Fast Online Dynamic Association Rule Mining

Yew-Kwong Woon  Wee-Keong Ng  Amitabha Das
Nanyang Technological University
Nanyang Avenue
Singapore 639798, SINGAPORE
davidwyk@singnet.com.sg  wkn@acm.org  asadas@ntu.edu.sg

Abstract

At present, there are no association rule mining algorithms that are suitable for use in electronic commerce because they do not consider that new products are introduced and old ones are retired frequently and they assume that support thresholds do not change. In this paper, a new algorithm called Fast Online Dynamic Association Rule Mining (FOLDARM) is introduced for mining in electronic commerce. It uses a novel tree structure known as a Support-Ordered Trie Itemset (SOTrieIT) structure to hold pre-processed transactional data. It allows FOLDARM to generate large 1-itemsets and 2-itemsets quickly without scanning the database. In addition, the SOTrieIT structure can be easily and quickly updated when transactions are added or removed. It also stores data that is independent of the support threshold and thus can be used for mining with varying support thresholds without any degradation in performance. Experiments have shown that FOLDARM outperforms Apriori, a classic mining algorithm, by up to two orders of magnitude (100 times).

1. Introduction

Since the introduction of the Apriori algorithm [1] in 1994, there has been sustained interest in researching new association rule mining algorithms that can perform more efficiently. However, to the best of our knowledge, these existing algorithms are not designed for use in a fast-changing and widely distributed environment like the Internet. In electronic commerce where a huge number of transactions can arrive at a virtual store from all over the world 24 hours a day, transaction databases are expected to be updated frequently. Thus, existing algorithms that are designed to mine static databases that are not expected to change, will not be able to perform as well with databases that are constantly changing. There are some algorithms that can perform incremental mining, which means that they can improve mining speed by reusing past mined information. However, in the next section, we will see that such algorithms cannot cope with databases with very frequent updates.

In addition, in a highly competitive setting like electronic commerce, companies will need to constantly introduce new products and remove unpopular products to satisfy the increasingly demanding needs of the now empowered customer. This means that the number and type of unique items in the database will change very often. Unfortunately, existing algorithms assume that unique items are fixed and thus, each time items are added or removed, the algorithms must mine the database from scratch and discard valuable past mined results.

Finally, given the dynamism and volatility of transactional data in electronic commerce, companies cannot predict a suitable support threshold to set for the mining process. Using too high a threshold may result in too many unimportant rules while too low a threshold may result in certain important rules being passed over. Therefore, there is a need to mine the database with several different support thresholds before an optimal threshold can be determined. This critical need has not been effectively tackled by current algorithms.

Recently, we have introduced an innovative algorithm called Rapid Association Rule Mining (RARM) [2] to mine static databases efficiently. In this paper, a new algorithm called Fast Online Dynamic Association Rule Mining (FOLDARM) is proposed as an extension of RARM so that association rule mining can be performed more efficiently in electronic commerce. Like RARM, FOLDARM constructs a new data structure called Support-Ordered Trie Itemset (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 will be used to update the SOTrieIT without the need for prior knowledge of the support threshold. This structure is sorted according to the support counts of each node in descending order. FOLDARM
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 [3]. Therefore, by eliminating this need, FOLDARM achieves significant speed-ups. Subsequently, it applies the Apriori algorithm to obtain larger-sized itemsets. Moreover, unlike RARM, FOLDARM allows transactions and unique transactional items to be added and removed without the need to destroy the current SOTrieIT and rebuild it.

It is now clear to see how FOLDARM obtains its name; it is *fast* because the SOTrieIT is constructed in an incremental manner; it is *online* because users can vary the support threshold while maintaining the same performance; it is *dynamic* because the unique set of items, which we define as the *Universal Itemset*, can be changed and be easily accommodated in the SOTrieIT. Experiments have been conducted to study the performance of FOLDARM and compare it against Apriori [1]. FOLDARM is found to be up to 100 times faster than Apriori.

The rest of the paper is organized as follows. The next section reviews related work. Section 3 gives a description of the problem while Section 4 presents the new tree structure. Section 5 describes the FOLDARM algorithm. Time and space complexity of the new structure will be examined in Section 6. Performance evaluation is discussed in Section 7. Section 8 compares the salient features of FOLDARM with existing algorithms to explain its edge over them. Finally, the paper is concluded and recommendations for future work are made in Section 9.

