# Compact Transaction Database for Efficient Frequent Pattern Mining

Qian Wan and Aijun An
Department of Computer Science and Engineering
York University, Toronto, Ontario, M3J 1P3, Canada
Email: {qwan, aan}@cs.yorku.ca

*Abstract*— **Mining frequent patterns is one of the fundamental and essential operations in many data mining applications, such as discovering association rules. In this paper, we propose an innovative approach to generating compact transaction databases for efficient frequent pattern mining. It uses a compact tree structure, called *CT-tree*, to compress the original transactional data. This allows the *CT-Apriori* algorithm, which is revised from the classical *Apriori* algorithm, to generate frequent patterns quickly by skipping the initial database scan and reducing a great amount of I/O time per database scan. Empirical evaluations show that our approach is effective, efficient and promising, while the storage space requirement as well as the mining time can be decreased dramatically on both synthetic and real-world databases.**

## I. Introduction

A transaction database is a set of records representing transactions, where each record consists of a number of items that occur together in a transaction. The most famous example of transaction data is market basket data, in which each transaction corresponds to the set of items bought by a customer during a single visit to a store. Text documents can also be represented as transaction data. In this case each document is represented by a set of words, which can be considered a transaction. Another example of transaction databases is a collection of web pages, where each page is treated as a transaction containing the out-going links on the page.

Transaction databases have important role in data mining. For example, association rules were first defined for transaction databases [3]. An association rule *R* is an implication of the form $X \Rightarrow Y$, where *X* and *Y* are set of items and $X \cap Y = \emptyset$. The support of a rule $X \Rightarrow Y$ is the fraction of transactions in the database which contain $X \cup Y$. The confidence of a rule $X \Rightarrow Y$ is the fraction of transactions containing *X* which also contain *Y*. An association rule can be considered interesting if it satisfies the minimum support threshold and minimum confidence threshold, which are specified by domain experts.

The most common approach to mining association rules consists of two separate tasks: in the first phase, all frequent itemsets that satisfy the user specified minimum support are generated; the second phase uses these frequent itemsets in order to discover all the association rules that meet a confidence threshold. Since the first problem is more computationally expensive and less straightforward, a large number of efficient algorithms to mine frequent patterns have been developed over the years [1], [2], [4], [5], [8], [11]. In this paper, therefore, we address this problem and focus on optimization of I/O operations in finding frequent patterns.

The most important contributions of our work are as follows.

1) We propose an innovative approach to generating *compact transaction databases* for efficient frequent pattern mining. Each unique transaction of the original database has only one entry in the corresponding compact database, with a count number recording the number of its occurrence in the original database.
2) We design a novel data structure, *Compact Transaction Tree* (*CT-tree*), to generate a *compact transaction database*, and revise the *Apriori* algorithm as *CT-Apriori* to take advantage of *compact transaction databases*. Experiment results show that our approach is very practical, and the amount of disk space as well as the running time can be decreased dramatically on both synthetic and real-world databases.
3) These techniques can be easily extended to other data mining fields, such as sequential pattern mining and classification, in which compact databases can not only reduce space requirements but also reduce the overall mining time.

The organization of the rest of this paper is as follows: Section 2 describes the formal definition of *compact transaction database* and discusses its generation. Section 3 develops the notion of a *CT-tree* and introduces an algorithm to generate a compact transaction database from this data structure. Section 4 then explains how to use *CT-Apriori* algorithm to discover frequent patterns from a *compact transaction database* efficiently. Empirical evaluations of our approaches on selected synthetic and real-world databases are presented in Section 5. Finally, we conclude with a discussion of future work in Section 6.

## II. Compact Transaction Database

Before introducing the general idea of a compact transaction database, we will first define a transaction database and discuss some of its properties.

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of *m items*. A subset $X \subseteq I$ is called an *itemset*. A *k*-itemset is an itemset that contains *k* items.

*Definition 2.1:* A *transaction database TDB* = $\{T_1, T_2, \ldots, T_N\}$ is a set of *N* transactions, where each transaction $T_n$ ($n \in$

TABLE I

AN EXAMPLE TRANSACTION DATABASE *TDB*

| TID | List of itemIDs |
|---|---|
| 001 | A, B, C, D |
| 002 | A, B, C |
| 003 | A, B, D |
| 004 | B, C, D |
| 005 | C, D |
| 006 | A, B, C |
| 007 | A, B, C |
| 008 | B, C |
| 009 | B, C, D |
| 010 | C, D |

TABLE II

THE *compact transaction database* OF *TDB*

| ***head*** | | | | |
|---|---|---|---|---|
| Item | C | B | D | A |
| Count | 9 | 8 | 6 | 5 |
| ***body*** | | | | |
| Count | List of itemIDs | | | |
| 3 | C, B, A | | | |
| 1 | C, B, D, A | | | |
| 1 | B, D, A | | | |
| 1 | C, B | | | |
| 2 | C, B, D | | | |
| 2 | C, D | | | |

$\{1, 2, \ldots, N\}$) is a set of items such that $T_n \subseteq I$. A transaction *T* contains an itemset *X* if and only if $X \subseteq T$.

