Rapid Association Rule Mining

Amitabha Das
Nanyang Technological University
Nanyang Avenue
Singapore 639798
asadas@ntu.edu.sg

Wee-Keong Ng
Nanyang Technological University
Nanyang Avenue
Singapore 639798
wkn@acm.org

Yew-Kwong Woon
Nanyang Technological University
Nanyang Avenue
Singapore 639798
davidwyk@singnet.com.sg

ABSTRACT

Association rule mining is a well-researched area where many algorithms have been proposed to improve the speed of mining. In this paper, we propose an innovative algorithm called Rapid Association ***Rule*** Mining (RARM) to once again break this speed barrier. It uses a versatile tree structure known as the *Support-Ordered Trie* Ztemset (SOTrieIT) structure to hold pre-processed transactional data. This allows RARM to generate large 1-itemsets and 2-itemsets quickly without scanning the database and without candidate 2-itemset generation. It achieves significant speed-ups because the main bottleneck in association rule mining using the Apriori property is the generation of candidate 2-itemsets. RARM has been compared with the classical mining algorithm Apriori and it is found that it outperforms Apriori by up to two orders of magnitude (100 times), much more than what recent mining algorithms are able to achieve.

Categories and Subject Descriptors

H.2.8 [Database Applications]: Data Mining

General Terms

Data Mining

Keywords

Association rule mining, electronic commerce, index data structure

1. INTRODUCTION

Since the dawn of the Internet era in 1994, electronic commerce is growing at, such an astonishing rate that companies around the world race to move their business online in order to position themselves for the near future when the Internet would dominate worldwide trading.

In a Forrester Research article, Global *eCommerce Approaches Hypergrowth* [10], it is estimated that global Internet trade would reach US$6.8 trillion in 2004, amounting to 8.6% of the global sales of goods and services. Hence, it is obvious as well as inevitable that companies will realign their business strategies with the Internet.

One of the most important areas that needs addressing is ***Customer*** Relationship Management (CRM) [4] which is now a US$11.5 billion market. In a fast-changing environment like the Internet, data is abundant in the form of transactional data. In electronic commerce, thousands and millions of transactions can easily take place in a single day. Embedded within these data is valuable hidden knowledge about the behavior of customers that could be unraveled with data mining techniques. Hence, the importance of data mining in electronic commerce, particularly in CRM, is undisputed.

Market basket analysis [3] or the mining of association rules is the most practical and beneficial data mining technique to be used in electronic commerce. This is particularly true in CRM because it focuses on the buying habits of customers and helps to decide which products to be bundled together. It can be employed in a wide variety of applications such as marketing, promotional bundling of products and planning an optimal virtual store layout.

In this paper, a new algorithm called ***Rapid*** Association ***Rule*** Mining (RARM) is proposed to further push the speed barrier so that association rule mining can be performed more efficiently in electronic commerce. To achieve large speed-ups even at low support, thresholds, RARM constructs a new data structure called Support-Ordered *Trie* Ztemset (SOTrieIT). This trie-like tree structure stores the support counts of all 1-itemsets and 2-itemsets in the database. All transactions that arrive are pre-processed; all 1-itemsets and 2-itemsets are extracted from each transaction. The extracted information is used to update the SOTrieIT. This structure is then sorted according to the support counts of each node in descending order.

RARM uses SOTrieIT to quickly discover large 1-itemsets and 2-itemsets without scanning the database. The need to generate candidate 1-itemsets and 2-itemsets constitutes the main bottleneck in large itemset generation, as observed in [9]. Therefore, by eliminating this step, RARM achieves significant speed-ups even though subsequently, it also applies the Apriori algorithm to obtain larger-sized itemsets.

Experiments have been conducted to study the performance of RARM and compare it against Apriori [2]. RARM is found to outperform Apriori by up to two orders of magnitude, far exceeding what the latest algorithms can achieve.

The rest, of the paper is organized as follows. The next

CIKM'01, November 5-10, 2001, Atlanta, Georgia, USA.

section reviews related work. Section 3 gives a description of the problem while Section 4 presents the new tree structure. Section 5 describes the RARM algorithm. Time and space complexity of the new structure will be examined in Section 6. Performance evaluation is discussed in Section 7. Finally, the paper is concluded and recommendations for future work are made in Section 8.

2. RELATED WORK

Since the introduction of the Apriori algorithm [2] in 1994, there has been sustained interest in discovering new association rule mining algorithms that could perform more efficiently. Incremental algorithms are presented in [5, 6, 11] and algorithms which support dynamic support thresholds are introduced in [1, 8]. However, in this paper, we are only interested in breaking the speed barrier. Hence, in this section, we will only look at two influential algorithms which achieve impressive speed-ups against Apriori.

The ***Direct Hashing*** and Pruning (DHP) algorithm [9] is the next widely used algorithm for the efficient mining of association rules. It is built on top of Apriori but it employs a hashing technique to reduce the size of candidate itemsets and database. This amounts to significant speed-ups because the dominating factor in the generation of large itemsets is the size of the candidate itemscts, particularly that of the candidate 2-itemsets. Jong et al. divides the DHP algorithm into the following three main parts:

1. A set of large 1-itemsets is obtained and a hash table for 2-itemsets is built.

2. Candidate k-itemsets are generated based on the hash table built from the previous iteration. The hash table serves as a filtering mechanism; DHP only adds an itemset to the set of candidate k-itemsets if it is hashed to an entry whose value is larger or equal to the minimum support count needed. Large k-itemsets are determined and a hash table for the candidate $(k+1)$-itemsets is built as well.

