PERGAMON Mathematical and Computer Modelling 38 (2003) 739–751

MATHEMATICAL AND COMPUTER MODELLING

www.elsevier.com/locate/mcm

Trie: An Alternative Data Structure for Data Mining Algorithms

F. BODON AND L. RÓNYAI
Computer and Automation Institute, Hungarian Academy of Sciences
Lágymányosi u. 11., H-1111 Budapest, Hungary
bodon@mit.bme.hu
ronyai@sztaki.hu

Abstract—Frequent itemset mining is one of the most important data mining fields. Most algorithms are APRIORI based, where hash-trees are used extensively to speed up the search for itemsets. In this paper, we demonstrate that a version of the trie data structure outperforms hash-trees in some data mining applications.

Tries appear to offer simpler and scalable algorithms which turned out to be faster for lower support thresholds. To back up our claims, we present test results based on real-life datasets. © 2003 Elsevier Ltd. All rights reserved.

Keywords—Data mining, Frequent itemset, Trie, Hash-tree.

1. INTRODUCTION

Finding frequent itemsets is one of the most researched fields of data mining. The problem was first presented in [1]. A later paper [2] is considered as one of the most important contributions to the subject. Its main algorithm, APRIORI, not only influenced the association rules mining community, but it affected other data mining fields, such as episode rules mining, as well.

Association rule and frequent itemset mining became a widely researched area, and hence, faster and faster algorithms have been presented. Many of them are APRIORI-based algorithms or APRIORI modifications. Those who adopted APRIORI as a basic search strategy tended to adopt the whole set of procedures and data structures as well [3–6]. Since the scheme of this important algorithm was not only used in basic association rules mining but also in other data mining fields (hierarchical association rules [7–9], association rules maintenance [10–13], sequential pattern mining [14,15], episode mining [16]), it seems appropriate to critically examine it, and improve if possible.

A central data structure of the algorithm is the so-called hash-tree. In this paper, we will show that a version of the familiar trie data structure can profitably substitute hash-trees. It provides simpler algorithms, which are faster for a range of applications and avoids the need of fine tuning by employing a self-adjusting approach.

The rest of the paper is organized as follows. In Section 2, the problem is presented, in Section 3, the APRIORI algorithm is introduced, and in Section 4, hash-trees are described. In

This research was supported in part by Hungarian OTKA Grants T42481 and T42706, NWO-OTKA Grant N34010, and the EU-COE Grant of MTA SZTAKI.

0895-7177/03/$ - see front matter © 2003 Elsevier Ltd. All rights reserved. Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX
doi: 10.1016/S0895-7177(03)00275-9

Section 5, we show how tries can be used instead of hash-trees, and in Section 6, we will examine the advantages of tries against hash-trees. Section 7 describes experiments whose results seem to support our claims on the merits of the trie-based solution. In the conclusion, we summarize the results.

2. PROBLEM STATEMENT

Association rules mining came from efforts to discover useful patterns in customers' transaction databases. Here *useful pattern* means a conditional implication, for example the following one: 87% of the customers that have bought beer have also bought chips.

A customers' transaction database consists of pairs (t_id, I), where t_id is the unique identifier of the transaction, and I is a set of products (itemset), so if $\mathcal{I}$ is the set of all products ($\mathcal{I} = \{i_1, \dots, i_n\}$, where i_j is a product, or item), then $I \subseteq \mathcal{I}$. We call an itemset that has k elements a k-itemset. The transaction $t_j = (t\_id_j, I_j)$ contains an itemset X if $X \subseteq I_j$. The support of an itemset X in a customers' transaction database ($\mathcal{T}$), denoted as $\text{supp}_{\mathcal{T}}(X)$, is the fraction of those transactions that contain X, so

$$\text{supp}_{\mathcal{T}}(X) = \frac{|\{t_j : t_j \text{ contains } X\}|}{|\mathcal{T}|}.$$

Let $\mathcal{T}$ be a set of transactions over itemset $\mathcal{I}$. Let $X, Y \subseteq \mathcal{I}$ be disjoint itemsets and $c, s > 0$ real parameters. We say that the association rule

$$X \xrightarrow{c,s} Y$$

holds in $\mathcal{T}$, if $\text{supp}_{\mathcal{T}}(X \cup Y)/\text{supp}_{\mathcal{T}}(X) \geq c$ and $\text{supp}_{\mathcal{T}}(X \cup Y) \geq s$. Here c is the confidence threshold and s is the support threshold of the rule. In the association rules mining problem, we would like to find all association rules whose support and confidence are greater than given threshold values, traditionally denoted by *min_conf* and *min_supp*. The rules that meet these requirements are called valid association rules, and the itemsets that have support at least *min_supp* are called frequent itemsets.

Discovering valid association rules is useful in many settings. For example, it may provide a better understanding of consumers' preferences, when applied to a consumers' database. This improved knowledge can be profitable in cross-marketing, attached mailing, discount scheduling, catalogue design, and store layout, just to mention a few of the opportunities.