## 2. Related Work

The Apriori algorithm [1] is the first successful algorithm for mining association rules. Since its introduction, it has popularized the task of mining association rules and sparked off many research papers. It introduces a method to generate candidate itemsets \( C_k \) in a pass \( k \) using only large itemsets \( L_{k-1} \) in the previous pass. The idea rests on the fact that any subset of a large itemset must be large as well. Hence, \( C_k \) can be generated by joining \( L_{k-1} \) and deleting those that contain any subset that is not large. This would result in a significantly smaller number of candidate itemsets being generated.

After Apriori, the *Direct Hashing and Pruning* (DHP) algorithm [3] is the next most widely used algorithm for the efficient mining of association rules. It employs a hash technique to reduce the size of candidate itemsets and the database. This amounts to significant speed-ups because the dominating factor in the generation of large itemsets is the size of the candidate itemsets. DHP has significant speed improvements due to the reduced size of the candidate itemsets generated. However, it incurs additional overhead due to the need to do hashing and to maintain a hash table. After some experiments [3], it is concluded that the hash technique should only be applied during the generation of candidate 2-itemsets to achieve speed-ups of up to 3 times against Apriori.

The *Fast Update* (FUP) algorithm [4] is an incremental algorithm which makes use of past mining results to speed up the mining process. Its successor, the *Fast Update Two* (FUP2) algorithm [5], is a faster version and generalization of it. By setting bounds for the support counts of candidate itemsets, it is able to reduce the size of \( C_k \) and improve its efficiency. By re-using past mining information, FUP2 reduces the number of candidate sets and hence achieves a speed improvement of up to 2 times over Apriori. However, when the size of the updates exceeds 40% of the original database, Apriori performs better.

Incremental mining is brought to another new level when the Adaptive algorithm [6] is introduced. This algorithm is not only incremental but also adaptive in nature. By inferring the nature of the incremental database, it can avoid unnecessary database scans. Experiments have shown that it can perform up to seven times faster than Apriori.

Unlike all the discussed algorithms, the *Continuous Association Rule Mining Algorithm* (CARMA) [7] allows the user to change the support threshold and continuously displays the resulting association rules with support and confidence bounds during the first scan or phase. During the second phase, it determines the precise support of each itemset and extracts out all the large itemsets. With the support lattice, CARMA can readily compute large itemsets for varying support thresholds. However, experiments reveal that CARMA only performs faster than Apriori at support thresholds of 0.25% and below.

Finally, the *Frequent Pattern-growth* (FP-growth) algorithm [8] is a most recent association rule mining algorithm which achieves impressive results. It uses a compact tree structure called a *Frequent Pattern-tree* (FP-tree) to store information about large 1-itemsets. 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. 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, since the construction and use of the FP-trees are complex,
the performance of FP-growth is reduced to be on par with Apriori at support thresholds of 3% and above. It only achieves significant speed-ups at support thresholds of 1.5% and below. Moreover, it is only incremental to a certain extent depending on the FP-tree watermark (validity support threshold).

The features and performance of the discussed algorithms are presented in Figure 1. To sum up, none of the algorithms are suitable for use in electronic commerce because there is not one that is incremental, supports dynamic thresholds and at the same time, performs fast enough for online queries. Moreover, all of them assume that the universal set of unique items do not change. Should the unique items change, mining must be done from afresh and past mined information cannot be reused. In contrast, our proposed algorithm, FOLDARM possesses all the essential features of an ideal mining algorithm for electronic commerce.

3. Problem Description

The problem of mining association rules is described as follows: Let the universal itemset, \( I = \{a_1, a_2, \ldots, a_n\} \) 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 \( \sigma_X \) 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 \( X \implies Y \), where \( X \subseteq I \), \( Y \subseteq I \) and \( X \cap Y = \emptyset \). The association rule \( X \implies Y \) 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 \( X \implies Y \) 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 \( \text{Digital Camera} \implies \text{Printer} \) means that whenever customers buy digital cameras, they also buy printers \( c\% \) of the time and this trend occurs \( s\% \) of the time. This problem consists of finding large itemsets first and then generating association rules from the large itemsets. We will only address the first sub-problem of finding large itemsets because it is much more computationally expensive and thus it is the main bottleneck in association rule mining.

4. Data Structure

Though our new data structure, the SOTrieIT, is described in detail in our previous work [2], we will briefly describe it here again for completeness.