3. Same as part 2 except that hash tables are not used. This is because part 3 is only used in later iterations where significantly lesser candidate itemsets are generated.

It is obvious that DHP incurs additional overheads due to the need to do hashing and to maintain a hash table. Therefore, after experiments are performed, it is concluded that the hashing technique should only be applied during the generation of candidate 2-itemsets and not for the generation of large candidate itemsets. This allows DHP to achieve speed-ups of up to three times against Apriori.

Recently, Han et al. proposed the Frequent Pattern ***Growth*** (FP-growth) [7] algorithm which achieves impressive results compared to Apriori. It completely eliminates the candidate generation bottleneck by using a new tree structure called Frequent Pattern *Tree* (FP-tree) which stores critical itemset information. This compact structure also removes the need for database scans and it is constructed using only two scans. In the first database scan, large 1-itemsets L_1 are obtained and sorted in support descending order. In the second scan, items in the transactions are first sorted according to the order of L_1. These sorted items are used to construct the FP-tree. Refer to [7] for the construction details and completeness proof of the FP-tree.

FP-growth then proceeds to recursively mine FP-trees of decreasing size to generate large itemsets without candidate generation and database scans. It does so by examining all the ***conditional*** pattern bases of the FP-tree, which consists of the set of large itemsets occurring with the suffix pattern. Conditional FP-trees are constructed from these conditional pattern bases and mining is carried out recursively with such trees to discover large itemsets of various sizes. However, the construction and use of the FP-trees are complex and cause the performance of FP-growth to be on par with Apriori at support thresholds of 3% and above. It only shows significant speed-ups at support thresholds of 1.5% and below.

3. PROBLEM DESCRIPTION

The problem of mining association rules can be formally described as follows: Let I = $\{a_1, a_2, ..., a_m\}$ be a set of literals called items. Let D be a database of transactions, where each transaction T contains a set of items such that $T \subseteq I$. An *itemset* is a set of items and a k-itemset is an itemset that contains exactly k items. For a given *itemset* $X \subseteq I$ and a given transaction T, T contains X if and only if $X \subseteq T$. Let σ_x be the ***support count*** of an itemset X, which is the number of transactions in D that contain X. Let s be the ***support threshold*** and $|D|$ be the number of transactions in D. An itemset X is large or frequent if $\sigma_X \geq |D| \times s\%$. An **association** rule is an implication of the form $X \Longrightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = 0$. The association rule $X \Longrightarrow Y$ holds in the database D with ***confidence*** $c\%$ if no less than $c\%$ of the transactions in D that contain X also contain Y. The association rule $X \Longrightarrow Y$ has ***support*** $s\%$ in D if $\sigma_{X \cup Y} = |D| \times s\%$.

For a given pair of confidence and support thresholds, the problem of mining association rules is to discover all rules that have confidence and support greater than the corresponding thresholds. For example, in a computer hardware shop, the association rule Scanner $\Longrightarrow$ Printer means that whenever customers buy scanners, they also buy printers $c\%$ of the time and this trend occurs $s\%$ of the time.

This problem can be decomposed into two sub-problems as discussed in [2]:

1. Finding of large itemsets
2. Generation of association rules from large itemsets

Most researchers addressed the first sub-problem only because it is more computationally expensive. Moreover, once the large itemsets are identified, the corresponding association rules can be generated in a simple and straightforward manner. Hence, only the first sub-problem will be addressed in this paper. For a discussion of optimized algorithms for the generation of association rules, more information can be found at [1].

4. DATA STRUCTURE

This section first describes a general trie-like structure, followed by an enhanced version of it and finally the **most** optimized version. A **trie** is a tree structure whose organization is based on a *key space* decomposition. In key space decomposition, the key range is equally subdivided and the

Table 1: An example transaction database with N = 4.

TID	Items
100	AC
200	BC
300	ABC
400	$ABCD$

splitting position within the key range for each node is predefined. An *alphabet trie* is a trie used to store a dictionary of words. The following data structures will be based on the alphabet trie.

4.1 A Complete TrieIT

Given a database D of transactions, we store it as a forest of lexicographically-ordered tree nodes known as *Trie Itemset* (TrieIT) so that the first sub-problem of association rule mining-finding large itemsets-can be done efficiently. Let the set of items $I = \{a_1, a_2, .., a_N\}$ in the database be ordered so that for any two items $a_i \in I, a_j \in I$ $(1 \leqslant i, j \leqslant N)$, $a_i \prec a_j$ if and only if $i < j$. Likewise, every transaction $T \in D$ is also ordered with respect to the ordering in I.