The association rule mining problem is always solved in two steps. First, frequent itemsets and their supports are determined, and then by splitting a frequent itemset into two subsets, association rules can be generated. If the confidence of this association rule is higher than the minimum confidence threshold c, then we found a valid rule.

Several different algorithms have been published after 1993, but the second step turns out to be the same for all. The reason for that might be that the solution is simple and relatively fast compared to the first step. Experiments show that the second step is at least a hundred times faster on the average than finding frequent itemsets. It seems, therefore, to be justified to focus on the efficiency of the first phase, the determination of frequent itemsets.

The first, and maybe the most important, solution for finding frequent itemsets is the APRIORI algorithm [2]. Later faster and more sophisticated algorithms have been suggested, most of them being modifications of APRIORI [3–6]. So if we improve algorithm APRIORI then we improve a whole family of algorithms. Next, we briefly review algorithm APRIORI.

3. ALGORITHM APRIORI

The algorithm scans the transaction datasets several times. After the first scan, the frequent one-itemsets are found, and in general, after the k^{th} scan the frequent k-itemsets are extracted.

The method does not determine the support of every possible itemset. In an attempt to narrow the domain to be searched, before every pass it generates *candidate* itemsets. An itemset becomes a candidate if every subset of it is frequent. Obviously, every frequent itemset needs to be a candidate too, and hence, only the support of candidates is calculated. Frequent k-itemsets generate the candidate $k+1$-itemsets after the k^{th} scan.

After all the candidate $k+1$-itemsets have been generated, a new scan of the transactions is effected and the precise support of the candidates is determined. Those candidates that have low support are thrown away. The algorithm ends when no candidates can be generated.

The intuition behind candidate generation is based on the following simple fact.

FACT 3.1. *Every subset of a frequent itemset is frequent.*

This is immediate, because if a transaction t supports an itemset X, then t supports every subset $Y \subseteq X$.

Using the fact indirectly, we infer that if an itemset has a subset that is infrequent, then it cannot be frequent. So in the algorithm APRIORI only those itemsets will be candidates whose every subset is frequent. The frequent k-itemsets are available when we attempt to generate candidate $k+1$-itemsets. The algorithm seeks candidate $k+1$-itemsets among the sets which are unions of two frequent k-itemsets. After forming the union, we need to verify that all of its subsets are frequent; otherwise it should not be a candidate. To this end, it is clearly enough to check if all the k-subsets of X are frequent.

Next, the supports of the candidates are calculated. This is done by reading transactions one by one. For each transaction t, the algorithm decides which candidates are supported by t. For fast execution of this step, algorithm APRIORI uses a hash-tree. In the rest of the paper, we make the (realistic) assumption that the items are from an ordered set, and transactions are stored as sorted itemsets.

4. DETERMINING SUPPORT WITH HASH-TREES

Here, we briefly outline the approach based on hash-trees. Suppose that we now have all the k-itemset candidates for some $k > 1$, and want to calculate their support.

These candidate k-itemsets will be stored in a hash-tree. A hash-tree is a rooted (downward), directed tree. Leaves of the tree contain the candidates and their support count. The interior nodes guide us to the leaves. The root of the hash-tree is defined to be at depth 1. Every interior node stores a hash-table, and a hash-table at depth j can only point to nodes that are at depth $j+1$.

When we add a new itemset, we start from the root, and go down the tree until a leaf is reached. At an interior node at depth j, we decide which branch to follow by applying a hash-function to the j^{th} item of the itemset, i.e., the j^{th} largest item of the candidate. The leaf we arrive at will store the candidate and its count. When the number of the itemset in a leaf at depth j exceeds a specified threshold *leaf_max_size*, it is converted to an interior node, and the itemsets it contains are distributed among new leaves according to the j^{th} hash values.[1]

Figure 1 shows a hash-tree, which contains five candidates of size 3. Here, the items are capital letters. The hash value of a letter is its sequential number in the English alphabet modulo 6 (0 for A, 1 for B, etc.). As G and M have the same hash value 1, the sets $\{A, E, G\}$ and $\{A, E, M\}$ are stored in the same leaf. The root has two children and the tree has four leaves.

Once we have inserted all the candidates into the tree, we can set out to process the transactions sequentially and count the support of the candidates. Let $t = (t\_id, I)$ be a transaction, where $I = \{i_1 < i_2 < \cdots < i_n\}$ is an ordered set of n items. For every k-subset X of I, we have to check if X is among the candidates, and if so, then we increase the count of X at the appropriate

[1]It should be noted that if k is the length of the candidates, leaves at depth $k+1$ cannot be converted to interior node even if the number of the itemsets exceeds *leaf_max_size*.