An example transaction database *TDB* is shown in Table I. In *TDB*, $I = \{$A, B, C, D$\}$ and $N$ = 10.

The *support* (or occurrence frequency) of a pattern *A*, where *A* is an itemset, is the percentage of transactions in *D* containing *A*: *support*(*A*)=$\|\{t \mid t \in D, A \subseteq t\}\| \ / \ \|\{t \mid t \in \text{D}\}\|$, where $\|X\|$ is the cardinality of set *X*. A pattern in a transaction database is called as a *frequent pattern* if its *support* is equal to, or greater than a user-specified minimum support threshold, *min_sup*.

Given a transaction database $TDB$ and a minimum support threshold *min_sup*, the problem of finding the complete set of frequent patterns is called frequent-pattern mining. The two most important performance factors of the frequent pattern mining are the number of passes made over the transaction database and the efficiency of those passes [7]. As the data volume increases rapidly these days, the I/O read/write frequency plays an important role for the performance of database mining. Reducing the I/O operations during the mining process can improve the overall efficiency.

Our motivation for building a compact transaction database came from the following observations:

1) A number of transactions in a transaction database may contain the same set of items. For example, as shown in Table I, transaction {A, B, C} occurs three times, and transactions {B, C, D} and {C, D} both occur two times in the same database. Therefore, if the transactions that have the same set of items can be stored in a single transaction with their number of occurrence, it is possible to avoid repeatedly scanning the same transaction in the original database.
2) If the frequency count of each item in the given transaction database can be acquired when constructing the compact database before mining takes place, it is possible to avoid the first scan of database to identify the set of frequent items as most approaches to efficient mining of frequent patterns do.

*Definition 2.2:* The *Compact Transaction Database CTDB* of an original transaction database *TDB* is composed of two parts: *head* and *body*. The *head* of *CTDB* is a list of 2-tuples ($I_n$, $I_c$), where $I_n \in I$ is the name of an item and $I_c$ is the frequency count of $I_n$ in *TDB*; and all items in the *head* are ordered in frequency-descending order. The *body* of *CTDB* is a set of 2-tuples ($T_c$, $T_s$), where $T_s \in$ *TDB* is a unique transaction, $T_c$ is the occurrence count of $T_s$ in *TDB*, and the items in each transaction of the *body* are ordered in frequency-descending order.

The *compact transaction database* of the example transaction database *TDB* is shown in Table II. All four items in *TDB*, {A, B, C, D}, are listed in the head with their frequency count, ordered in frequency-descending order, {C:9, B:8, D:6, A:5}. The body consists of 6 unique transactions, instead of 10 in *TDB* (which is the total transaction count in the compact transaction database). The items in each transaction are ordered in frequency-descending order as well.

In the next section, we will discuss an efficient method to construct the *compact transaction database* by using a novel data structure *compact transaction tree*, denoted as *CT-tree*.

## III. *CT-tree*: DESIGN AND CONSTRUCTION

### A. *Illustration of CT-tree with an example*

To design an efficient data structure for compact transaction database generation, let's first examine an example using the transaction database shown in Table I.

First of all, the root of a tree is created and labeled with "ROOT". Every other node in this tree consists of two parts: item *id* and occurrence *count* of the path from *root* to this node. The *CT-tree* is constructed as follows by scanning the example transaction database once.

For the first transaction, after sorting the items of this transaction in order (we use lexicographic order in this paper), the first branch of the tree is constructed as {(A:0), (B:0), (C:0), (D:1)}. The last node (D:1) records the occurrence of the path ABCD. At the same time, the frequency count of all these items are recorded in a list as [A:1, B:1, C:1, D:1].

For the second transaction, since its ordered item list {A, B, C} shares a common path {A, B, C} with the first branch, no new branch is created, but the occurrence count of the last shared node is incremented by 1 as (C:1). And the frequency

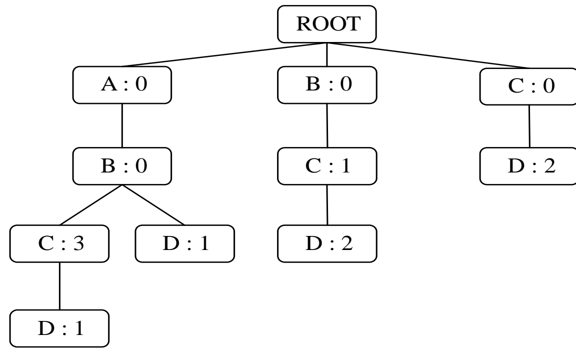


Fig. 1. *CT-tree* for the database *TDB* in Table I.

count of each item in this transaction is incremented by 1 in the list as [A:2, B:2, C:2, D:1].