DEFINITION 1. *A* complete *TrieIT is* a *set of tree nodes such that every tree* node w is a *2-tuple* $\langle w_l, w_c \rangle$ *where* $w_l \in I$ is *the* label *of the* node and w_c is a support *count.* Since *every* tree *node* corresponds *to some item* $a_i \in I$, *for brevity, we also* use w_i *to* refer *to a tree* node *that* corresponds *to* $a_i \in I$. *The following* conditions *hold:*

1. *Let,* $C(w_i)$ *be the ordered set of children* nodes *of* w_i. *If* $C(w_i) \neq 0$, *then* $C(w_i) \subseteq \{w_{i+1}, w_{i+2}, .. , w_N\}$.
2. *Given a node* w_i, *let* $w_k, w_{k+1}, \ldots, w_{i-1} (1 \leqslant k \leqslant i-1)$ *be the set of nodes* on *the* path from *the* root *to the* parent *of* w_i, *then* w_c *(the* count *of* w_i*) is a count of the itemset* $\{a_k, a_{k+1}, \ldots, a_i\}$ in *the database.* Hence, *the support count of* any k*-itemset can be obtained by following* a *set of* nodes *to* a *depth of k.*

Each complete *TrieIT* W_i corresponds *to* some $a_i \in I$ *such that the* root node has *label* a_i. *Then D is stored as* a *set of* complete *TrieITs denoted by W* where $W \subseteq \{W_1, W_2, \ldots, W_N\}$. ∎

In order to improve the efficiency of finding large itemsets using complete TrieITs, every transaction $T \in D$ is inserted into W as follows: Every ordered itemset $X \in P(T)$ *(P(T)* is the powerset of T) increases the count by one of node w_j of TrieIT W_i where a_i and a_j are the first and last items of X respectively. If TrieIT W_i or node w_j does not exist, it is created with an initial count of one. Thus, each $T \in D$ updates the support counts of all its sub-itemsets (from its powerset) in the corresponding TrieITs of W. Hence, no database scanning is required subsequently as the support counts are already stored in the TrieITs. The following example illustrates the concept of TrieITs and how transactions update the TrieITs.

4.1.1 Example

Figure 1(a) shows the TrieITs after the transactions 100 to 300 in Table 1 have been inserted into the tree. The numbers in brackets are the support counts. The special node *ROOT* simply contains pointers to the root nodes of the TrieITs. To understand how the TrieITs are updated, let us see what happens when transaction 400 is inserted.

From transaction 400, the following itemsets are extracted from it:

- 1-itemsets: *{A}, {B}, {C}, {D}*
- 2-itemsets: $\{A, B\}$, *{A, C}, {A, D}, {B, C}, {B, D}*, $\{C, D\}$
- 3-itemsets: *{A, B, C}, {A, B, D}, {A, C, D}, {B, C, D}*
- 4-itemsets: *{A, B, C, D}*

They are used to increment the support counts of their corresponding nodes in the TrieITs by one. To locate the node that corresponds to an itemset $\{a_k, a_{k+1}, .. , a_i\}$, simply trace the path $\{a_k, a_{k+1}, \ldots, a_i\}$ from the ROOT node. If a node does not exist, create a new node and initialized it with a support count of 1. Here, we need to create seven nodes with the label D to represent itemsets ending with the item D. The resultant set of complete TrieITs is shown in Figure 1(b). This set of TrieITs contains the maximum number of tree nodes because all N unique items appear in the database.

To obtain the support count of an itemset, simply locate its path along the tree nodes. The node representing the last item in the itemset would contain the support count for the itemset. For example, to find the support count of itemset *{A, B, D}*, locate the path from the ROOT node to the node containing the last item D. Thus, from Figure 1(b), the support count of itemset *{A, B, D}* is 1. This search is efficient because the nodes are sorted lexicographically. However, the set of complete TrieITs require a huge amount of memory space which scales up exponentially with N.

4.2 Support-Ordered Trie Itemset

This approach builds on the ideas presented in the paper on DHP [9]. In that paper, it is discovered that generation of large 2-itemsets is the main performance bottleneck. Using a hash table, DHP is able to improve performance significantly by reducing the size of the candidate 2-itemsets.

Similarly, this approach seeks to find a data structure that allows for quick generation of large 2-itemsets without the heavy processing and memory requirements of the previous two structures. The solution is a 2-level support-ordered tree which is called a *SOTrieIT* (Support-Ordered Trie Itemset). It is unnecessary and inefficient to go beyond two levels because the memory requirements will far outweigh the computation savings. This will be discussed in greater details in section 7.

DEFINITION 2. *A SOTrieIT* is a complete *TrieIT with* a *depth of* 1; i.e., *it* consists only *of* a root node w_i and some *child* nodes. *Moreover,* all nodes in *the* forest *of SOTrieIT* are sorted according *to their* support counts in descending order from *the left.* ∎

In constructing the set of SOTrieITs Y from a database D, only 1-itemsets and 2-itemsets $X \in P(T)$ of each transaction $T \in D$ are used to update the support counts in Y. In other words, the set of SOTrieITs only keeps a record of all 1-itemsets and 2-itemsets contained in a transaction. The first-level nodes represent 1-itemsets while second-level nodes represent 2-itemsets. Henceforth, we shall use the term SOTrieIT to denote a set of SOTrieITs.