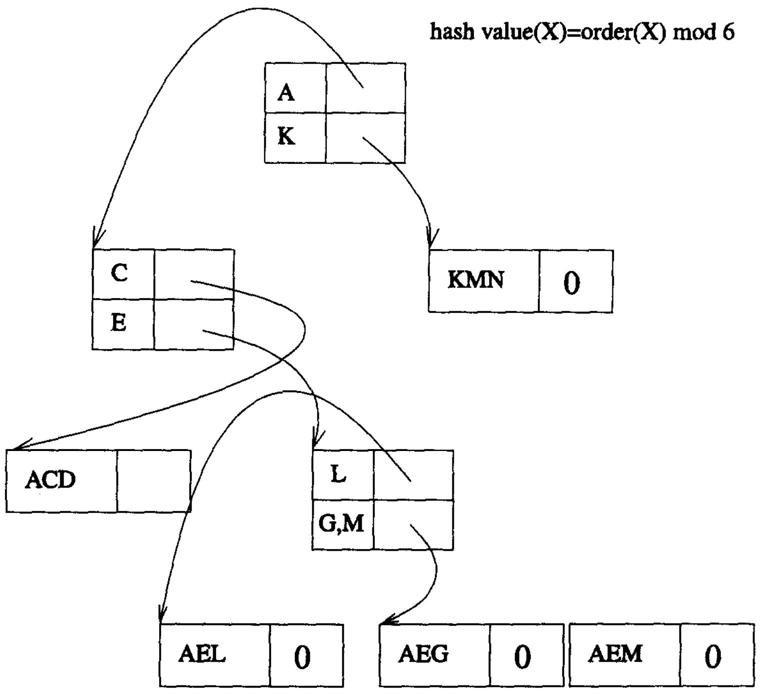

Figure 1. A hash-tree containing five candidates.

leaf by 1. We do not explicitly generate and test every k-subset of I. Navigation in the hash-tree often allows us to eliminate from consideration whole families of k-itemsets with a given prefix.

Please note that when we arrive at a leaf, then we have to test explicitly if the candidates stored in the leaf are actually subsets of I. The benefit of hash-trees is that the number of the explicit test is much less than the total number of candidates. A more detailed account can be found in [2].

As an illustration, suppose that we have to find candidates in the transaction

$$t = (157, \{A, C, E, F, G, J\}).$$

The candidates are represented by the hash-tree of Figure 1. We settle all subsets of t beginning with C just at the root, as there is no appropriate link to follow. Note also that the leaf of candidate $\{A, E, G\}$ also contains the itemset $\{A, E, M\}$ because G and M have the same hash value (6 mod 6 = 0 = 12 mod 6). Note that $\{A, E, G\}$ is supported by t, while $\{A, E, M\}$ is not.

Hash-trees can be implemented by arrays very simply. All hash-tables of the tree are described by the *hash array*. The jth hash-table corresponds to the jth row of the hash array, and the lth element is a pointer we have to follow for those items that have hash value l. Zero value means that there is no link for this hash value. The same index in the *type-array*, which is a bit array, stores whether the pointed node is an internal node or a leaf (1 if leaf, 0 if internal node). For every leaf, the *leaf-number vector* gives the number of candidates the leaf stores, the *leaf array* contains the candidates, and the *counter array* stores the counters. Both in the leaf-number vector and in the leaf array, the jth row corresponds to the jth leaf. These structures for the hash-tree given in Figure 1 are described below.

In our implementation of the APRIORI method, we made a small modification: the root of the hash-tree is always a hash-table. This does not require more memory (sooner or later the root is altered to a hash-table) but it can speed up the search if the *leaf_max_size* parameter is not set properly (i.e., to a value too high).

In the next section, we present a trie-based solution that can be a viable alternative for hash-trees. As we will see, trie search leads us exactly to those leaves where supported candidates are stored, and hence, no second tests are needed.

Hash Array

1.	2	0	0	0	4	0
2.	0	0	1	0	3	0
3.	3	0	0	0	0	2

Type Array

0	-	-	-	1	-
-	-	1	-	0	-
1	-	-	-	-	1

	Leaf-Number vc.	Leaf Array		Counter Array	
1.	1	ACD		0	
2.	1	AEL		0	
3.	2	AEG	AEM	0	0
4.	1	KMN		0	

Figure 2. Implementation of a hash-tree.

5. DETERMINING SUPPORT WITH A TRIE

The data structure *trie* was originally introduced to store and efficiently retrieve words of a dictionary (see, for example, [17]). A trie is a rooted (downward), directed tree like a hash-tree, however, here we make no differences between the type of inner nodes and leaves. The root is defined to be at depth 0, and a node at depth j can point to nodes at depth $j+1$. A pointer is also called *link*, which is labelled by a letter. There exists a special letter * which represents an end character. If node n_i points to node n_j, then we call n_i the parent of n_j, and n_j a child node of n_i.