For the third transaction, since its ordered item list {A, B, D} shares a common path {A, B} with the first branch, one new node (D:1) is created and linked as a child of (B:0). And the frequency count list becomes [A:3, B:3, C:2, D:2].

The scan of the fourth and fifth transactions leads to the construction of two branches of the tree, {(B:0), (C:0), (D:1)} and {(C:0), (D:1)}, respectively. And the frequency count list becomes [A:3, B:4, C:4, D:4].

After the scan of all the transactions, the complete *CT-tree* for the example transaction database *TDB* is shown in Fig. 1. And the frequency count list becomes [A:5, B:8, C:9, D:6], as shown in the *head* part of Table II in frequency-descending order.

Having built a CT-tree, the body part of the compact transaction database is constructed as follows. For every node *v* whose count value is greater than 0 in the *CT-tree*, a unique transaction *t* is created in the *body* part of *CTDB*. The count value associated with the node is recorded as the occurrence count of *t*, and the sequence of items labelling the path from the root to *v* is sorted in frequency-descending order and recorded as the item list of *t*. For example, no transaction is created for node A or B in the leftmost path because their count values are 0. Whereas transactions [3 C B A] and [1 C B D A] are created for nodes C and D, respectively, as shown in the first two rows in the *body* part of Table II.

### *B. Algorithm description*

Having shown the above example, we now define *CT-tree* as follows.

*Definition 3.1:* The **Compact Transaction Tree** (*CT-tree*) of a transaction database $TDB$ is a tree where each tree node *V* (except the root of the tree, which is labeled as "ROOT") is a 2-tuple ($v$, $v_c$) (denoted by $v : v_c$ in the tree), where $v$ is an item in $TDB$ and $v_c$ is the number of occurrences in $TDB$ of a unique transaction consisting of all the items in the branch of the tree from the root to node $V$.

The algorithm for generating a CT-tree from a transaction database and for generating a *compact transaction database* from a CT-tree is described as follows.

Method: Compact Transaction Database Generator.

Input: Original transaction database *TDB*.
Output: Compact transaction database *CTDB*.

```
1:  root[CTtree] ← ROOT
2:  list[item][count] ← null
3:  for each transaction T_n in TDB do
4:     T_o ← sort items of T_n in lexicographic order
5:     insert(T_o, CTtree)
6:  end for
7:  if CTtree is not empty then
8:     list ← sort list[item][count] in count descending order
9:     for each item i in list[item] do
10:       CTDB ← write i
11:       CTDB ← write count[list[i]]
12:    end for
13:    startNode ← child[root[CTtree]]
14:    write(startNode, CTDB)
15: else
16:    output "The original transaction database is empty!"
17: end if
```

**procedure *insert***($T$, *CTtree*)

```
1:  thisNode ← root[CTtree]
2:  for each item i in transaction T do
3:     if i is not in list[item] then
4:        list[item] ← add i
5:     end if
6:     list[count[i]] ← list[count[i]] + 1
7:     nextNode ← child[thisNode]
8:     while nextNode ≠ null and item[nextNode] ≠ i do
9:        nextNode ← sibling[nextNode]
10:    end while
11:    if nextNode = null then
12:       item[newNode] ← i
13:       if i is the last item in T then
14:          count[newNode] ← 1
15:       else
16:          count[newNode] ← 0
17:       end if
18:       parent[newNode] ← thisNode
19:       sibling[newNode] ← child[thisNode]
20:       child[newNode] ← null
21:       child[thisNode] ← newNode
22:       thisNode ← newNode
23:    else
24:       if item i is the last item in T then
25:          count[thisNode]++
26:       else
27:          thisNode ← nextNode
28:       end if
29:    end if
30: end for
```

**procedure *write***(*node*, *CTDB*)

```
1:  if count[node] ≠ 0 then
2:     count[newTrans] ← count[node]
3:     nextNode ← node
4:     while nextNode ≠ root[CTtree] do
```

5: *newTrans* ← insert item[*nextNode*]
6: *nextNode* ← parent[*nextNode*]
7: **end while**
8: **if** *newTrans* is not empty **then**
9: *newTrans* ← sort *newTrans* in *list* order
10: *CTDB* ← write *newTrans*
11: **end if**
12: **end if**
13: **if** child[*node*] $\neq$ *null* **then**
14: ***write***(child[*node*], *CTDB*)
15: **end if**
16: **if** sibling[*node*] $\neq$ *null* **then**
17: ***write***(sibling[*node*], *CTDB*)
18: **end if**

In the first two steps in the above method, the *root* of an empty *CT-tree* and a 2-dimension array *list* are initialized. All items in the original transaction database *TDB* will be stored in this *list* along with their support counts after constructing the CT-tree. From step 3 to step 6, a complete CT-tree is built with one database scan, where each transaction $T$ in *TDB* is sorted and inserted into the CT-tree by calling the procedure *insert*(*T*, *CT-tree*).

