T_3, T_4$ and $T_{10}$. Construct the sub-tree using this $T_3, T_4$ and $T_{10}$.

Let minimum support count = 3 and extract all frequent item sets containing e.

- Construct the sub-tree for e. Check if e is a frequent item by adding the counts along the linked list. If so, construct it.
- To build the sub-tree consider only the transactions containing a particular item sets and then removing that item sets from all transaction.

**Sub-tree construction for e**

1. Read the transaction which contains the item set e in the transaction table. The item set e is not found in the first and second transaction. The third transaction only contains e, that is {a, c, d, e}. Create 4 nodes for a, c, d,e and the path null $\rightarrow a \rightarrow c \rightarrow d \rightarrow e$. Set counts of a, c, d, e to 1.
2. Read the fourth transaction which contain the item set e: \{a, d, e\}. It shares common prefix item a with transaction 3 so the path for transaction 3 and 4 will overlap and the frequency count for node a will be incremented by 1 and 2 nodes for d and e will be created and add the link between the d’s and e’s.

3. Read the tenth transaction which contains the item set e :\{ b, c, e\}.

Here no common prefix item to share the item with prefix item. So
create 3 nodes for b, c, and e and the path null → b → c → e. Set counts 2 to b and c and add the link between c and between e.

As e is frequent, find the frequent item sets ending in e, i.e. de, ce, be and ae. i.e, decompose the problem recursively to find the frequent item set. To do this, first obtain the conditional FP-Tree for e.

**Conditional sub-tree construction for e**

- The conditional sub-tree for e would be built by the transactions containing e and then removing that item set from all transactions.

Example: Conditional Sub-tree on e.

<table>
<thead>
<tr>
<th>TID</th>
<th>ITEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>T₃</td>
<td>{a,c,d,e}</td>
</tr>
<tr>
<td>T₄</td>
<td>{a,d,e}</td>
</tr>
<tr>
<td>T₁₀</td>
<td>{b,c,e}</td>
</tr>
</tbody>
</table>

*Table 5.2 Transaction data set for item e*
- Update the support counts of e along the prefix paths. It reflects the number of transactions containing e, b and c should be set to 1.

- Remove the nodes e from sub-tree.
- Remove infrequent items from the paths. Example: b has a support of 1. That means there is only 1 transaction containing b and e and b has no link. So b is infrequent and remove b. Here c and d also has the support count 1, but it will not be removed. Because the items c and d has the link between c’s and d’s. so the count is 2, and they are frequent. So that items c and d are not removed.

![Conditional sub tree for e](image)

- The conditional sub-tree for e is used to find the frequent item sets ending in de, ce, and ae. Find the prefix paths form the conditional sub tree e for each item such as de, ce, and ae. Recursively extract frequent item sets and generate conditional sub-tree for each items using divide and conquer algorithm.
Example:

1. $e \rightarrow de \rightarrow ade$ (Here $\{d, e\}$, and $\{a, d, e\}$ are frequent)

Prefix paths ending in $de$ is

![Diagram of Prefix paths ending in de]

Conditional FP-Tree for $de$ is

![Diagram of Conditional FP-Tree for de]

2. $e \rightarrow ce(\{c, e\})$ (Here $\{c, e\}$ is frequent)

![Diagram of Conditional FP-Tree for ce]
Conditional FP-Tree for ce is

3. Similarly do the whole thing for b …etc). In this way the frequent item sets are found.

<table>
<thead>
<tr>
<th>suffix</th>
<th>Frequent Item sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>{e},{d,e},{a,d,e},{c,e},{a,e}</td>
</tr>
<tr>
<td>d</td>
<td>{d},{c,d},{b,c,d},{a,c,d},{b,d},{a,b,d},{a,d}</td>
</tr>
<tr>
<td>c</td>
<td>{c},{b,c},{a,b,c},{a,c}</td>
</tr>
<tr>
<td>b</td>
<td>{b},{a,b}</td>
</tr>
<tr>
<td>a</td>
<td>{a}</td>
</tr>
</tbody>
</table>

Table 5.3. Frequent Item sets

Analysis:

The *FP-tree* construction takes exactly two scans of the transaction database:

1. The first scan collects the set of frequent items.
2. The second scan constructs the Fast Mining
The cost of inserting a transaction $t$ into the $FP$-tree is the set of frequent items in $t$.