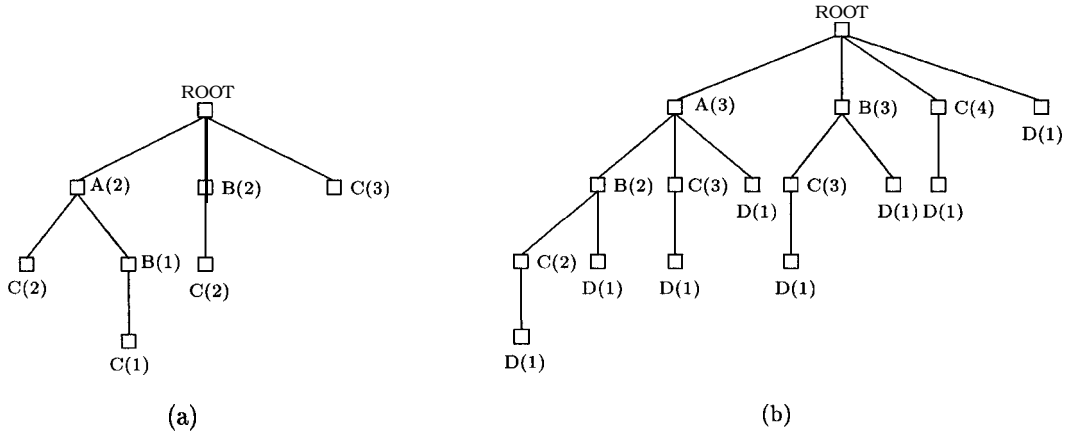

Figure 1: Resultant Complete TrieIT.

By keeping track of the support counts of all 1-itemsets and 2-itemsets, SOTrieIT allows both large 1-itemsets and 2-itemsets to be found very quickly. This is because there is no need to scan the database which could be very large. Instead, only the small SOTrieIT is scanned. Moreover, as the SOTrieIT is sorted according to the support counts of the itemsets, only part of the structure needs to be scanned. This feature is elaborated below in the discussion of the RARM algorithm. Finally, the amount of pre-processing work and memory needed are greatly reduced.

4.2.1 Example

Figure 2 represents the fully constructed SOTrieIT for the example transaction database in Table 1. To illustrate how nodes are created, let us examine what happens when a new transaction arrives. Note that only 1-itemsets and 2-itemsets are extracted from the transactions. The nodes arc created in a similar manner as in the complete TrieITs. When both transaction 100 and 200 arrive, the nodes created are the same as those created for the complete TrieITs as shown in Figures 2(a) and **2(b)** because only 1-itemsets and 2-itemsets are extracted in both cases. However, notice that in Figure 2(b), the node w_C under the ROOT node comes before the node w_A. This is because the nodes are sorted according to their support counts and w_C has a higher support count than w_A.

When transaction 300 arrives, the following itemsets are extracted: $\{A\}, \{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}$

Unlike the situation for the complete TrieITs, the itemset {A, B, C} is not extracted from the transaction. Figure 2(c) shows the resultant SOTrieIT when this transaction is processed. When transaction 400 arrives, the following itemsets are extracted: $\{A\}, \{B\}, \{C\}, \{D\}, \{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}$

The SOTrieITs are updated in a similar fashion for transaction 400 as seen in Figure 2(d). The final resultant SOTrieIT resembles the complete set of TrieITs except that it consists of only three levels of nodes and that sibling nodes are sorted according to their support counts.

4.3 Correctness

We need to show that with a SOTrieIT, the support counts of all 1-itemsets and 2-itemsets can be correctly obtained without scanning the database. Let T_s be a transaction of size s and $T_s = \{b_1, b_2, \ldots, b_s\}$. The 1-itemsets that are extracted and used to build W are $\{b_1\}, \{b_2\}, \ldots, \{b_s\}$ and the 2-itemsets extracted are $\{b_x, b_y\}$ where $0 < x < s$ and $x < y \leqslant s$. These itemsets update counts in the SOTrieITs accordingly. Every itemset increments or decrements the support count of its corresponding tree node depending on whether the transaction is added or deleted.

At any point in time, W contains all the support counts of all 1-itemsets and 2-itemsets that appear in all the transactions. Hence, there is no longer any need to scan the database during the generation of large 1-itemsets and 2-itemsets.

5. ALGORITHM RARM

In this section, the RARM algorithm that uses a SOTrieIT is described.

5.1 Pre-processing

Figure 3 **shows** the pre-processing steps taken whenever a transaction arrives. For every transaction that arrives, l-itemsets and 2-itemsets are first extracted from it. For each itemset, the SOTrieIT, denoted by Y, will be traversed in order to locate the node that stores its support count. Support counts of 1-itemsets and 2-itemsets are stored in first-level and second-level nodes respectively. Therefore, this traversal requires at most two redirections which makes it very fast. Y will then be sorted level-wise from left to right according to the support counts of the nodes in descending order. If such a node does not exist, it will be created and inserted into Y accordingly. Similarly, Y is then sorted after such an insertion.