Then, the *list* is sorted in frequency descending order and written as the head part of the compact transaction database *CTDB*, as shown in step 8 to step 12. After calling the procedure *write*(*startNode*, CTDB) in step 13 recursively, a unique transaction *newTrans* is written into the body of *CTDB* for each node whose count value is not equal to zero in the *CT-tree*. The occurrence count of *newTrans* is the same as the count value (step 2 of *write*), and the item list of *newTrans* is the sequence of items labelling the path from the node to the root (step 4 to step 7 of *write*), sorted in frequency-descending order (step 9 of *write*). Thus, a complete compact transaction database is generated.

## IV. *CT-Apriori* ALGORITHM

The Apriori algorithm is one of the most popular algorithms for mining frequent patterns and association rules [4]. It introduces a method to generate candidate itemsets $C_k$ in the pass $k$ of a transaction database using only frequent itemset $F_{k-1}$ in the previous pass. The idea rests on the fact that any subset of a frequent itemset must be frequent as well. Hence, $C_k$ can be generated by joining two itemsets in $F_{k-1}$ and pruning those that contain any subset that is not frequent.

In order to explore the transaction information stored in a compact transaction database efficiently, we modify the Apriori algorithm and the pseudocode for our new method, CT-Apriori, is shown as follows. We use the notation $X[i]$ to represent the *i*th item in $X$. The *k*-prefix of an itemset $X$ is the *k*-itemset $\{X[1],X[2],...,X[k]\}$.

Algorithm: *CT-Apriori* algorithm
Input: *CTDB* (Compact transaction database) and *min_sup* (minimum support threshold).
Output: $F$ (Frequent itemsets in *CTDB*)

1: $F_1 \leftarrow \{\{i\} \mid i \in$ items in the head of *CTDB*$\}$
2: **for** each $X,Y \in F_1$, **and** $X < Y$ **do**
3: $C_2 \leftarrow C_2 \cup \{X \cup Y\}$
4: **end for**
5: $k \leftarrow 2$
6: **while** $C_k \neq \emptyset$ **do**
7: **for** each transaction $T$ in the body of $CTDB$ **do**
8: **for** each candidate itemsets $X \in C_k$ **do**
9: **if** $X \subseteq T$ **then**
10: count[$X$] ← count[$X$] + count[$T$]
11: **end if**
12: **end for**
13: **end for**
14: $F_k \leftarrow \{X \mid$ support[$X$] $\geq$ *min_sup*$\}$
15: **for** each $X,Y \in F_k$, $X[i]=Y[i]$ for $1 \leq i \leq k$ **and** $X[k] < Y[k]$ **do**
16: $L \leftarrow X \cup \{Y[k]\}$
17: **if** $\forall J \subset L, |J| = k : J \in F_k$ **then**
18: $C_{k+1} \leftarrow C_{k+1} \cup L$
19: **end if**
20: **end for**
21: $k \leftarrow k + 1$
22: **end while**
23: **return** $F = \bigcup_k F_k$

There are two essential differences between this method and the Apriori algorithm:

1) The CT-Apriori algorithm skips the initial scan of database in the Apriori algorithm by reading the head part of the *compact transaction database* and inserting the frequent 1-itemsets into $F_1$. Then candidate 2-itemset $C_2$ is generated from $F_1$ directly, as shown in step 1 - 4 in the above algorithm.
2) In the Apriori algorithm, to count the supports of all candidate *k*-itemsets, the original database is scanned, during which each transaction can add at most one count to a candidate *k*-itemset. In contrast, in CT-Apriori, as shown in step 10, these counts are incremented by the occurrence count of that transaction stored in the body of the compact transaction database, which is, in most of the time, greater than 1.

## V. EXPERIMENTAL STUDIES

In this section, we report our experimental results on the generation of compact transaction databases as well as the performance of *CT-Apriori* using the compact transaction databases in comparison with the classic Apriori algorithm using traditional transaction databases.

### *A. Environment of experiments*

All the experiments are performed on a double-processor server, which has 2 Intel Xeon 2.4G CPU and 2G main memory, running on Linux with kernel version 2.4.26. All the programs are written in Sun Java 1.4.2. The algorithms are tested on two types of data sets: synthetic data, which mimic market basket data, and anonymous web data, which belong to the domain of web log databases. To evaluate the performance of the algorithms over a large range of data characteristics,

TABLE III

PARAMETERS USED IN THE SYNTHETIC DATA GENERATION PROGRAM

| Parameter | Meaning |
|---|---|
| $\|D\|$ | Total number of transactions |
| $\|T\|$ | Average size of transactions |
| $\|I\|$ | Average size of maximal potentially frequent itemsets |
| $\|L\|$ | Number of maximal potentially frequent itemsets |
| $N$ | Total number of items |

