Chapter 3

Distributed Canonical Order Tree

The advancement in computation, storage and computer networks is providing huge transactional data, posing challenges for organizing the same at the central or distributed databases. This in turn, posing challenges for the knowledge discovery process. This chapter develops representation of transactional database as a tree for given canonical order, known as CanTree and presents the proposed parallelize CanTree Construction on distributed environment. Introduction is given in Section 3.1. A brief overview of CanTree is provided in Section 3.2. The Distributed CanTree Construction has been developed by using a novel five step mechanism, is presented in Section 3.3. Experimentation and analysis are provided in Section 3.4.

3.1 Introduction

Data Mining has attracted a great deal of attention in the information domain, due to wide availability of huge amounts of data and the imminent need for converting such data into useful information and knowledge. The knowledge gained can be used for applications such as market analysis, fraud detection, customer retention and scientific discovery. Several algorithms were proposed for these applications [ZWZ03], [HK06], [AIS93A], [AS94], [SON95], [STA98], [MC02].

Data mining refers to extracting or mining knowledge from large amounts of data. One of the fundamental and important tasks of data mining is the mining of frequent patterns. The main utility of Frequent Patterns is to derive association rules which in turn
are useful for building knowledge. Various types of Association Rules starts from
Boolean Association rules to complex kind of generalized Association Rules are found in
literature [HK06].

Association rules represent an important class of knowledge that can be discovered
from data repositories [HK06], [TBAR97], [T98], [TC00]. The work on association rule
mining began with the development of the AIS algorithm [AIS93] for discovering
association rules. Some of the researchers [C98] [SHS00] [HPY00] suggests several
improvements. The FP-Tree algorithm [HPY00] builds a special tree structure in main
memory to avoid multiple passes over database.

In general the procedure followed in association rule mining algorithm can be broadly
divided into the following two phases.

1. Finding all frequent itemsets , each of these itemsets will occur at least as frequently as
   a pre-determined minimum support threshold.
2. Generating strong association rules from frequent itemsets.

Some algorithms developed for discovering unknown patterns through distributed
[GRSFM97], [SP98] and parallel [SK96], [ZOP96] environments.

On the other hand, even the parallel algorithms [SK96], [ZOP96] [ZPOL97] for
association rule mining has attracted researchers. The parallel environment is discussed
in Chapter 5. Zaki [Z01] designed pSPADE algorithm to overcome some common
problems of parallel algorithms. pSPADE algorithm has two phases. The first phase
deals with the sequence search space and the second phase deals with the association
work. These two phases carried out in distributed environment by focusing on shared-
memory systems and uses a recursive dynamic load balancing scheme for association
mining.
To overcome the limitations of the CanTree[LKH07], the present study in this thesis proposes the following five steps mechanism similar to [LWZZC08] in distributed environment.

- Sharding
- Tree Construction
- Merging
- Pattern Generation
- Association Rule Mining

By adapting apt divide-and-conquer philosophy in Sharding step. A heuristic is developed and demonstrated in Section 3.3 and 3.4 respectively.

The Association Rule Induction is of two step process.

- Find the frequent itemsets (minimum support).
- Form the relevant association rules (minimum confidence).

## 3.2 Canonical Order Tree (CanTree)

As stated in the Literature Survey, tree-based algorithms FELINE [CZ03] and AFPIM [KS04] suffer from several problems/weaknesses. The FELINE algorithm [CZ03] requires a large amount of computation for searching common items and mergeable paths during the construction of CATS trees. In addition, it needs extra downward traversals during the mining process. The AFPIM algorithm [KS04] requires an additional mining parameter (namely, preMinsup). Finding an appropriate value for this parameter is not
straightforward. Leung et al. [LKH07] stated that both FELINE and AFPIM algorithms need lots of swapping, merging, and splitting of tree nodes, because items in the trees are arranged according to a frequency-dependent ordering. So, when the database is updated, item frequencies may have changed. This results in changes in the ordering.