Every leaf l represents a word which is the concatenation of the path's letters that starts from the root and ends at l. Note that if the first k letters are the same in two words, then the first k steps on their paths are the same as well. A trie that stores the words *mile, milk, tea, tee, teeny* can be seen in Figure 3.

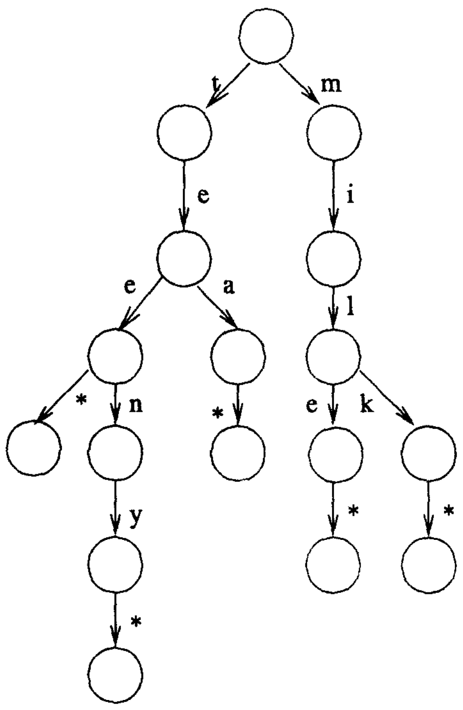

Figure 3. A trie containing five words.

To find whether a word (concatenation of letters plus the * character) is stored in the tree, we have to start from the root node, and move forward by applying its letters in order. If we cannot move forward because there is no link with the required label, then the word is not included in the dictionary.

Inserting a word into a trie is simple. We have to start from the root node as if we would like to find our word. If we get to a node which has no link labelled with the next letter L of the word, then we create a new node and a link pointing to it, whose label is L. We repeat this procedure till we reach the end of the word.

Tries are suitable to store and retrieve not only words but any finite ordered sets. In this setting, a link is labelled by an element of the set, and the trie contains a set if there exists a path where the links are labelled by the elements of the set, in increasing order.

In our data mining context, the alphabet is the (ordered) set of all items $\mathcal{I}$. A candidate k-itemset,

$$C = \{i_1 < i_2 < \cdots < i_k\},$$

can be viewed as the word $i_1 i_2 \cdots i_k$ composed of letters from $\mathcal{I}$. We do not need the * symbol, because at every phase we know the depth of nodes that represent candidates.

Figure 4 presents a trie that stores the same candidates as the hash-tree of Figure 1. Numbers in the nodes serve as identifiers and will be used in the implementation of the trie.

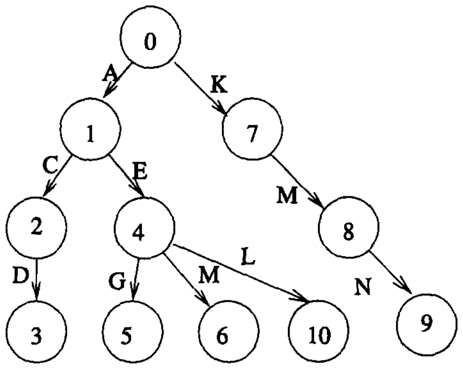

Figure 4. A trie containing five candidates.

The general strategy of support-calculation remains the same but we use a trie rather than a hash-tree to store candidates. When all the candidates have been inserted, we process transactions as before. For a transaction $t = (t\_id, I)$, we take all ordered k-subsets X of I and search for them in the trie structure. If X is found (as a candidate), then we increase the support count of this candidate by one. Here, again, we do not generate all k-subsets of I, but rather we perform early quits if possible. More precisely, if we are at a node at depth d by following the l^{th} item link, then we move forward on those links that have the labels $i \in I$ with index greater than l, but less than $|I| - k + d + 1$.

Note that in trie search there is no additional need of comparing X to the candidate corresponding to a leaf. If we arrive at a leaf (equivalently: a node at depth k), then the candidate stored here must be X itself.

Next, we outline the implementation of tries we employed in our program. A trie is represented by a *linknumber vector* and by two arrays, the *itemarray* and the *nodearray*. The j^{th} element of the vectors and j^{th} row of the arrays belong to the j^{th} node. The j^{th} element of the linknumber vector stores the number of links the j^{th} node has. The length of the j^{th} row of the two arrays is equal to the number of links the j^{th} node has. The l^{th} element of a row in the itemarray shows the label of the l^{th} link, and the same element in the nodearray stores the node it links to. Obviously, a counter for every leaf is also needed which will store the number of transactions that contained the itemset represented by the leaf.

Figure 5 shows the structure of the trie of Figure 4. The unique identifiers of the items can be found at the top.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
unique identifiers:	0	1	2	3	4	5	6	7	8	9	10	11	12	13

	Linknr. Vector	Parent Vector	Item-Array	Node-Array
0.	2	–	0 5	1 7
1.	2	0	2 4	2 4
2.	1	1	4	3
3.	0	2	|	|
4.	3	1	6 11 12	5 10 6
5.	0	4	|	|
6.	0	4	|	|
7.	1	0	9	8
8.	1	7	12	9
9.	0	8	|	|
10.	0	4	|	|

Figure 5. Implementation of the trie of Figure 4.