5.5.1.1 Refined Search Divide and Conquer (Hybrid) Algorithm (RSDCA)

Input : A database DB, represented by RSDCA constructed
Output  : The complete set of frequent patterns.
Method : RSDCA (FP-sub tree, null).

Algorithm: RSDCA.

Input:
- $d$, a transaction database.
- $Min$-sup, the minimum support count threshold.

Output: The complete set of frequent patterns.

Method:

The RSDCA is constructed in the following steps:

(a) Scan the transaction database $D$ once. Find the set of frequent items $F$, and their support counts. Sort $F$ in support count ascending order as $L$, the list of frequent items.

(b) Create the root of RSDCA and label it as “null”. For each transaction $Trans$ in $D$ do the following. Select and sort the frequent items in $Trans$ according to the order of $L$.

(c) To build the sub-tree for a particular item, first consider the transaction containing that item set only, and then remove that item from all transaction. Using this result construct the sub tree for that item.
(d) Let the sorted frequent item list in transaction be \([p/P]\), where \(p\) is the first element & \(P\) is the remaining list. Call \(\text{ins}_\text{tree} ([p/P], T)\). It performs as follows.

- If \(T\) has a child \(N\) then \(N.\text{item-name} = p.\text{item-name}\), and increment \(N\)’s count by 1; else create a new node \(N\), and assign its count be 1 and its parent link be linked to \(T\), and its node-link to the odes with the same item-name via the node-link structure.
- If \(P\) is non empty, call \(\text{ins}_\text{tree} (P, N)\) recursively.

**Procedure**

Procedure RSDCA (FP-sub tree, \(A\))

if tree contains a single path \(P\) then

{  
  let \(P\) be the single prefix-path part of Tree;
  let \(Q\) be the multiple-path part with the top node is labeled as null root;
  generate pattern \(BUA\) with support \(-\text{count} = \) minimum support count of nodes in \(B\);
}

else for each \(a_i\) in the heard of Tree

{  
  construct pattern \(B= a_i \cup A\) with support-count = \(a_i \cdot \text{support-count}\);
  construct \(B\)’s conditional pattern base and then \(B\)’s conditional sub-tree \(Tree_B\);  
}
if Tree \_B \neq \_ then \\

call RSDCA(FP-sub tree\_B, B); \\
\} \\

With the properties, the algorithm correctly finds the complete set of frequent itemsets in transaction database \textit{DB}.

\textbf{5.5.2 Description}

The major bottlenecks in \textit{Apriori} and \textit{Fp-Growth} algorithms are in some aspects.

1. The \textit{Apriori} algorithm needs to scan the database multiple times. Multiple database scans are very costly in huge database. Reducing the number of scans in the database improve the efficiency of this \textit{Apriori} algorithm.

2. The \textit{Apriori} algorithm generates a huge number of candidates. It is tedious one to storing and counting these candidates. To hit this problem, some studies focus on reducing the number of candidates. One dominant operation in the \textit{Apriori} algorithm is support counting.

3. FP-Tree may not fit in memory

4. FP-Tree is expensive to build. It takes time to build, but once it is built, frequent item sets are read off easily. Time is wasted especially if support threshold is high, as the only pruning that can be done is on
single items. Support can only be calculated once the entire data-set is added to the FP-Tree.

5. If user wants to change the minimum count, the entire FP-Tree must be reconstructed. It is difficult one.

This proposed approach *fastly* reads $M$ transactions at a time and updates the appropriate support counts. For like when the “Bus” reaches the end of the transaction database, it has made one scan over the data and it starts over at the beginning for the next scan. The “passengers” on the bus are *candidate itemsets*. If an *itemset* is on the bus, its support is updated each time a transaction containing the *itemset* is scanned.