Leung et al. proposed CanTree [LKH07] without considering any candidate item sets and solved the above mentioned problems. The construction of the CanTree requires only one database scan. This is different from the construction of an FPTree [HPY00] where two database scans are required. In CanTree, items are arranged according to some canonical order, which can be determined by the user prior to the mining process or at runtime during the mining process. Specifically, items can be consistently arranged in lexicographic order or alphabetical order as shown in illustrations in Section 3.2.1.

The following are some of the properties of CanTree:

• Property 1: Items are arranged according to a canonical order, which is a fixed global ordering.

• Property 2: The ordering of items is unaffected by the changes in frequency caused by incremental updating.

• Property 3: The frequency of a node in the CanTree is at least as high as the sum of frequencies of all its children.

3.2.1 Illustrations of CanTree Construction

Consider the sample database as in Table 3.1. Figure 3.1 shows the original tree (CanTree) and the trees after each transaction. These trees are constructed by considering the canonical order in lexicographic order.
Table 3.1 Sample Database to illustrate CanTree

<table>
<thead>
<tr>
<th>TID</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>{a,d,b,g,e,c}</td>
</tr>
<tr>
<td>$t_2$</td>
<td>{d,f,b,a,e}</td>
</tr>
<tr>
<td>$t_3$</td>
<td>{a}</td>
</tr>
<tr>
<td>$t_4$</td>
<td>{d,a,b}</td>
</tr>
<tr>
<td>$t_5$</td>
<td>{a,c,b}</td>
</tr>
<tr>
<td>$t_6$</td>
<td>{c,b,a,e}</td>
</tr>
<tr>
<td>$t_7$</td>
<td>{a,b,c}</td>
</tr>
<tr>
<td>$t_8$</td>
<td>{a,b,c}</td>
</tr>
</tbody>
</table>

(a) After inserting $t_1$  (b) After inserting $t_2$  (c) After inserting $t_3$  (d) After inserting $t_4$
After inserting $t_5$ (f) After inserting $t_6$ (g) After inserting $t_7$ (h) After inserting $t_8$

**Figure 3.1:** The Progressive CanTree from $t_1$ to $t_8$ or (a) to (h) respectively.

### 3.2.2 Complexity and Limitations

CanTree significantly reduces computation and time because they easily find mergeable paths and require only upward path traversals. The computational time required to construct the CanTree is $O(mn)$ where $m$ is the maximum length of the transaction and $n$ is the number of transactions.

Although CanTree offers a simple single-pass construction process, it usually yields poor compaction in tree size compared to the FP-Tree [HPY00]. However tree-
restructuring [TAJL08] process can be carried out in order to reduce tree size and falls into post processing category.

The construction time of the CanTree can be reduced effectively by adopting distributed computing strategies. The following Section develops CanTree Construction by distributed paradigm.

3.3 Distributed CanTree Construction

This chapter presents a proposed parallelize CanTree Construction procedure in distributed environments. This method partitions the computations in such a way that each compute node executes independently. Such partitions eliminate computational dependencies between compute nodes and thereby communication overhead among them.

This method is focused mainly on algorithms such as Sharding and Merging which is given in detail in the successive sections. These algorithm help in reducing memory use and computational cost on each node. The above factors lead to design of the construction of CanTree in distributed environment.

This model adapts divide and conquer methodology. The given database is split into several parts and distributed over different processors (Sharding Algorithm). This process of splitting database is called sharding [LWZZC08]. Then on each processor, the transactions are ordered by the user, prior to the mining process. The predefined canonical order has mainly two advantages.

- CanTree requires only one scan
- Canonical order makes the tree merging process simpler.
This divide and conquer approach can also be implemented in Map Reduce Environment of HPC (High Performance Computing) [LWZZC08]. In proposed method the distribution of data to different processors is done using threads. The tree construction is carried out using the OpenMP (Multi-processing tool). The synchronization between processors is achieved internally by these OpenMps.