Hash-trees only store candidates; frequent itemset are stored in another data structure (such as a lexicographically ordered list). In contrast, we intend to use tries to store frequent itemsets as well. This way association rule generation becomes much faster, and the negative border of the frequent itemsets (see [6] for more details) can easily be generated. This is quite advantageous in on-line rule mining, or in rule maintenance.

Upon determination of the support of the candidates, infrequent candidates have to be pruned. This can be done by freeing those rows in the item and node array that have low support. If we delete a node, the link that points to it needs to be deleted too. For the efficient implementation of deleting nodes, we need a *parent vector* that holds a reference to the parent of every node. We can generate the parent vector during the trie building phase; when a new node is added, we set its parent value immediately.

Deleting infrequent nodes without causing "gaps" in the item- and nodearray can be solved easily by a simple scan of the nodes. For this we need two index variables both with initial value the index of the root node. The first variable is increased after every step while the second only when a frequent node is found. If the node at the position of the first variable is frequent, then the node has to be moved to the place represented by the second variable and we have to alter the links that point to it to the new place. Note that to move a node to a new position does not mean copying elements in the representing row, but only setting a pointer to the new position. If the node is infrequent, then it has to be deleted together with the link that points to it.

Thus, we use essentially the standard procedures for maintaining the trie. The routine for infrequent node deletion appears to be new, and hence, a pseudocode is given below.

```
procedure deleting_infrequent_nodes
begin
  index1=1;
  index2=1;
  while (index1<number_of_nodes)
  {
    if counter[index1]<min_supp*|$\mathcal{T}$| then  //infrequent
    {
      row index2 of itemarray,nodearray,counter,linknr
           <- row index1 of itemarray,nodearray,counter,linknr;
      pointer_modification(parent_vector[index1],index2);
```

```
        index2++;
      }
      else
      {
        free(line index1 of itemarray,nodearray,counter,linknr);
        delete_link(parent_vector[index1],index1);
      }
      index1++;
    }
  }
```

5.1. An Improvement

Here, we show how the time of finding supported candidates in a transaction can be greatly reduced by keeping a little extra information. The point is that we often perform superfluous moves in trie search in the sense that there are no candidates in the direction we are about to explore. To illustrate this, consider the following example. Assume that after determining frequent four-itemsets only candidate $\{A, B, C, D, E\}$ was generated, and Figure 6 shows the resulting trie.

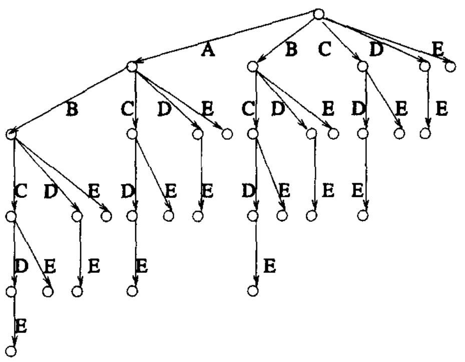

Figure 6. A trie with a single five-itemset candidate.

If we search for five-itemset candidates supported by the transaction $(t\_id, \{A, B, C, D, E, F, G, H, I\})$, then we must visit every node of the trie. This appears to be unnecessary since only one path leads to a node at depth 5, which means that only one path represents a five-itemset candidate. Instead of visiting merely six nodes, we touch 32 of them. At a node we also have to decide which link to follow, which can greatly affect running time if a node has many links.

To avoid this superfluous travelling, at every node we store the length of the longest directed path that starts from it. When searching for k-itemset candidates at depth d, we move downward only if the maximal path length at this node is $k - d$. Storing counters needs memory, but as experiments proved, it can seriously reduce search time for large itemsets. Note that this trick cannot be applied to hash-trees because a leaf at depth d does not necessarily store d-itemset candidates.

We have seen two different data structures for storing and retrieving candidates. In the next section, we give a comparison of their merits/disadvantages in the context of finding frequent itemsets.

6. COMPARISON OF THE DATA STRUCTURES

Tries appear to have three advantages over hash-trees.

Speed: Under certain circumstances, we can find faster the candidates supported by a transaction, if candidates are stored in a trie. The main reason for this speedup is that we do not have to perform an additional second checking phase. Trie search leads us exactly to those leaves that are representations of supported candidates.

Simplicity: We will see that trie makes candidate generation much simpler, because it is very easy to find those k-itemset pairs whose first $k-1$ items are the same.

Operation: A hash-tree needs to be fine tuned by two parameters for better performance (table size, *leaf_max_size*). Parameters that are suitable for one dataset may give poor results with another or with a different support threshold value. In contrast, the memory requirement and speed of the trie does not depend on parameters. It is, therefore, simpler for the user to work with tries; there is no need to understand the underlying data structures.