5.2 Mining of large itemsets

Figure 4 shows the steps taken when the mining process is started. The SOTrieIT, Y, is first traversed to discover large l-itemsets and 2-itemsets. In our approach, depth-first search is used, starting from the leftmost first-level node. As Y is sorted according to support counts, the traversal can be stopped the moment a node is found not to satisfy the minimum support threshold. After large 1-itemsets and 2-itemsets are found, the algorithm proceeds to discover other larger itemsets using the Apriori algorithm. Experiments in

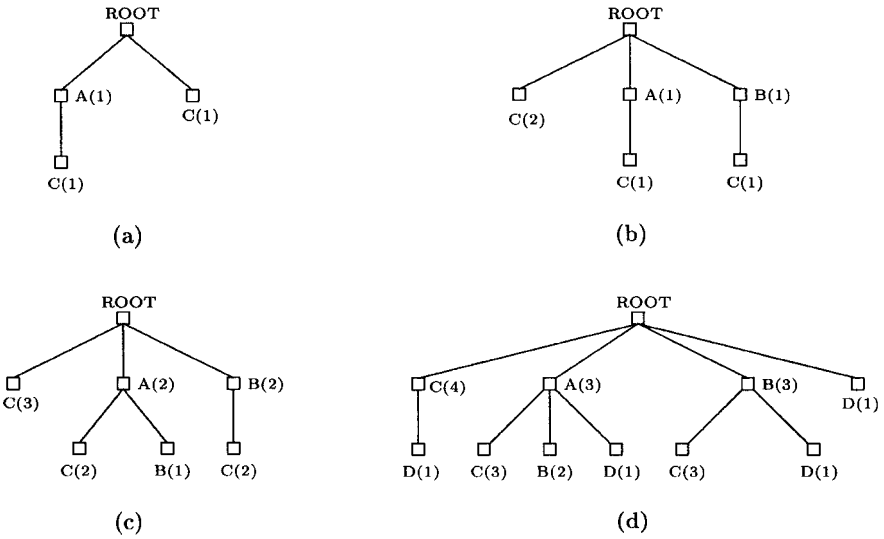

Figure 2: Resultant SOTrieIT.

```
1  Let Y be a set of SOTrieITs
2  for (k = 1; k ≤ 2; k++) do begin
3     Obtain all k-itemsets of the transaction and
      store them in C_k
4     foreach itemset X ∈ C_k do begin
5        Traverse Y to locate nodes along the path that
         represents X
6        if such a set of nodes exists in Y then
7           Increment the support count of the leaf node
8           Sort updated node among siblings according
            to its new support count in descending order
9        else
10          Create a new set of nodes with support
            counts of 1 that represent a path to X
11          Insert them into Y according to their support
            counts in descending order from the left
12       endif
13    endfor
14 endfor
```

Figure 3: Pre-processing Algorithm.

```
1  Let N_q^p be the q^th child node of parent node p.
2  Let NC_p be the number of child nodes under node p
3  Let I_n be the itemset represented by node n.
4  for (x=1; x ≤ NC_ROOT; x++) do begin
5     Let X = N_x^ROOT.
6     if σx ≥ |D| x s% then begin
7        Add Ix to L1.
8        for (y=1; y ≤ NC_X; y++) do begin
9           if σ_{N_y^X} ≥ |D| x s% then
10             Add I_{N_y^X} to L2.
11       endfor
12    endif
13 endfor
14 Run Apriori from its third iteration to find the rest
   of the large itemsets from 3-itemsets onwards.
```

Figure 4: Mining Algorithm.

Section 7 prove that the savings obtained during the generation of large 1-itemsets and 2-itemsets are enough to greatly improve its performance.

5.2.1 Example

To illustrate the mining algorithm, we use the same transaction database found in Table 1 and the SOTrieIT structure in Figure 2(d). Suppose the support threshold is set at 80%. Then the minimum support count to qualify an itemset to be large is 4. Figure 5 shows the traversal path taken in obtaining the large 1-itemsets and 2-itemsets. The bold numbers on the arrows denote the sequence with which the SOTrieIT is traversed. The RARM algorithm stops traversing the SOTrieIT at the third traversal when it encounters the item *A* which has a support count of 3. This is because all other nodes that come after first-level node *A* will have a support count of 3 or less. Therefore, there will not be any more large itemsets in the rest of the SOTrieIT. The algorithm terminates and the only large itemset is {C} and

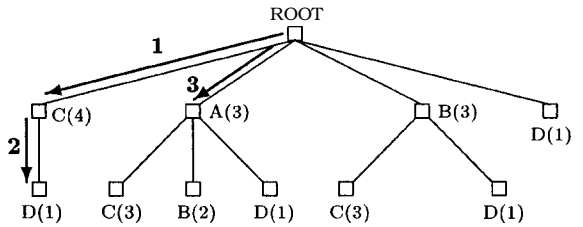

Figure 5: Traversal path of the SOTrieIT at a support threshold of 80%.

the total number of traversals is only three.