### 3.3.1 Illustrations of Distributed CanTree Construction

![CanTrees at distributed node](image)

(a) After inserting \( t_1 \) and \( t_2 \) processor

(b) After inserting \( t_3 \) and \( t_4 \)

(c) After inserting \( t_5 \) and \( t_6 \)

(d) After inserting \( t_7 \) and \( t_8 \) processor

Figure 3.2 CanTrees at distributed node

Once the CanTrees are constructed at different processors they are united at the root node (Merging Algorithm). Then the mining process is carried out on amalgamated CanTree by applying the FP-growth [HPY00] algorithm. Then the frequent patterns are
generated by satisfying the predefined minimum support and the Association Rules are extracted by satisfying predefined minimum confidence. CanTree Construction algorithm, FP-growth and Association Rule Extraction algorithms are provided and the same is implemented in Java Platform.

The proposed five step mechanism has been provided in Section 3.3.2 after giving a motivating illustrations in Section 3.3.1.

![Figure 3.3 Amalgamated CanTrees](image)

The Transactional data given in Table 3.1 is distributed to four processors. Then the CanTrees at each location is shown in the Figure 3.2. The Merging process for obtaining
CanTree corresponding to transactional database given in Table 3.1 is shown in Figure 3.3.

### 3.3.2 Five Steps Mechanism of Distributed CanTree Construction

**Step 1:** **Sharding:** Divide the Database into successive parts and store the parts into different nodes. Such division and distribution of data is called sharding, and each part is called shard.

**Step 2:** **Tree Construction:** CanTree is generated for each shard. The CanTree is generated in multithreaded/multiprocessor environment. The corresponding CanTrees is sent to root node.

**Step 3:** **Merging:** The CanTrees are merged and FP-Growth method is applied on the merged tree to uncover frequent patterns.

**Step 4:** **Pattern Generation:** To generate the frequent patterns from the CanTree.

**Step 5:** **Association Rule Mining:** Extracting the Association Rules from the frequent patterns.

Following are the Algorithms for implementation of the steps 1 to 5.
Algorithm 3.1 Sharding

**Input:** Transaction Database (D), Number of Processors (p)

**Output:** CanTrees $CT_1, CT_2, \ldots, CT_p$

**Method:**

1. Set $t_{\text{count}}$ to number of transactions in ‘D’.
2. $\text{shardingsize} := \text{floor}(t_{\text{count}}/p)$
3. Divide (D) into ‘p’ buckets where each bucket have ‘shardingsize’ transactions, except may be for the last bucket, the bucket are named as $DB_1, DB_2, \ldots, DB_p$
4. Call in parallel for $i = 1$ to $p$
   
   $CT_i := \text{CanTree}(DB_i)$
5. Return $\{CT_1, CT_2, \ldots, CT_p\}$

Algorithm 3.2 CanTree Construction

**Input:** Transaction Database (DB)

**Output:** CanTree (CT)

**Method:**

1. Define the root ‘r’ of CanTree ‘r’;
2. For each transaction $T_i$ in DB
   
   do
3. Reorder $T_i$ based on canonical ordering.
4. Call $r = \text{Construct Tree}(T_i, r)$
5. done
Procedure:

ConstructTree(T, r)

Input: Single Transaction(T), Root of the tree ‘r’

Output: Root of the tree ‘r’

(1) Let the items in ‘T’ be [p/Y], where p is the first element and Y is the remaining sequence
(2) Call Insert_Tree([p/Y], r)
(3) return r

Procedure:

Insert_Tree([p/Y], x)

Input: p: item under consideration
Y: remaining items in the transaction
‘x’: current node of the tree

(1) If ‘x’ has a child ‘n’ satisfying n.itemname=p.itemname
(2) then
(3) increment n’s count by 1
(4) else{
(5) attach [p/Y] as a child node to ‘x’
(6) return;
}
(7) If Y is non-empty, call Insert_Tree(Y, n) recursively.
Algorithm 3.3  Merge (Merging two CanTrees)

