A Large-Itemset-Based Index Structure for Supporting Personalized Information Filtering on the Internet*

Ye-In Chang, Tsu-I Chen and Jun-Hong Shen
Dept. of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C
E-mail: changyi@cse.nsysu.edu.tw

ABSTRACT
The World Wide Web creates many new challenges for information retrieval. Information Filtering (IF) can find good matches between the web pages and the users' information needs. In an information filtering system, users are associated with profiles that describe what they need, while data are represented in the same form of user profiles. Comparing data with profiles, the users who are interested in the data are identified and informed. Therefore, a critical issue of the information filtering service is how to index the user profiles for an efficient matching process. In this paper, first, we propose a *count-based tree* method, to reduce the large storage space as needed by Yan and García-Molina’s tree method. Next, by applying the technique for mining association rules, we propose a large-item set-based method, the *count-major large demset*, to further reduce the storage space. From our simulation results, the cost of storage space of our methods is less than that of the tree method.

**Keywords:** index, information filtering, personalization, profile, web usage mining.

1. INTRODUCTION
Owing to the booming development of the WWW, it creates many new challenges for information filtering [2, 4, 5]. Information Filtering (IF) is an area of research that develops tools for discriminating between relevant and irrelevant information. In an information filtering system, users first give descriptions about what they need, i.e., user profiles, to start the services [3, 7]. A profile index is built on these profiles. A series of incoming web pages will be put into the matching process. Each incoming web page is represented in the same form of the user profile. In this way, the users who are interested in an incoming web page can be identified by comparing the descriptions of the web page with each user profile [8, 9]. This concept has been applied to various information system on the Internet, e.g., SIFT [10].

A critical issue of the information filtering service is how to index the user profiles for an efficient matching process. Using index structure on information filtering service, the users’ profiles can be expressed in the *Boolean* model. In the Boolean model, the user specifies keywords that he (or she) wants in documents received; a document is similarly represented [9]. The set of keywords appeared in the document represents its high frequency in the document. In an information filtering service, the documents are recommended to the users if all keywords in their profiles appear in the document. We look up the words one by one, and stop as soon as we find a word not in the document. If all the words appear in the document, the profile matches. In [9], Yan and García-Molina have proposed four methods based on the Boolean model. Among them, the tree structure provides the best performance in terms of the storage space. Instead of using the Boolean model, the users’ profiles can be expressed in the *vector space* model. In the vector space model [8], users’ profiles and documents are identified by keywords. A profile is similarly represented. For a document-profile query, a similarity measure (such as the dot product) can be computed to determine how “similar” the two are.

Since the matching process in the Boolean model, in general, is much simpler and requires less time than that in the vector space model, in this paper, we adopt the Boolean model. Based on the tree method proposed by Yan and García-Molina [9], there are many duplications of keywords appearing in the tree structure. It needs large storage space to store the information, although the matching process is simple. Therefore, in this paper, first, we propose a *count-based tree* method, which takes the count of each keyword into consideration, to reduce the large storage space as needed by the tree method [9]. Next, by applying one of the techniques for mining association rules, the *Apriori* algorithm [1], we propose a new method. Basically, an association rule is that given a database of sales transactions, it is desirable to discover the important associations among items such that the

---

*This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-92-2213-E-110-001.
Table 1: Example 1 of profiles

<table>
<thead>
<tr>
<th>Profile</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>b c d e g h i</td>
</tr>
<tr>
<td>P₂</td>
<td>a b c e g i j</td>
</tr>
<tr>
<td>P₃</td>
<td>a b c d e g h i</td>
</tr>
<tr>
<td>P₄</td>
<td>c d e f g h</td>
</tr>
<tr>
<td>P₅</td>
<td>a c d f g i j</td>
</tr>
</tbody>
</table>

The presence of some items in a transaction will imply the presence of other items in the same transaction. In the method, we make use of the concept of the large itemset which is used in mining association rules, where a large itemset is represented by a set of items appearing in a sufficient number of transactions. The method is called the count-major large itemset method. In this method, we first cluster profiles with similar interests into the same group [6, 11]. Next, for each cluster, we apply the mining association rules techniques to help us construct the index structure. From our simulation results, the cost of storage space of our methods is less than that of the tree method.