4.1. A Complete TrieIT

The database \( D \) of transactions is stored in a forest of lexicographically-ordered tree nodes known as Trie Itemset (TrieIT). Let the set of items \( I = \{a_1, a_2, \ldots, a_N\} \) be ordered so that for any two items \( a_i \in I, a_j \in I \) \((1 \leq i, j \leq N)\), \( a_i < a_j \) if and only if \( i < j \).

Definition 1 (Complete TrieIT)

A complete TrieIT is a tree structure such that every tree node \( w \) is a 2-tuple \((w_i, w_j)\) where \( w_i \in I \) is the label of the node and \( w_j \) 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 \( C(w_i) \neq \emptyset \), then \( C(w_i) \subseteq \{w_{i+1}, w_{i+2}, \ldots, w_N\} \).
2. Given a node \( w_i \), let \( w_k, w_{k+1}, \ldots, w_{i-1} \) \((1 \leq k \leq i - 1)\) be the set of nodes on the path from the root to the parent of \( w_i \), then \( w_k \) 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 \( k \).

Let \( W_i \) be a complete TrieIT whose root node has label \( a_i \). Then the \( D \) is stored in a set of complete TrieITs denoted by \( W \) where \( W \subseteq \{W_1, W_2, \ldots, W_N\} \).

For every transaction that arrives, the complete TrieIT needs to be updated with the powerset of the transaction items. The amount of storage space needed by the complete TrieIT scales exponentially with respect to the number of unique items. Hence, with its expensive computation and storage requirements, the complete TrieIT is not a practical data structure. We will discuss a better alternative in the next section.

4.2. Support-Ordered Trie Itemset

This new design builds on the ideas presented in the paper on DHP [3]. In that paper, it is discovered that generation of large 2-itemsets is the main performance bottleneck. Using a hashtable, 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
Definition 2 (SOTrieIT)

A SOTrieIT has the same properties as the complete TrieIT except the following:

1. Let \( Y_i \) be a SOTrieIT whose root node has label \( a_1 \).
2. The nodes are sorted according to their support counts in descending order from the left.

In other words, the set of SOTrieITs only keeps a record of all 1-itemsets and 2-itemsets contained in a transaction. Therefore, much computation time is saved as compared to the case with the complete TrieIT where the power-set of transaction items is extracted. The first-level nodes represent 1-itemsets while second-level nodes represent 2-itemsets. The resultant structure will be much smaller than the multi-level complete TrieIT. Henceforth, we shall use the term SOTrieIT to denote a set of SOTrieITs. 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 during the generation of large 1-itemsets and 2-itemsets.

**Example** Figure 3 represents the fully constructed SOTrieIT for the example transaction database in Figure 2. 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. When both transaction 100 and 200 arrive, the nodes created are shown in Figures 3(a) and 3(b). Notice that in Figure 3(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: \( \{ A \}, \{ B \}, \{ C \}, \{ A, B \}, \{ A, C \}, \{ B, C \}, \{ B, D \}, \{ C, D \} \). The SOTrieITs are updated in a similar fashion for transaction 400 as seen in Figure 3(d).

**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 \( T \) 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 \( x < y \leq s \). 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 FOLDARM**

**5.1. Pre-processing**

Figure 4 shows the pre-processing steps taken whenever a transaction is added or deleted. For every transaction that arrives, 1-itemsets and 2-itemsets are first extracted from it. For each itemset, the SOTrieIT \( Y_i \) 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 that makes it very fast.
1 Let $Y$ be a set of SOTrieITs
2 for $(k = 1; k \leq 2; k++)$ do begin
3   Obtain all $k$-itemsets of the transaction and store them in $C_k$
4   foreach itemset $X \in C_k$ 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 or decrement support count of the leaf node depending on the nature of update
8        if its support count falls to 0, remove node and its child nodes (if any)
9        Sort the updated node according to its new support count in descending order
11      else
12        Create a new set of nodes with support counts of 1 that represent a path to $X$
13        Insert nodes into $Y$ according to their support counts in descending order from the left
14     endif
15   endfor
16 endfor

Figure 4. Pre-processing Algorithm.

$Y$ 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. For deletions, the steps are similar except that the support counts of the affected nodes are decremented and nodes are deleted if their support counts fall to zero.