6.1. Retrieval Speed

The speed of the candidate retrieval process of the two approaches (hash-trees vs. tries) cannot be compared easily at a theoretical level. However, here we try to find reasons why tries turned out to be faster than hash-trees at certain support ranges. To offer evidence, in Section 7, we give experimental results on a real-life dataset with different characteristics.

In a hash-tree, we find the candidates that are supported by a transaction in two steps. First, we find the leaf that stores the potential candidate, and then we decide by directly comparing X to the candidates Y residing at the leaf. We reach a leaf by calculating hash values of items in the transaction. We calculate at most k hash values, where k is the size of the candidates. In the second step, we have to make at most n comparisons for each potential candidate to decide whether it is supported, where n is the number of items in the transaction.

When we search for a subset X of t in a trie, then arrival to a leaf means that X is actually a candidate. No second step (comparing X to Y) is needed. To arrive to a leaf, we have to traverse k pointers. A step along a link takes longer than jumping in a hash-tree where just a hash value is calculated. This is due to the fact that link labels are stored in an ordered list, and to decide which link to follow, we have to make $\lceil \log_2 l_i \rceil$ comparisons, where l_i is the number of children of the i^{th} node.

We can say that if candidate Y is supported by the transaction, then we make at most n comparisons, and we count k hash values in a hash-tree, and we make about $\sum_{i=1}^{k} \lceil \log_2 l_i \rceil$ comparisons when a trie is employed. However, with a hash-tree we have to check if all the other candidates in the leaf are supported by t or not. This can greatly increase the transaction processing time. Also, note that calculating a hash value always takes longer than evaluating a link comparison.

We do not see an easy way to give theoretical comparison of the time demand of the two approaches. It is obvious, nevertheless, that we can reduce time of finding all supported candidates in a hash-tree if we reduce the number of the false candidates. False candidates are due to collisions in the hash-tables, and we know that the chance of collisions can be reduced by increasing the size of the hash-table. However, increasing the table leads to higher memory need. We give some experimental results in Section 7 on the dependence of the behavior of hash-trees on the table sizes and leaf sizes. Also, we compare their performance to that of tries.

6.2. Simplicity

We recall the method of generating $k+1$-itemset candidates in APRIORI: first, we find two frequent k-itemsets whose first $k-1$ items are the same, and we join them. Next, we check if all the k subsets of the resulting X are frequent. If all subsets are frequent, then X becomes a candidate.

A useful feature of trie is that for two itemsets whose first $k-1$ items are the same, their representing paths in the tree agree up to the first $k-1$ nodes. Since we store frequent itemsets in a trie, candidate generation can easily be done by finding those nodes at depth $k-1$ that point to at least two leaves. This can be achieved by a single scan of a part of the nodes.

6.3. Ease of Use

Interest in data mining methods is spreading like brushfire. It is, therefore, important to have efficient and easy-to-use algorithms for the major data mining tasks. Here the second requirement is also important. We should not burden the users, if possible, with the inner technical details of our algorithms, such as data representation, speed/memory trade-offs, etc. We should merely expect users to understand the purpose and the outcome of data mining. In general, they do not care about and they do not want to deal with the inner operation of the algorithms.

They want to find valid association rules without understanding how a hash-tree works. We recall that a hash-tree has two parameters for tuning its performance. The first one is the size of hash-tables, and the second is the limit to the number of candidates a leaf can store. In contrast, tries offer a self-adjusting approach. They evolve naturally, without the need of setting parameters.

Understanding inner workings and learning the parameters of different data structures is not an unpleasant task for a researcher. For other users, the simplicity of the operation is vital. The lack thereof may be the main obstacle of using data mining algorithms.

7. EXPERIMENTAL RESULTS

We analyzed web log data of a popular Hungarian news site to find frequently visited sets of articles. The original database was a text file of size 2.8 Gbytes. After removing redundant attributes and restructuring the file to fit the market-basket model, we ended up with a file of 30 Mbytes. The number of items was around 40000, and the database contained approximately 9900000 transactions. Since frequent itemsets mining is of interest in much bigger databases, in our experiments we used a computer with modest capabilities. It had 128 Mbytes main memory, and PII-350 Mhz processor with Linux operating system (kernel version 2.4.10) on it.

Tables 1–3 show how the running time and memory need of a hash-tree changes with the two parameters (table size or modulus, *leaf_max_size*). Time is given in seconds, memory usage in Kbytes. It can be seen that if the parameters are far from the optimal, then the time and memory need reflects very poor results. Test results also prove that the optimal parameters of hash-tree may change even if the same database is used, but with a different support threshold. What is more, good values for a certain support threshold can provide poor result with other support thresholds.

Comparison of memory and time consumption of the three data structures can be seen in Table 4. The notation *trie_impr* stands for the improved trie where we store the length of the longest path for every node. Here we only present the results of hash-trees with optimal (or very close to the optimal) parameters.