TABLE IV

PARAMETERS SETTINGS OF SYNTHETIC DATA SETS

| Transaction Database | $\|T\|$ | $\|I\|$ | $\|D\|$ |
|---|---|---|---|
| T5.I3.D100k | 5 | 3 | 100k |
| T10.I5.D100k | 10 | 5 | 100k |
| T20.I10.D100k | 20 | 10 | 100k |
| T10.I5.D200k | 10 | 5 | 200k |
| T15.I10.D200k | 15 | 10 | 200k |
| T20.I15.D200k | 20 | 15 | 200k |

we have tested the programs on various data sets and only the results on some typical data sets are reported here. Moreover, these two algorithms generate exactly the same set of frequent patterns for the same input parameters.

The synthetic data sets that we used in our experiments were generated using the procedure described in [4]. These transactions mimic the actual transactions in a retail environment. The transaction generator takes the parameters shown in Table III.

Each synthetic data set is named after these parameters. For example, the data set T10.I5.D20K uses the parameters $|T| = 10$, $|I| = 5$, and $|D| = 20000$. For all the experiments, we generate data sets by setting $N = 1000$ and $|L| = 2000$ since these are the standard parameters used in [4]. We chose 4 values for $|T|$: 5, 10, 15 and 20. We also chose 4 values for $|I|$: 3, 5, 10 and 15. And the number of transactions are set to 100,000 and 200,000. Table IV summarizes the data set parameter settings.

We report experimental results on two real-world data sets. One of them was obtained from http://kdd.ics.uci.edu/databases/msweb/msweb.html. It was created by sampling and processing the web logs of Microsoft. The data records the use of www.microsoft.com by 38000 anonymous, randomly-selected users. For each user, the data lists all the areas of the web site that user visited in a one week time frame. The data set contains 32711 instances (transactions) with 294 attributes (items); each attribute is an area of the www.microsoft.com web site.

The other data set was first used in [9] to discovery interesting association rules from Livelink [1] web log data. This data set is not publicly available for proprietary reasons. The log files contain Livelink access data for a period of two months (April and May 2002). The size of the raw data is 7GB. The data describe more than 3,000,000 requests made to a Livelink server from around 5,000 users. Each request corresponds to an entry in the log files. The detail of data preprocessing, which transformed the raw log data into the data that can be used for learning association rules, was described in [9].

The resulting session file used in our experiment was derived from the 10-minute time-out session identification method. The total number of sessions (transactions) in the data set is 30,586 and the total number of objects [2] (items) is 38,679.

### B. Generation of compact databases

To evaluate the effectiveness of compact transaction databases, we compared the compact transaction database with the original database in terms of the size of the databases and the number of transactions in the databases. The compression results are summarized in Table V.

As the experimental data show, the proposed approach guarantees a good compression in the size of the original transaction database with an average rate of 16.2%, and an excellent compression in the number of transactions with an average rate of 28.0%.

In the best case, a compression down to 63.1% of the size of the original transaction database and 34.3% of the number of transactions can be achieved in the Microsoft web data. Moreover, as can be seen, much higher compression rates are achieved in real-world data sets, which indicates that the compact transaction database provides more effective data compression in real-world applications.

### C. Evaluation of efficiency

To assess the efficiency of our proposed approach, we performed several experiments to compare the relative performance of the Apriori and CT-Apriori algorithms. Fig. 2 and Fig. 3 illustrate the corresponding execution times for the two algorithms on two different types of databases with various support thresholds from 2% down to 0.25%.

From these performance curves, it can be easily observed that CT-Apriori performs better in all situations. As the support threshold decreases, the performance difference between the two algorithms becomes prominent in almost all the cases, showing that the smaller the support threshold is, the more advantageous CT-Apriori is over Apriori. The performance gaps between these two methods are even more substantial on the T15.I10.D200K and Microsoft data sets, as shown in Fig. 2 and Fig. 3 respectively.

It is easy to see why this is the case. First, Apriori needs one complete database scan to find candidate 1-itemsets, while CT-Apriori can generate them from the head part of compact transaction database. Even though it takes time to construct a compact transaction database, the resultant compact transaction database can be used multiple times for mining patterns with different support thresholds. Second, when the support threshold gets lower, these two algorithms have to scan databases more times to discover the complete set of frequent patterns. For instance, the Apriori algorithm requires 18 passes

[1]Livelink is a web-based product of Open Text Corporation.

[2]An object could be a document (such as a PDF file), a project description, a task description, a news group message, a picture and so on [9].