5.2. Update of Universal Itemset

Figure 5 shows how the SOTrieIT is updated when the universal itemset is changed. In algorithms like FP-growth that use a similar data structure to store itemset information, the structure must be rebuilt to accommodate updates to the universal itemset. 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.

Example To illustrate the mining algorithm, we use the same transaction database found in Figure 2 and the SOTrieIT structure in Figure 3(d). Suppose the support threshold is set at 75%. Then the minimum support count to qualify an itemset to be large is 3. Figure 7 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. 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. But 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 $\{A, D\}$ will not be explored. 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 9 (out of a maximum 10). The scenario changes favorably.
Therefore, there will not be any more large itemsets in the traversals and the only large itemset is \( SOTrieIT \); once constructed, it can be used for varying support thresholds. In addition, in a best case scenario where we increase the support threshold slightly. The next example demonstrates more clearly the advantage of ordering the \( SOTrieIT \) by the support counts of nodes.

![Figure 7. Traversal path of the SOTrieIT at a support threshold of 75%.
](image)

When we increase the support threshold slightly. The next example demonstrates more clearly the advantage of ordering the \( SOTrieIT \) by the support counts of nodes.

For a minimum support threshold of 80%, the minimum support count needed is 4. Figure 8 shows the traversal path taken in obtaining the large 1-itemsets and 2-itemsets. The FOLDARM 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 after only 3 traversals and the only large itemset is \( \{C\} \).

The above examples illustrate the usefulness of the \( SOTrieIT \); once constructed, it can be used for varying support thresholds. In addition, in a best case scenario where the minimum support threshold is high, it may only need one traversal to discover all large itemsets while in a worst case scenario, it may only need a number of traversals whose cost is definitely lesser than that of scanning a large database. Time and space complexity issues will be explored further in the next section.

6. Time and Space Complexity

6.1. Pre-processing

**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 \( (\ell C_1 + sC_2) \) itemsets are pre-processed. Hence, its complexity is \( O(s^2) \). 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, it will take a maximum of \( 2 \times (\ell C_1 + sC_2) \) units of time to move to all the nodes required by a transaction of size \( s \).

**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 that has the first position in a set of lexicons. It will have \( N - 1 \) 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

This 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.
### 7. Performance Evaluation

The section evaluates and compares the relative performance of the Apriori and FOLDARM algorithms by conducting experiments on a Pentium-III machine with a CPU clock rate of 1.7 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. Future experiments will be conducted to tackle this issue. The SOTrieIT structure is implemented using a combination of integer arrays and files. Implementation details are omitted due to the lack of space. In spite of extra file I/O requirements, FOLDARM maintains its impressive performance.

The method used for generating synthetic data is similar to the one used in [1]. To describe an experiment, we use the notation Tw.x.Ny.Dz modified from the one used in [1] 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. We added the y parameter to represent the databases in a clearer manner. The databases used here are similar to those in [8]. The first one is T25.100.N1K.D10K which is denoted as D1 while the second is T25.200.N10K.D100K which is denoted as D2. The following sections analyze the performance of FOLDARM as compared to the algorithms introduced in Section 2 in different scenarios.

#### 7.1. Static Databases

Figure 10 shows the execution times (excluding preprocessing time of FOLDARM) for the two different static databases of both Apriori and FOLDARM. The databases are termed static because they are not expected to change over time. From the graphs, it can be quickly observed that FOLDARM outperforms Apriori in all situations. In Figure 10(a), FOLDARM maintains a steady speed-up of about 10 times for support thresholds ranging from 3% to 1.5% in D1. However, when the support threshold falls below 1.5%, the speed-up is significantly reduced. At a support threshold of 0.5%, FOLDARM only manages a speed-up of 1.2 times.

The situation changes dramatically in D2. Figure 10(b) uses a log scale for the time axis because of the vast difference between the execution times of FOLDARM and Apriori. FOLDARM performs at least 80 times faster than Apriori for support thresholds ranging from 3% to 2%. Its performance peaks at a support threshold of 1.5% where it performs more than 160 times faster than Apriori. This speed-up falls to 54 times at a support threshold of 0.5%.

**Explanation** The poor performance of FOLDARM in D1, especially at lower support thresholds, is due to the fact that more larger-sized frequent itemsets exist at lower support thresholds and FOLDARM uses the Apriori algorithm to discover large k-itemsets for k > 3. Hence, the computation savings in the first two iterations are insignificant compared to the huge computation costs needed in obtaining larger frequent itemsets.