If the support threshold is high, then hash-trees can be tuned to outperform tries both in memory requirement and in speed. However, if one of the parameters is not near to the optimal value, then hash-trees give poor result in at least one respect.

At lower support threshold, tries outperform hash-trees as far as speed is concerned. The memory need of fine-tuned hash-trees is still smaller than the memory requirement of tries, but the range of parameters where this holds is narrower. Notice that at min_supp = 0.003, if leaf threshold shrinks from 500 to 100, then the memory need increases from 665 Kbytes to 11.1 Mbytes which is a 17-fold increase.

The last figure shows how fast the improved trie approach finds candidates, as a function of the size of candidates (min_supp = 0.002 was used in this test). It is apparent that with *trie_impr* the time decreases steeply, as the candidate size passes six. The basic algorithm performs much worse in that range.

Table 1. Test results with a hash-tree at min_supp = 0.01.

Memory need (Kbytes).

Modulus	Leaf Threshold								
	5	10	25	60	100	300	500	2500	10000
10	19	19	19	19	18	17	17	17	17
25	25	25	24	22	19	17	17	17	17
100	51	50	44	20	18	18	18	18	18
500	157	145	105	22	22	22	22	22	22
2500	645	584	398	42	42	42	42	42	42
10000	2477	2232	1496	115	115	115	115	115	115

Running time (sec.).

Modulus	Leaf Threshold								
	5	10	25	60	100	300	500	2500	10000
10	11078	1351	529	402	384	388	388	388	389
25	1180	464	363	318	325	336	336	337	337
100	315	291	277	270	277	277	277	277	277
500	263	256	249	256	257	257	257	257	257
2500	257	250	242	253	253	253	253	253	253
10000	262	255	246	257	257	257	257	257	257

Table 2. Test results with a hash-tree at min_supp = 0.005.

Memory need (Kbytes).

Modulus	Leaf Threshold								
	5	10	25	60	100	300	500	2500	10000
10			145	145	145	145	145	143	143
25		154	154	154	154	151	149	143	143
100		249	247	238	223	151	145	145	145
500		595	569	498	367	149	149	149	149
2500		2090	1904	1458	929	169	169	169	166
10000		7475	3739	5011	3017	242	242	242	242

Running time (sec.).

Modulus	Leaf Threshold								
	5	10	25	60	100	300	500	2500	10000
10			13396	1986	1760	1613	1652	2303	2318
25		6652	1774	1135	1111	1165	1207	1670	1675
100		876	791	773	778	1050	1014	1050	1052
500		750	733	721	686	770	770	770	770
2500		756	736	710	654	680	684	695	695
10000		833	753	711	658	682	679	680	680

Table 3. Test results with a hash-tree at min_supp = 0.003.

Memory need (Kbytes).

Modulus	Leaf Threshold								
	5	10	25	60	100	300	500	2500	10000
10				566	566	566	566	566	564
25			575	575	575	575	575	567	564
100		719	719	718	715	685	645	566	566
500		1609	1585	1541	1462	768	577	572	572
2500		4839	4652	4206	3677	736	592	592	592
10000		15871	15135	13407	11413	1140	665	665	665
25000		37917	36082	31791	26868	1945	812	812	812

Running time (sec.).

Modulus	Leaf Threshold								
	5	10	25	60	100	300	500	2500	10000
10				23369	11453	6566	6570	6816	9932
25			10355	4707	4002	3920	3953	6066	6877
100		2308	2051	1963	1960	2111	2371	3767	3767
500		2071	2035	2020	2019	1761	2118	2251	2252
2500		2245	2205	2173	2128	1455	1703	1835	1835
10000		2261	2218	2181	2132	1421	1670	1801	1807
25000		2261	2217	2179	2129	1400	1664	1810	1810

Table 4. A comparison of tries and hash-trees.

Data Structure	min_supp	Parameters	Memory Need	Running Time
trie	0.01		35	325
trie_impr	0.01		37	317
hash-tree	0.01	500, 60	22	256
hash-tree	0.01	2500, 25	398	242
trie	0.005		287	629
trie_impr	0.005		310	562
hash-tree	0.005	500, 500	149	770
hash-tree	0.005	2500, 100	929	654
trie	0.003		1134	1120
trie_impr	0.003		1223	904
hash-tree	0.003	10000, 500	665	1670
hash-tree	0.003	250000, 300	1945	1400

8. CONCLUSION

Hash-trees are used widely in frequent itemset mining algorithms. In this paper, we proposed an advanced version of trie for solving this important data mining task. Experiments proved that the performance of tries is close to the performance of hash-trees with optimal parameters at high support threshold; however, at lower threshold the trie-based algorithm outperforms the hash-tree based ones.