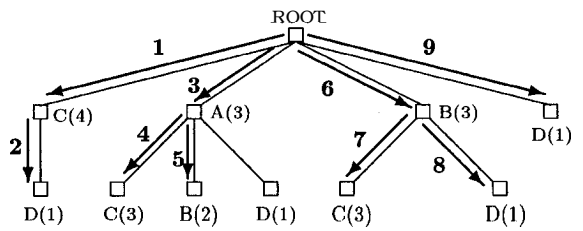

Figure 6: Traversal path of the SOTrieIT at a support threshold of 75%.

For a support threshold of 75%, the minimum support count needed is 3. Figure 6 shows the traversal path taken in obtaining the large 1-itemsets and 2-itemsets. During the generation of the first two large itemsets, the moment a first-level node with a support count lower than 3 is encountered, the rest of its siblings and subtrees are not scanned. However, when a second-level node is found not to have satisfied the minimum support count, only its subsequent siblings will be ignored. In this case, at the fifth traversal, when the node that represent itemset $\{A, B\}$ is found to have a support count of less than 3, the node that represent itemset $\{A, D\}$ is ignored. The final large 1-itemsets and 2-itemsets found are $L_1 = \{\{A\}, \{B\}, \{C\}\}$ and $L_2 = \{\{A, C\}, \{B, C\}\}$ and the total number of traversals is nine.

6. TIME AND SPACE COMPLEXITY

6.1 Pre-processing

We discuss the time and space complexities of the pre-processing phase in this section.

6.1.1 Time Complexity

The amount of time to pre-process a transaction depends on the amount of time to extract 1-itemsets and 2-itemsets from the transaction, to traverse the SOTrieIT to increment the support counts of the respective nodes, and to create new nodes in the SOTrieIT for items that are not encountered yet.

For a transaction of size s, only $({}^sC_1 + {}^sC_2)$ itemsets are pre-processed. Hence, its complexity is $O(s^2)$. For example, when transaction 400 of Table 1 arrives, the following ten itemsets are extracted: $\{A\}, \{B\}, \{C\}, \{D\}, \{A, B\}, \{A, C\}, \{A, D\}, \{B, C\}, \{B, D\}, \{C, D\}$

According to our formulation, this gives a total of $\frac{4!}{3!1!} + \frac{4!}{2!2!} = 4 + 6 = 10$ itemsets which is correct. As the SOTrieIT is only two levels deep, it takes at most two links to reach the desired node. Suppose it also takes one unit of time to move over one link and update/create the node and one unit of time to extract one itemset, it will take a maximum of $2 \times ({}^sC_1 + {}^sC_2)$ units of time to move to all the nodes required by a transaction of size s.

61.2 Space Complexity

In a database with N unique items, there will be N first-level nodes in the SOTrieIT. For each first-level node, since the SOTrieIT is created in a trie-like manner, it will contain only items that are lexicographically larger than itself. The first-level node who has the largest number of child nodes is the one which has the first position in a set of lexicons. It will have N-l child nodes. Subsequent first-level nodes will have one less child node than the previous one. Therefore, for N unique items, a maximum of only $\sum_{x=1}^{N} x$ nodes, inclusive of both first-level and second-level nodes, are needed to store the entire pre-processing information. Hence, its complexity is $O(N^2)$.

6.2 Mining of large itemsets

The next section discusses the time complexity of the mining phase as compared to that of Apriori. Space complexity will not be mentioned because this phase also involves the SOTrieIT whose space complexity is already discussed in the previous section.

6.2.1 Time Complexity

To compare the time complexity of RARM and Apriori, we shall focus only on the scanning process to obtain the support counts of itemsets. This is enough to see the vast improvement of RARM over Apriori.

For each pass of the Apriori algorithm, there is a need to scan the entire database regardless of the desired support threshold. Suppose the database is of size a and the average size of each transaction is b. Then, Apriori takes $O(ab)$ units of time to scan the database at each pass.

For the first two passes of RARM, only the SOTrieIT Y needs to be scanned. According to Section 6.1.2, Y has $R \times \sum_{x=1}^{N} x$ nodes and since each node has one link from its parent, in the worst case, it will take at most $2 \times R \times \sum_{x=1}^{N} x$ units of time to traverse the entire structure where $N \ll a$. In addition, the time needed also depends on the desired support threshold which would further reduce the number of traversals. Therefore, the average total amount of scanning time will be far less than $O(ab)$.

7. PERFORMANCE EVALUATION

This section evaluates and compares the relative performance of the Apriori and RARM algorithms by conducting experiments on a Pentium-III machine with a CPU clock rate of 1 GHz, 256 MB of main memory and running on a Windows 2000 platform. The algorithms are implemented in Java and hence large memory requirements of the Java Virtual Machine prevented us from scaling up the experiments. This issue will be tackled in future experiments.

The SOTrieIT structure is implemented using a combination of integer arrays and files. Implementation details are omitted due to the lack of space. Despite extra file I/O requirements, RARM maintains its efficient performance. Detailed analysis of the results is performed to explain the

Parameter	Meaning
N	Number of unique items
$\|D\|$	Number of transactions
$\|L\|$	Number of maximal potentially large itemsets
$\|T\|$	Average size of transactions
$\|I\|$	Average size of maximal potentially large itemsets

Figure 7: Definition of Parameters.

improvements of RARM over Apriori. Figure 7 shows the various parameters used and their meanings.