Interestingly, in larger databases like D2, FOLDARM performs exceptionally well. The obvious vast improvement of FOLDARM in D2 can be explained by Figure 11 which shows the number of candidate k-itemsets generated during the mining of D1 and D2 for a support threshold of 0.5%. From Figure 11, it is clear that the main difference in candidate generation between D1 and D2 is in the number of candidate 2-itemsets generated. Thus, by eliminating the need for candidate 2-itemset generation, FOLDARM is able to outperform Apriori by up to two orders of magnitude because in D2, the maximum size of the large itemsets, k*, is much lower than that of D1. If k* increases indef-

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Number of unique items</td>
</tr>
<tr>
<td></td>
<td>Number of transactions</td>
</tr>
<tr>
<td></td>
<td>Number of maximal potentially large itemsets</td>
</tr>
<tr>
<td></td>
<td>Average size of the transactions</td>
</tr>
<tr>
<td></td>
<td>Average size of the maximal potentially large itemsets</td>
</tr>
</tbody>
</table>

Figure 9. Definition of Parameters.
of FP-growth is on par with Apriori for support thresholds ranging from 3% to 1.5% in addition, the performance of FOLDARM will eventually be reduced to that of Apriori. However, we can conclude from the experiments that as databases and universal itemsets increase in size, $k^*$ will decrease and thus FOLDARM will scale up very well against Apriori. This finding is particularly crucial in an electronic commerce where transactions (database size) are expected to arrive by the thousands and a wide variety of products (universal itemset size) is available for purchase.

Another important point to note is that after the first 2 passes, FOLDARM actually uses Apriori to mine larger frequent itemsets without relying on the SOTrieIT structure. Despite the fact that FOLDARM only differs from Apriori during the first 2 passes, it is much faster and scalable; this confirms that candidate itemset generation during the first 2 passes constitutes the main bottleneck in Apriori.

FP-growth is currently the fastest algorithm for mining static databases. However, due to the lack of time, FP-growth is not implemented but its performance against FOLDARM can be evaluated using Apriori as a basis for relative comparisons. The experiments conducted in [8] report 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 $D_1$. 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 efficiently at a wide range of support thresholds instead of only at low support thresholds. FOLDARM overcomes this limitation of FP-growth and consistently outperforms Apriori at all support thresholds and it can even perform up to two orders of magnitude faster than Apriori.

7.2. Dynamic Databases

A dynamic database is one with frequent updates; transactions are added and removed frequently. In [5], the FUP2 algorithm is presented to make use of past mining results to mine new transactional updates more efficiently. Due to time constraints, FUP2 is not implemented for comparison studies with FOLDARM. As before, with the results in its performance analysis section, we can easily assess its performance as compared to FOLDARM with the performance of Apriori as our reference point.

In a database of the type T10.I4.N1K.D100K with an addition and deletion of 5000 transactions, it is found that FUP2 performs only about twice as fast as Apriori as seen in the experiments in [5]. This is its best performance against Apriori through the use of past mined knowledge. In addition, it is discovered that when the size of the updates of a database exceeds 40% of the its original size, Apriori can even outperform FUP2. Both Apriori and FOLDARM mine a dynamic database as if it were a static database because they do not make use of past results. Since FOLDARM performs up to 100 times faster than Apriori in a much larger database, we can safely conclude that FOLDARM will definitely outperform FUP2 by a wide margin.

Explanation In spite of its ability to reuse mined results and hence reduce the number of candidate itemsets generated, FUP2 still performs slower than FOLDARM because it needs to scan the database during the generation of large 1-itemsets and 2-itemsets. FOLDARM can quickly do so simply by scanning the SOTrieIT structure which is definitely many times smaller than the database. In situations when the updates are high, the performance of FUP2 drops because the updated database becomes so different from the original one that past mining results are not helpful in determining new large itemsets. In processing useless old information, precious computation time is wasted. On the other hand, FOLDARM does not need to retain past mining results. Moreover, frequent database updates do not affect it because it always mines the database from scratch. Note that the SOTrieIT structure does not need to be constructed from scratch when the database is updated because it is continuously updated each time a transaction is added or retired.