Input: X, Y are the roots of CanTree1, CanTree2

Output: X

(1) for each child node ‘c’ of Y
(2) do
(3) If X has a child node ‘x’ such that x.item_name=c.item_name
(4) then
(5) increment ‘x’ count value by the count value of ‘c’
(6) call Merge(x,c) recursively
(7) else
(8) attach the sub_tree rooted at ‘c’ in Y to X
(9) done
Algorithm MergeP (Merging ‘P’ number of CanTrees)

**Input:** P: number of trees to merge

\[ \text{CT}_1, \text{CT}_2, \ldots, \ldots, \text{CT}_p \text{CanTrees} \]

**Output:** MCT: Merged CanTree

1. If P > 2{
2. \( T_1 := \text{MergeP}(\text{CT}_1, \text{CT}_2, \ldots, \ldots, \text{CT}_{\lfloor p/2 \rfloor}, \text{CT}_{\lceil p/2 \rceil}) \)
3. \( T_2 := \text{MergeP}(\text{CT}_{\lceil p/2 \rceil +1}, \ldots, \ldots, \text{CT}_p, p - \lfloor p/2 \rfloor) \)
4. \( \text{MCT} = \text{Merge}(T_1, T_2) \)}
5. else
6. \{ \( \text{MCT} = \text{Merge}(\text{CT}_1, \text{CT}_2) \) \}
7. return MCT

Algorithm 3.4 Frequent_Patterns[HPY00]

**Input:** CanTree

**Output:** The complete set of frequent patterns

**Method:** Call \( FP\text{-Growth}(\text{CanTree}, \text{null}) \).

Procedure \( FP\text{-Growth}(\text{Tree}, \alpha) \)
if Tree contains a single path P

then

for each combination (denoted by β) of the nodes in the path P

do

generate pattern β U α with support = minimum support of nodes in β

done

else

for each a_i in the header of Tree

do

generate pattern β = a_i U α with support = a_i.support

construct β’s conditional pattern base and β’s conditional CanTree

Tree_β

done

if Tree_β ≠ ∅

then call FP-Growth (Tree_β, β)
Algorithm 3.5 Generate Association Rules

**Input:** F: Frequent item sets

**Output:** R: Association Rules

**Method:** R= rules(AR)

Procedure rules(F)

1. R:= φ

2. For all f ∈ F do begin
   a. m:=1
   b. H_m:=∪i∈f{i};
   c. Repeat
      i. For all h ∈ H_m do
         1. If \( \frac{s_f}{s_{f-h}} \geq c_{min} \)
         2. then R:=RU\{((f-h)→h)\};
         3. else H_m:= H_m\{h\}
      ii. H_m+1:=candidates(H_m)
      iii. m:=m+1;
   d. until H_m= φ or m>|f|;
(3) end

(4) return R

Procedure candidates($F_k$)