TABLE V

GENERATION OF COMPACT TRANSACTION DATABASES

| Transaction Databases | Size of Databases | | | Number of Transactions | | |
|---|---|---|---|---|---|---|
| | Original (Kb) | Compact (Kb) | Compression Ratio (%) | Original | Compact | Compression Ratio (%) |
| T5.I3.D100K | 2,583 | 2,238 | 13.4 | 100,000 | 67,859 | 32.1 |
| T10.I5.D100K | 4,541 | 4,349 | 4.2 | 100,000 | 83,095 | 16.9 |
| T20.I10.D100K | 8,451 | 8,227 | 3.7 | 100,000 | 89,023 | 11.0 |
| T10.I5.D200K | 9,000 | 8,644 | 4.0 | 200,000 | 166,161 | 16.9 |
| T15.I10.D200K | 13,358 | 11,108 | 16.8 | 200,000 | 142,863 | 28.6 |
| T20.I15.D200K | 17,913 | 14,155 | 21.0 | 200,000 | 151,306 | 24.3 |
| | Average Compression Ratio | | 10.4 | Average Compression Ratio | | 21.6 |
| Microsoft Web Data | 545 | 344 | 36.9 | 32,711 | 11,233 | 65.7 |
| LiveLink Web Data | 3,275 | 2,262 | 30.9 | 30,586 | 21,921 | 28.3 |
| | Average Compression Ratio | | 33.9 | Average Compression Ratio | | 47.0 |
| Compression Ratio | 16.2% | | | 28.0% | | |

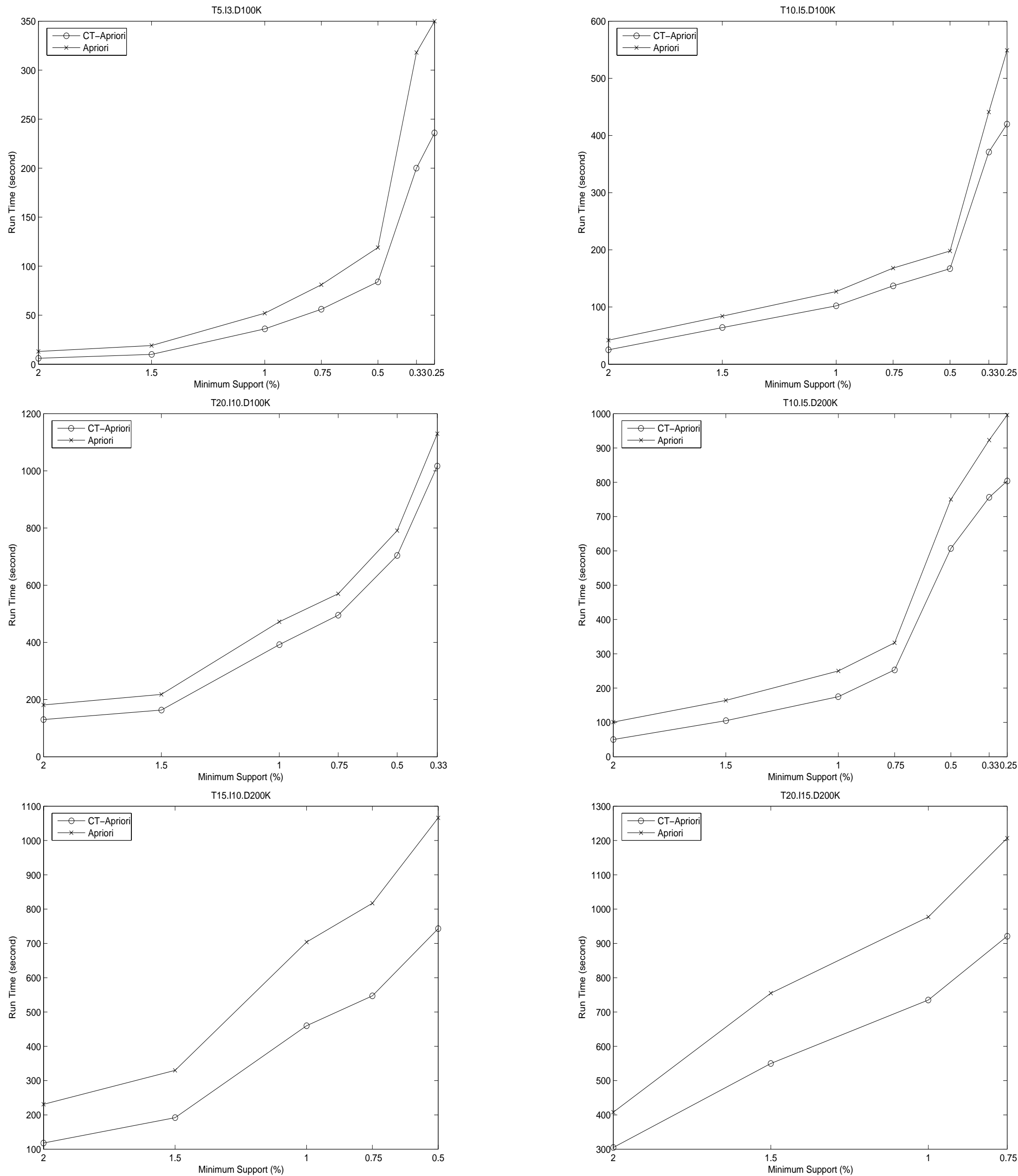


Fig. 2. Execution times on synthetic databases.