7.3. Dynamic Support Threshold

CARMA is currently the only algorithm that allows the user to modify the support threshold on the fly. We will once again use the results presented in [7] as a form of comparison. In a database of the form T10.I4.N10K.D100K, it is found that Apriori outperforms CARMA for support thresholds of 0.5% and above. It is only when the support thresh-
old thresholds are at 0.25% and below that CARMA begins to outperform Apriori only by less than 1.5 times. On the other hand, FOLDARM consistently outperforms Apriori by wide margins at various support thresholds and supports dynamic support thresholds. This is because the SOTrieIT can be reused for mining at different support thresholds without additional computation.

Explanation  The poor performance of CARMA is attributed to its need to maintain a lattice of potentially large itemsets. It will be faster only when the user does not need a precise set of large itemsets because in this case, CARMA does not need to re-scan the database. FOLDARM performs much faster because the SOTrieIT stores threshold-independent information and thus its performance will not be affected even if the user changes the support thresholds frequently to obtain an optimal threshold.

7.4. Dynamic Universal itemset

As none of the algorithms discussed takes into consideration of the fact that unique items in the database will vary, it is not possible to conduct experiments for comparison. However, we can approximate the most probable results by examining the characteristics of each algorithm. When new transactions with new items are added to a database or when old transactions with obsolete items are retired, all the discussed algorithms would have to mine the updated database from scratch. Therefore, their performance against FOLDARM in such a scenario can be deduced from the previous sections. When the universal itemset is changed, the SOTrieIT can be easily updated as seen in Figure 5. Hence, it should retain its performance edge and outperform all the algorithms seen in the previous sections.

7.5. Pre-processing and Storage 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. FOLDARM spends an average of only 180 ms and 250 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 FOLDARM and Apriori because pre-processing is done outside of the actual mining process itself.

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 6.1 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 Comparison of features

This section focuses on the salient features of FOLDARM and elaborates on why it is more suitable for

Figure 10. Execution times for two databases of the form $T_w,I_x,N_y,D_z$ 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.
use in electronic commerce as compared to existing algorithms discussed in Section 2. For an association rule mining algorithm to be useful in electronic commerce, it must have the following capabilities:

1. **General Incremental Mining Support**: The algorithm must obtain association rules quickly that reflect the latest changes to the transaction database at a certain point in time. This can only be achieved if the algorithm can perform general incremental mining which means that past mining results are exploited during the mining of a database which has many additions and deletions of transactions since the last mining operation that was carried out on it.

2. **Dynamic Threshold Support**: The fast-changing nature of electronic commerce prevents the prediction of a suitable support threshold for the mining process. Hence, the algorithm must allow the user to change the support thresholds of the mining process until an optimum value is found without significant performance degradation. In other words, it must be able to create and reuse mining information that is independent of the support threshold.

3. **Dynamic Universal itemset Support**: In a highly-competitive environment like electronic commerce, new products must be introduced frequently and old ones be retired to satisfy the changing and demanding needs of the empowered customer. Therefore, the algorithm must not assume that items in the database remain fixed. It must be able to reuse past mining results efficiently regardless of changes made to the set of unique items in the database.

Apriori [1], DHP [3] and FP-growth [8] all attempt to improve mining speed but do not satisfy any of the above criteria. FUP [4], FUP2 [5] and the Adaptive algorithm [6] satisfy the first criteria but do not take into consideration the other two. CARMA [7] fulfills the second criteria but does not satisfy the other two and is extremely slow. Finally, no algorithms exist to date that satisfies the third criteria. With the SOTrieIT, FOLDARM satisfies all three criteria and even performs faster than all algorithms.

### 9. Conclusions

The increasing 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 urgent need for faster algorithms to mine such rich data. We have proposed a new algorithm called FOLDARM which uses an efficient novel structure known as the SOTrieIT. By simply eliminating the need for candidate 1-itemset and 2-itemset generation, FOLDARM is able to achieve significant speed-ups. Experiments have shown that FOLDARM is much faster than Apriori. By using Apriori as a basis for comparisons, FOLDARM is proven to be faster than some prominent existing algorithms. In addition, as FOLDARM can maintain its performance while the support threshold and universal itemset change, it is ideal for use in electronic commerce. Therefore, though there are additional pre-processing and storage requirements, they are both insignificant and worthwhile considering the advantages that they reap. As databases and their universal itemsets grow in size, it will be more difficult to maintain the SOTrieIT. Hence, parallel versions of the SOTrieIT and the FOLDARM algorithm need to be researched to meet the rising demands of mining larger databases.

### References