At the start of the first scan, the passengers on the bus are the set of *length-1* candidates. At each stop, checks the passengers on the bus according to the following rules.

- When the support count of a candidate *itemset* $X$ passes the support threshold, check whether $X$ can be joined with some other *frequent itemsets* with the same length to generate new candidates. If so, add the new candidate on the bus. Generate a *length-k*, *itemset* $X$, *subsets of* $X$ have accumulated support greater than or equal to the support threshold. When a candidate *itemset* $X$ has travelled for one complete scan, it is removed from the bus. If at that time, the support for $X$ is greater than or equal to the support threshold, output $X$ and its support as a frequent pattern.
For example, let us consider mining a transaction database $TDB$ with 20,000 transactions and support threshold 50. Let the interval between stops be 5,000. By overlapping the counting of different lengths of itemsets, it can save some database scans. On the other hand, it can also explore efficient support counting. It optimizes the structure used in the Apriori algorithm for counting candidates.

Frequent items in each candidate are sorted in support ascending order according to their popularity in the first $M$ transactions. Such an order can reduce the number of inner loops in counting. Reordering items incurs some overhead, but for some data, it may be beneficial overall.

### 5.5.3 Divide and Conquer approach

The Divide and Conquer is simple and very efficient algorithm. In this approach the problem is dividing into several small problems and at last the solution is combined, so it is called Divide and Conquer approach. Heapify, quick sort, merge sort, binary search, Strassen’s et al fast matrix multiplication, and the Fast Fourier Transform are the well-known examples of Divide and Conquer approach. This approach is applied in several data structures such as binary search tree, multi way search tree, skip tree, etc.

Divide and Conquer approach is a powerful tool for solving complex problems such as Tower of Hanoi. In these examples, if the size of the problem is constant, splitting and combining the solutions is proportional to
the size of the problem “n” then the cost of the divide and conquer algorithm will be \(O(n \log n)\).

This approach normally adapted in multi processor system for execute the programs, where the communication between processors need not planned in advance. This approach efficiently uses the memory caches. For this reasons, the sub problems are solved with in the cache. Divide and Conquer algorithm yield accurate results than an equivalent iterative method.

For example, one can add "N" numbers either by a simple loop that adds each datum to a single variable, or by a D&C algorithm called pair wise summation that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate.

Mining frequent patterns from large databases plays an essential role in many data mining tasks and has broad applications. Most of the previously proposed methods adopt apriori-like candidate-generation-and-test approaches. However, those methods may encounter serious challenges when mining datasets with prolific patterns and/or long patterns.

In this work, we develop a class of novel and efficient pattern-growth methods for mining various frequent patterns from large databases. Pattern-growth methods adopt a divide-and-conquer approach to decompose both the mining tasks and the databases. Then, they use a pattern fragment growth method to avoid the costly candidate-generation-and-test processing
completely. Moreover, effective data structures are proposed to compress crucial information about frequent patterns and avoid expensive, repeated database scans. A comprehensive performance study shows that pattern-growth methods, FP-growth and H-mine, are efficient and scalable. They are faster than some recently reported new frequent pattern mining methods.

Interestingly, pattern growth methods are not only efficient, but also effective. With pattern growth methods, many interesting patterns can also be mined efficiently, such as patterns with some tough non-antimonotonic constraints and sequential patterns. These techniques have strong implications to many other data mining tasks.

The Refined Search Divide and Conquer (Hybrid) Algorithm (RSDCA) is compared with various existing algorithms such as Apriori, FP-Growth, AprioriTID and Eclat. The execution time of the proposed algorithm is very less than the above mentioned algorithms for different datasets.