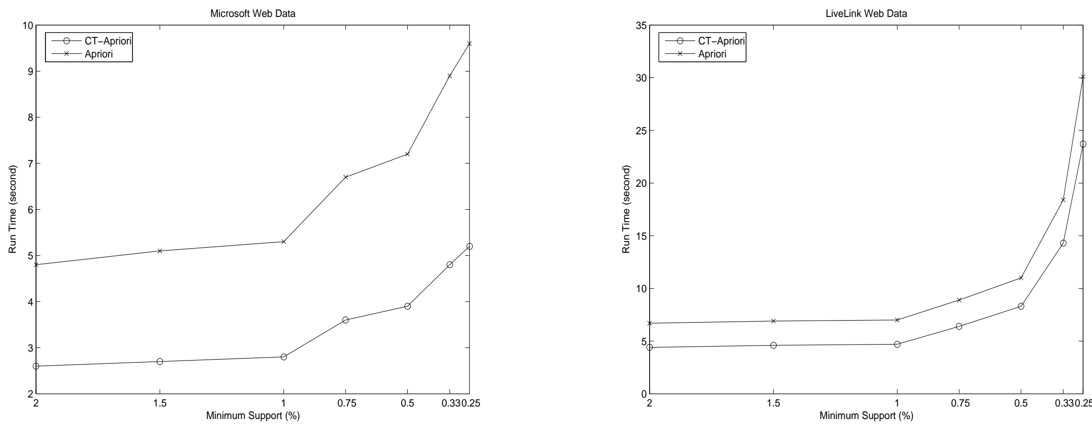


Fig. 3. Execution times on real-world databases.

over the database T15.I10.D200K when the support threshold is set to 0.25%.

As shown in the above section, the number of transactions in a compact transaction database is always smaller than that in its corresponding original database, which results in time saving in each scan of the database. The time saved in each individual scan by CT-Apriori collectively results in a significant saving in the total amount of I/O time of the algorithm.

## VI. Related Work

Data compression is an effective method for reducing storage space and saving network bandwidth. A large number of compression schemes have been developed based on character encoding or on detection of repetitive strings, and comprehensive surveys of compression methods and schemes are given in [6], [10], [16].

There are two fundamentally different types of data compression: lossless and lossy. As we have mentioned at the beginning of our experimental evaluations, the set of frequent patterns generated from an original transaction database and its corresponding compact transaction database are identical with the same input parameters, therefore, the compact transaction database approach proposed in this paper is lossless.

The major difference of our approach from others is that our main purpose of compression is to reduce the I/O time when mining patterns from a transaction database. Our compact transaction database can be further compressed by any existing lossless data compression technique for storage and network transmission purposes.

Mining frequent patterns is a fundamental step in data mining and considerable research effort has been devoted to this problem since its initial formulation. A number of data compression strategies and data structures, such as prefix-tree (or trie) [2], [5], [7] and FP-tree [8], have been devised to optimize the candidate generation and the support counting process in frequent patterns mining.

The concept of prefix-tree is based on the set enumeration tree framework [14] to enable itemsets to be located quickly.

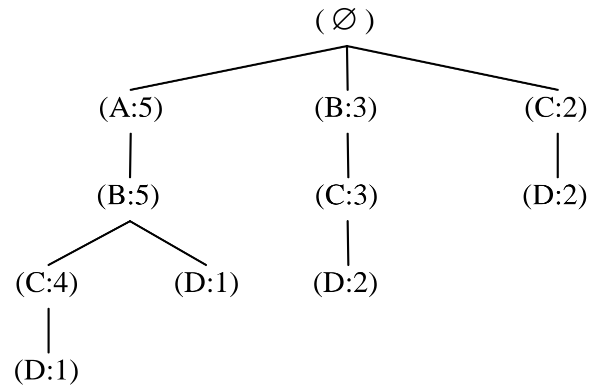


Fig. 4. Prefix-tree for the database *TDB* in Table I.

Fig. 4 illustrates the prefix-tree for the example transaction database in Table I. The root node of the tree corresponds to the empty itemset. Each other node in the tree represents an itemset consisting of the node element and all the elements on nodes in the path (prefix) from the root. For example, the path ($\emptyset$)-(B:3)-(C:3)-(D:3) in Fig. 4 represents the itemset $\{$B, C, D$\}$ with support of 3.

It can be seen that the set of paths from the root to the different nodes of the tree represent all possible subsets of items that could be present in any transaction. Compression is achieved by building the tree in such a way that if an itemset shares a prefix with an itemset already in the tree, the new itemset will share a prefix of the branch representing that itemset. Further compression can also be achieved by storing only frequent items in the tree.