Moreover, tries are particularly suitable for efficient implementation of candidate generation because pairs of items that produce candidates have the same parents. Thus, candidates can be easily obtained by a simple scan of a part of the trie.

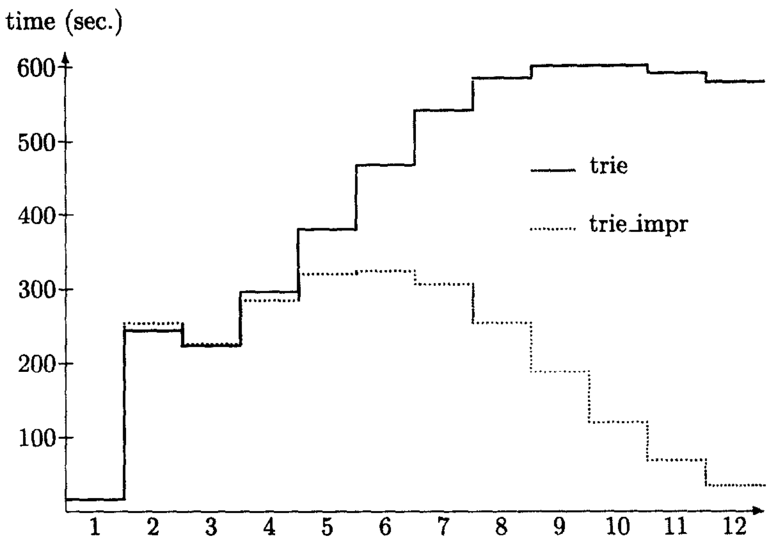

Figure 7. Candidate size.

Last, maybe the most important feature of tries is their self-adjusting behavior. We showed that if the parameters of the hash-tree are not close to the optimal values, then the hash-tree based algorithm can give very poor results. Consequently, efficient use of the hash-tree based algorithm requires users to understand the technical details. In contrast, no tuning is needed for tries, and hence, the algorithm is easier to use.

REFERENCES

1. R. Agrawal, T. Imielinski and A. Swami, Mining association rules between sets of items in large databases, In *Proc. of the ACM SIGMOD Conference on Management of Data*, pp. 207–216, (1993).
2. R. Agrawal and R. Srikant, Fast algorithms for mining association rules, *The International Conference on Very Large Databases*, 487–499, (1994).
3. M.S. Chen, J.S. Park and P.S. Yu, An effective hash based algorithm for mining association rules, In *Proc. of ACM SIGMOD International Conference Management of Data*, (1995).
4. S. Brin, R. Motwani, J.D. Ullman and S. Tsur, Dynamic itemset counting and implication rules for market basket data, *SIGMOD Record (ACM Special Interest Group on Management of Data)* **26** (2), 255–264, (1997).
5. A. Sarasere, E. Omiecinsky and S. Navathe, An efficient algorithm for mining association rules in large databases, In *Proc. 21st International Conference on Very Large Databases (VLDB)*, Zurich, Switzerland; Gatech Technical Report No. GIT-CC-95-04, (1995).
6. H. Toivonen, Sampling large databases for association rules, *The VLDB Journal*, 134–145, (1996).
7. R. Srikant and R. Agrawal, Mining generalized association rules, In *Proc. of the 21st International Conference on Very Large Databases (VLDB)*, Zurich, Switzerland, (1995).
8. J. Han and Y. Fu, Discovery of multiple-level association rules from large databases, In *Proc. of the 21st International Conference on Very Large Databases (VLDB)*, Zurich, Switzerland, (1995).
9. Y. Fu, Discovery of multiple-level rules from large databases, (1996).
10. D.W.L. Cheung, J. Han, V. Ng and C.Y. Wong, Maintenance of discovered association rules in large databases: An incremental updating technique, *ICDE*, 106–114, (1996).
11. D.W.L. Cheung, S.D. Lee and B. Kao, A general incremental technique for maintaining discovered association rules, *Database Systems for Advanced Applications*, 185–194, (1997).
12. S. Thomas, S. Bodagala, K. Alsabti and S. Ranka, An efficient algorithm for the incremental updation of association rules in large databases, 263.
13. N.F. Ayan, A.U. Tansel and M.E. Arkun, An efficient algorithm to update large itemsets with early pruning, *Knowledge Discovery and Data Mining*, 287–291, (1999).
14. R. Agrawal and R. Srikant, Mining sequential patterns, In *Proc. 11th Int. Conf. Data Engineering, ICDE*, (Edited by P.S. Yu and A.L.P. Chen), pp. 3–14, IEEE Press, (1995).
15. R. Srikant and R. Agrawal, Mining sequential patterns: Generalizations and performance improvements, Technical Report, IBM Almaden Research Center, San Jose, CA, (1995).
16. H. Mannila, H. Toivonen and A.I. Verkamo, Discovering frequent episodes in sequences, (1995).
17. D.E. Knuth, *The Art of Computer Programming, Volume 3*, Addison-Wesley, (1968).