5.6 DATASET DESCRIPTION

5.6.1 Chess

The chess dataset generated and described by Alen Shapiro and donated by Rob Holte [45]. It consists of 3196 instances and 36 attributes. The attribute Summaries: Classes White-can-win (won) and White-cannot-win (nowin). The missing attributes is none. The classes Distribution are as follows: In 1669 of the positions (52%), White can win and in 1527 of the
positions (48%), White cannot win. The format for instances in this database is a sequence of 37 attribute values. Each instance is a board description for attribute is the classification: "win" or "no win".

5.6.2 Connect

This dataset contains 42 attributes and 67557 transactions. This database contains all legal positions in the game of connect-4 for a 6x7 grid, in which neither player has won yet, and in which the next move is not forced. Thus, every attribute contains a nominal value which describes if a given position is void or if it has been occupied by one player. The task is to predict which player is likely to win the match.

5.6.3 Mushroom

Mushroom records drawn from the Audubon Society Field Guide to North American Mushrooms written by G. H. Lincoff and Alfred A. Knopf donated by Donor: Jeff Schlimmer [45]. This dataset includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like leaflets three let it be for Poisonous Oak and Ivy. The dataset contains 119 numbers of attributes which are valued nominally and 8124 transactions.
5.6.4 Pumsb

Pumsb is a dataset containing “n” number of attribute information also its create and populate the data table to find the data table.

5.7 EXPERIMENTAL RESULTS

Refined search Divide and Conquer (Hybrid) Algorithm (RSDCA) testified in voluminous database. Database is divided into multiple partitions where each partition can be held in main memory. The Divide and Conquer method is used to divided the database into multiple partitions, and it used recursively to find the frequent itemsets. The whole voluminous database is rapidly scanned only twice.

First scan: Partitions are read into main memory one by one. Local frequent patterns are mined with respect to their relative support threshold.

Second scan: Consolidates global frequent patterns. Each global frequent pattern must be frequent in at least one partition. Therefore, only those local frequent patterns should be counted and tested in the second scan.

The major conclusions for this fast mining method are in two aspects.

i. Partitioning the database is non-trivial when the database is biased.
ii. On the other hand, a low global support threshold may lead to a much lower local threshold and thus produce a huge number of local frequent patterns. By fast mining, it can observe the patterns hidden behind the data more accurately, more steadily and more proficiently.

The following figure 3.3 illustrated the comparison of exact time obtained by implement the proposed RSDCA algorithm. Comparison is state with Apriori, FP-Growth, AprioriTid and Eclat with support level used to calculate the execution time testified through various datasets.
Figure 5.3 Comparison of Execution Time between Apriori, FP-Growth, AprioriTid, RSDCA and Eclat with various support levels using Chess Dataset
**Figure 5.4** Comparison of Execution Time between Apriori, FP-Growth, AprioriTid, RSDCA and Eclat with various support levels using Connect Dataset
Figure 5.5 Comparison of Execution Time between Apriori, FP-Growth, AprioriTid, RSDCA and Eclat with various support levels using Mushroom Dataset
Figure 5.6 Comparison of Execution Time between Apriori, FP-Growth, AprioriTid, RSDCA and Eclat with various support levels using Pumb Dataset
The proposed RSDCA algorithm is compared with traditional Apriori, Fp-Growth, Eclat and AprioriTid. The rapid searching technique is adapted with frequent pattern mining process. The proposed RSDCA algorithm is performed better than other algorithms. The proposed algorithm is working well in low support level, compared with high. This proposed algorithm is reduced 10% of execution time in low support level.

5.8 SUMMARY

This chapter clearly demonstrates the proposed RSDCA algorithm for association rule mining efficiency in terms of speed and minimizing wastage of candidate sets. The various JAVA programs are written for RSDCA and other existing algorithms. Based on the results with various datasets, The proposed algorithm is working well in low support level, compared with high and reduced 10% of execution time in low support level the graphs are drawn using MATLAB. The experimental and comparative studies are further described in Chapter 6.