The FP-growth method proposed in [8] uses another compact data structure, FP-tree (Frequent Pattern tree), to represent the conditional databases. FP-tree is a combination of prefix-tree structure and node-links, as shown in Fig. 5.

All frequent items and their support counts are found by the first scan of database, and are then inserted into the header table of FP-tree in frequency descending order. To facilitate tree traversal, the entry for an item in the header table also contains the head of a list that links all the corresponding

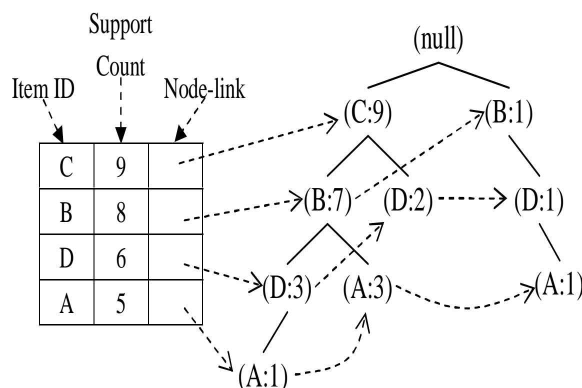


Fig. 5. FP-tree for the database *TDB* in Table I.

nodes of the FP-tree.

In the next scan, the set of sorted (frequency descending order) frequent items in each transaction are inserted into a prefix-tree as a branch. The root node of the tree is labeled with "*null*", every other node in the FP-tree additionally stores a counter which keeps track of the number of itemsets that share that node. When the frequent items are sorted in their frequency descending order, there are better chances that more prefixes can be shared, thus the FP-tree representation of the database can be kept as small as possible.

The *compact transaction database* and *CT-tree* data structure introduced in the previous sections are very different from above approaches. First of all, the prefix-tree and FP-tree data structure are constructed in the main memory to optimize the frequent pattern mining process, whereas *CT-tree* is designed to generate *compact transaction database* and store it to disk for efficient frequent pattern mining and other mining process, in which compact database can save storage space and reduce mining time.

In addition, the counter associated with each node in the prefix-tree and FP-tree stores the number of transaction containing the itemset represented by the path from the root to the node. However, each path from every node of the *CT-tree* to the root represents a unique transaction, and the associated counter records the number of occurrences of this transaction in the original transaction database.

Moreover, for a given transaction database, the number of nodes and node-links in FP-tree will change with different minimum support threshold specified by the user. But there is only one unchanged *CT-tree* for every transaction database. And the FP-tree structure can be constructed from *a compact transaction database* more efficiently in only one database scan, since the head part of a *compact transaction database* lists all items in frequency descending order, and the body part stores all ordered transactions associated with their occurrence counts.

## VII. Conclusions

We have proposed an innovative approach to generating compact transaction databases for efficient frequent pattern mining. The effectiveness and efficiency of our approach are verified by the experimental results on both synthetic and real-world data sets. It can not only reduce the number of transactions in the original databases and save storage space, but also greatly reduce the I/O time required by database scans and improve the efficiency of the mining process.

We have assumed in this paper that the CT-tree data structure will fit into main memory. However, this assumption will not apply for very large databases. In that case, we plan to partition original databases into several small parts until the corresponding CT-tree can be fit in the available memory. This work is currently in progress.

## VIII. Acknowledgments

This research is supported by Communications and Information Technology Ontario (CITO) and Natural Sciences of Engineering Research Council of Canada (NSERC).

## References

[1] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proceedings of ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000.

[2] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.

[3] R. Agarwal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., USA, May 1993.

[4] R. Agarwal and R. Strikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[5] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the International ACM SIGMOD Conference*, pages 85–93, May 1998.

[6] T. Bell, I. H. Witten and J. G. Cleary. Modelling for Text Compression. In *ACM Computing Surveys*, 21, 4 (December 1989), 557.

[7] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the International ACM SIGMOD Conference*, pages 255–264, Tucson, Arizona, USA, May 1997.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 1–12, Dallas, TX, May 2000.

[9] X. Huang, A. An, N. Cercone, and G. Promhouse. Discovery of interesting association rules from livelink web log data. In *Proceedings of IEEE International Conference on Data Mining*, Maebashi City, Japan, 2002.

[10] D. A. Lelewer and D. S. Hirschberg. Data Compression. In *ACM Computing Surveys*, 19, 3 (September 1987), 261.

[11] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent itemsets by opportunistic projection. In *Proceedings of ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Canada, July 2002.

[12] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192, July 1994.

[13] J. S. Park, M. S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995.

[14] R. Rymon. Search through systematic set enumeration. In *Proceedings of 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 539-550, 1992.

[15] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[16] J. A. Storer. Data Compression: Methods and Theory. In *Computer Science Press*, New York, NY, 1988.