The method used for generating synthetic data is the same as the one used in [2]. To describe an experiment, we use the notation Tw.Ix.Ny.Dz where w is the average size of transactions, x is the average size of maximal potentially large itemsets, y is the number of unique items and z is the size of the database. The databases created are similar to those used in [7]. The first one is T25.I10.N1K.D10K which is denoted as D_1 while the second one is T25.I20.N10K.D100K which is denoted as D_2.

7.1 Comparison of RARM and Apriori

Figure 8 shows the execution times for the two different databases of both Apriori and RARM. From the graphs, it can be quickly observed that RARM outperforms Apriori in all situations. In Figure 8(a), RARM maintains a steady speed-up of at least 20 times for support thresholds ranging from 3% to 1.5% in D_1. However, when the support threshold falls below 1.5%, the speed-up is significantly reduced. This is due to the fact that larger-sized frequent itemsets exist and RARM uses the Apriori algorithm for discovering large k-itemsets for $k > 3$. Hence, the computation savings in the first two iterations are insignificant compared to the huge computation costs in obtaining larger frequent itemsets.

The situation changes dramatically in D_2. Figure 8(b) uses a log scale for the time axis because of the vast difference between the execution times of RARM and Apriori. RARM performs at least 70 times faster than Apriori for support thresholds ranging from 3% to 1%. Its performance peaks at a support threshold of 2% where it performs 120 times faster than Apriori. We cannot determine the execution time of Apriori for a support threshold of 0.5% because there is insufficient memory to hold the number of candidate 2-itemsets. RARM does not have this memory problem because it can obtain large 2-itemsets without generating candidate 2-itemsets. The lowest support threshold that allows us to compare the performance of the two algorithms is 0.75% and at this threshold, RARM outperforms Apriori by a factor of 47.

The vast improvement of RARM in D_2 can be explained by Figure 9 which shows the number of candidate k-itemsets generated during the mining of D_1 and D_2 for a support threshold of 0.75%. From Figure 9, it is obvious that the main difference in candidate generation between D_1 and D_2 is in the generation of candidate 2-itemsets and that D_1 has larger-sized candidate itemsets. Thus, by eliminating the need for candidate 2-itemset generation, RARM achieves a much greater speed-up in D_2 as it contains more than twice the number of candidate 2-itemsets as compared to D_1. As

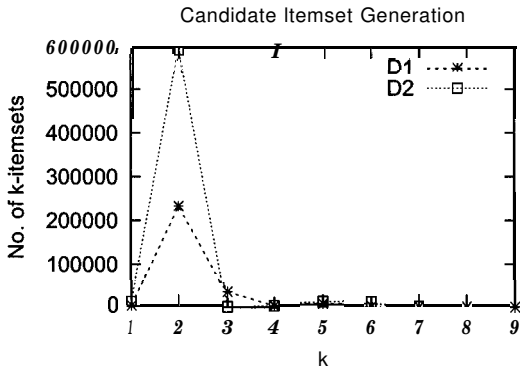

Figure 9: Size of candidate k-itemsets for D_1 and D_2 at a support threshold of 0.75%.

for higher support thresholds, RARM is able to outperform Apriori by up to two orders of magnitude because in D_2, the maximum size of the large itemsets, k^*, is much lower than that of D_1. If k^* increases indefinitely, the performance of RARM will eventually be reduced to that of Apriori. However, we can conclude from the experiments that as databases increase in size, k^* will decrease and thus RARM will scale up well against Apriori.

7.2 Comparison of RARM and FP-growth

Due to the lack of time, FP-growth is not implemented but its performance against RARM can be evaluated using Apriori as a basis for relative comparisons. The experiments conducted in [7] reported an overall improvement of only an order of magnitude for FP-growth over Apriori. In addition, the performance of FP-growth is on par with Apriori for support thresholds ranging from 3% to 1.5% in D_2. The poor performance of FP-growth can be attributed to the cost in recursively constructing FP-trees. Hence, significant speed-ups can only be noticed in lower support thresholds when Apriori cannot cope with the exponential increase in candidate itemset generation. This is undesirable because we want to mine databases at a wide variety of support thresholds quickly instead of only at low support thresholds. RARM overcomes this limitation of FP-growth and consistently outperforms Apriori at all support thresholds and experiments show that it can even perform up to two orders of magnitude faster than Apriori. In future, FP-growth will be implemented to directly access its performance against RARM but in this case, it is clear that RARM will outperform FP-growth.

7.3 Pre-processing Requirements

As pre-processing is carried on a transaction at the moment it arrives in the database, it is distributive by nature and thus will not burden a system excessively. RARM spends an average of only 22 ms and 48 ms in pre-processing a single transaction found in D_1 and D_2 respectively. This amount of time is insignificant considering that it will result in major speed-ups in the mining process. This requirement should not be taken into consideration in comparing the performance of RARM and Apriori because pre-processing is done outside of the actual mining process itself.