(1) $E := \emptyset$

(2) forall $f_1, f_2 \in F_k$

    with $f_1 = \{a_1, \ldots, a_{k-1}, a_k\}$

    and $f_2 = \{a_1, \ldots, a_{k-1}, a'_k\}$

    and $a_k < a'_k$ do begin

(2) $f := f_1 \cup f_2 = \{a_1, \ldots, a_{k-1}, a_k, a'_k\}$

(3) if $\forall a \in f : f - \{a\} \in F_k$

(4) then $E := E \cup \{f\}$

(5) end

(6) return $E$

### 3.4 Experimentation and Analysis

Tests are performed using Pima Indian diabetes data set by repeatedly using the same size of records in order to have different size dataset and implemented the above algorithms. Experiments are performed using Intel processors on Java platform. Different libraries like JOMP and Xstream libraries are used. The goal is to define and implement an OpenMP set of directives and library routines for shared memory parallel
programming in Java. It is possible to write shared memory parallel programs using Java’s native threads model. However, a directive system has a number of advantages over the native threads approach. The resulting code is much closer to a sequential version of the same program. Indeed, with a little care, it is possible to write an OpenMP program which compiles and runs correctly when the directives are ignored. This makes subsequent development and maintenance of the code significantly easier. It is also assumed that, with the increasing familiarity of programmers with OpenMP, it would make parallel programming in Java a more attractive proposition. Another problem with using Java native threads is that for maximum efficiency on shared memory parallel architectures, it is necessary to use exactly one thread per processor and to keep these threads running during the whole lifetime of the parallel program. To achieve this, it is necessary to have a runtime library which dispatches tasks to threads, and provides efficient synchronization between threads. In particular a fast barrier is crucial to the efficiency of many shared memory parallel programs. Such barriers are not trivial to implement and are not supplied by the java.lang.Thread class. Similarly loop self-scheduling algorithms require careful implementation in a directive system. This functionality is also supplied by the runtime library.

### 3.4.1 Sample Script/Steps

#### Compiling using JOMP

Using the following command to compile a file using JOMP

```
Java jmp.compiler.Jomp MyFile
```

where as MyFile.Jomp is input file.

Xstream is a simple library to serialize objects to XML and back again.
• Create element name to class name aliases for any custom classes using xstream.alias(String elementName, Class cls);

• Convert an object to XML using xstream.toXML(Object obj);

• Convert XML back to an object using xstream.fromXML(String xml);

The performance of the five-step Methodology performance is evaluated using Pima Indian Diabetes Data Set. The following are the characteristics of dataset and the detailed description of the dataset is available in section 7.3 of chapter 7.

**Data Set Characteristics**

- **Number of Instances**: 768
- **Area**: Life
- **Attribute Characteristics**
  - Integer, Real
- **Number of Attributes**: 8
- **Date Donated**: 1990-05-09
- **Missing Values**: Yes

**Table 3.2**

The following table depicts computational time for conventional method [LKH07] and the present Five Steps Methodology

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Number of Transactions</th>
<th>Conventional Model (Time in seconds)</th>
<th>Proposed Model (Time in Seconds)</th>
<th>Difference (Time in Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9750</td>
<td>0.88</td>
<td>0.78</td>
<td>0.10</td>
</tr>
<tr>
<td>2</td>
<td>17325</td>
<td>1.20</td>
<td>0.97</td>
<td>0.23</td>
</tr>
<tr>
<td>3</td>
<td>25850</td>
<td>1.61</td>
<td>1.08</td>
<td>0.53</td>
</tr>
<tr>
<td>4</td>
<td>34650</td>
<td>1.97</td>
<td>1.11</td>
<td>0.86</td>
</tr>
</tbody>
</table>
Figure 3.4 Computational time involved in conventional and proposed model for different sizes of transactions.

From the Table 3.2 and Figure 3.4, it implies that when the number of transactions are increasing the proposed model computational time is decreasing compared to the conventional model.

As stated by [TAJL08] as the available memory increases beyond GBytes, several prefix-tree data structures, which capture all the database information in a tree, have been proposed for mining frequent patterns with one database scan [CZ03] [LKH07] [LLY07] [LW07]. CanTree, proposed in [LKH07], captures complete database information in a prefix-tree structure, while facilitating incremental and interactive mining via an FP-growth mining technique. Although CanTree requires only one database scan, it usually results in sparse (less compact) tree and poor mining performance compared to the FPTree due to the frequency-independent canonical order of items insertion. CPTree (Compact
Pattern Tree) proposed by [TAJL08_A] which can be efficiently constructed as a prefix-tree structure with one database scan that captures the full database information, inheriting the compactness of the FPTree. This thesis modifies and extends the works of Tanbeer et al. [TAJL08_A] [TAJL08_B] and Vadivel et al. [VVT10] in order to derive incidental and global knowledge through Compact Pattern Trees in distributed environment. These procedures are explained in detail in the next chapter.