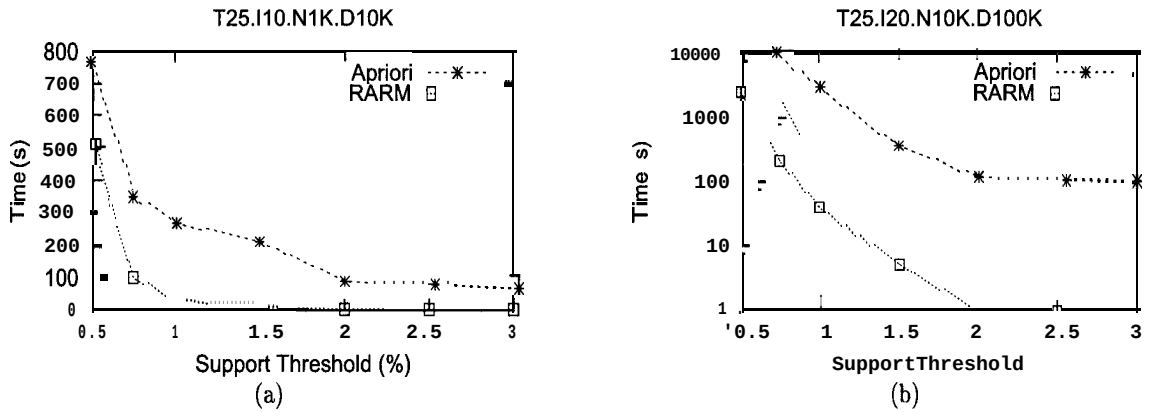

Figure 8: Execution times for two databases of the form $\mathbf{T}w.\mathbf{I}x.\mathbf{N}y.\mathbf{D}z$ where w is the average size of transactions, x is the average size of maximal potentially large itemsets, y is the number of unique items and z is the size of the database, at varying support thresholds.

7.4 Storage Requirements

The SOTrieIT structure resides in both memory and files. As primitive integer arrays are employed in memory for storing the first-level nodes, the SOTrieIT only takes up only 2 KB and 14 KB in D_1 and D_2 respectively. Second-level nodes grow exponentially with respect to N as seen in Section 6.12 and as such, they cannot be stored in memory. Instead, they are stored in files which are named after the labels of their parents. These files take up approximately 2 MB and 53 MB for D_1 and D_2 respectively. Therefore, it can be concluded that by distributing the SOTrieIT structure among memory and files, scalability is ensured as hard disk space is currently in the realm of tens of gigabytes.

8. CONCLUSIONS

The rising popularity of electronic commerce presents new challenges to association rule mining. Due to the easy availability of huge amount of transactional data, there is a greater need for faster and scalable algorithms to exploit this knowledgebase. We have proposed a new algorithm called RARM which uses an efficient trie-like structure known as the SOTrieIT. By eliminating the need for candidate 1-itemset and 2-itemset generation, RARM is able to achieve significant speed-ups. Experiments have shown that RARM is much faster than Apriori and FP-growth. It also maintains its sharp edge at various support thresholds and is scalable. Though there are additional pre-processing and storage requirements, they are both insignificant and worthwhile considering the immense speed-up they can help achieve.

The SOTrieIT is a simple and yet highly dynamic structure which can be put to greater use in association rule mining. Due to the dynamic nature of the Internet, current algorithms are inadequate to handle web-based databases. The SOTrieIT may be a good tool to impart dynamism into static algorithms. Hence, methods and algorithms to exploit the SOTrieIT will be explored in future work.

9. REFERENCES

[1] C. C. Aggarwal and P. S. Yu. Online generation of association rules. In *Proc. 14th Int. Conf. on Data* Engineering, pages 402-411, Orlando, Florida, USA, 1998.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th* Int. *Conf.* on Very Large Databases, pages 487-499, Santiago, Chile, 1994.

[3] M. Berry and G. Linoff. *Data* mining *techniques: for* marketing, sales, and *customer* support. Wiley Computer Pub., New York, 1997.

[4] A. Berson, S. Smith, and K. Thearling. Building data mining applications for *CRM*. McGraw-Hill, New York, 2000.

[5] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. 12th Int. Conf.* on Data Engineering, pages 106-114, New Orleans, Louisiana, 1996.

[6] D. W. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. 5th* Znt. Conf. *on Database* Systems *for Advanced Applications,* pages 185-194, Melbourne, Australia, 1997.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD* Int. *Conf.* on Management *of Data,* pages 1-12, Dallas, Texas, 2000.

[8] C. Hidber. Online association rule mining. In *Proc. ACM SIGMOD Int. Conf.* on Management *of* Data, pages 145-154, Philadelphia, Pennsylvania, USA, 1999.

[9] J. S. Park, M. S. Chen, and P. S. Yu. Using a hash-based method with transaction trimming for mining association rules. *IEEE Trans.* on *Knowledge* and *Data* Engineering, 9(5):813-824, Sept. 1997.

[10] M. R. Sanders. Global *eCommerce Approaches Hypergrowth.* http://www.forrester.nl/ER/Research/Brief/Excerpt/0,1317,9229,FF.html, April 2000.

[11] N. L. Sarda and N. V. Srinivas. An adaptive algorithm for incremental mining of association rules. In *Proc. 9th Int.* Conf. on *Database* and *Expert Systems,* pages 240-245, Vienna, Austria, 1998.