The rest of the paper is organized as follows. Section 2 presents a survey of the tree method. Section 3 presents the count-based tree method. Section 4 presents the proposed large-itemset-based method. In Section 5, we study the performance. Finally, Section 6 states the conclusions.

2. YAN AND GARCIA-MOLINA’S TREE METHOD

In Yan and Garcia-Molina’s tree method [9], a tree derived from a set of profiles is called an index tree. In such a tree, the internal node keeps a keyword and the leaf node records a profile. The root is a pseudo node. For the profiles shown in Table 1, Figure 1 shows the related index structure for the tree method. As shown in Figure 1, the internal nodes in a tree structure can be shared by their descendants. Moreover, the same keyword may appear in several internal nodes; while a profile can only appear in one leaf node. In the index tree, each profile Pᵢ is represented by a path starting from the pseudo root node, passes a consecutive sequence of internal nodes, and ends at a leaf node. The keywords on the internal nodes passed by this path, are the same as the keywords specified in Pᵢ. Therefore, a profile can be inserted by traversing and creating the corresponding path. To find the matches for a web page, a traversal of the index tree from the root has to be conducted. That is, all the paths, which may cover a set of profiles, have to be examined. If the keyword in an internal node is not contained in the set of keywords extracted from a web page, all the paths containing this internal node will not cover any matched profiles.

3. THE COUNT-BASED TREE METHOD

Take the profiles shown in Table 1 as an example, the tree method [9] stores 33 keywords (including the root) as shown in Figure 1. To reduce the large storage space as needed by the tree method [9], we take the count of each keyword into consideration. Before constructing the count-based tree, we count the number of each keyword in those profiles. Moreover, those keywords are sorted in the descending order of their corresponding counts. For the profiles shown in Table 1, the order of those keywords is [c, d, e, g, i, a, b, f, h, j] and the related counts are 5, 4, 4, 4, 4, 3, 3, 3, 3, 3, and 3, respectively. Next, we sort the keywords in each profile according to the new order of keywords, resulting in the new profiles shown in Table 2. Then, according to those profiles shown in Table 2, based on the same steps in constructing the index tree [9], we construct the count-based tree shown in Figure 2, which needs only 27 keywords, as compared with 33 keywords based on the tree method.

4. THE LARGE-ITEMSET-BASED METHOD

In this section, first, we give a survey of the Apriori algorithm. Then, we present the method based on the technique of mining association rules: the count-major large itemset method.

A Survey of the Apriori Algorithm

In the Apriori algorithm [1], a set of items is called an itemset and k-itemset is an itemset that has k items. The algorithm calculates the support s of the itemset in the transaction set D, if s% of transactions in D contain it. An itemset is large if its support is above some user-defined minimum support threshold.

In the Apriori algorithm, it constructs a candidate set of
largest itemsets, counts the number of occurrence of each candidate itemset, and then determines large itemsets based on a pre-determined minimum support. The Candidate k-itemsets \( (C_k) \) is generated by a cross product of the Large \((k-1)\)-itemsets \( (L_{k-1}) \) with itself. The large \( k \)-itemsets \( (L_k) \) consist of only the candidate \( k \)-itemsets with sufficient support; that is, the count of the occurrence of \( k \)-itemsets in the transaction database is no less than a threshold, the minimum support. This process is repeated until no new candidate itemsets is generated.

Each profile can be regarded as a transaction, and each keyword can be regarded as an item. Therefore, our input, the profiles as shown in Table 1, can be regarded as the transaction database in which Profile and Keywords are replaced with Transaction and Items, respectively. Given the minimum support = 80\% (i.e., count > = 5 * 0.8), the resulting large itemset for the input shown in Table 1 contains \( L_1 = \{ \{ e \}, \{ d \}, \{ e \}, \{ g \}, \{ i \} \} \), and \( L_2 = \{ \{ e, d \}, \{ e, e \}, \{ e, g \}, \{ e, i \} \} \), where \( k = 2 \).

The Count-Major Large Itemset Method

Although we adopt the concept of the Apriori algorithm, we modify the way to choose the minimum support, which is predetermined and static during the processing of the original Apriori algorithm. In our revised Apriori algorithm, the minimum support is dynamically decided, which is the sum of the support of each candidate itemset divided by the number of the candidate itemsets. That is, the formula of the minimum support is as follows: \[ \sum_{i=1}^{C_n} \text{the support of } C_i | C_n \text{, where } C_n \text{ is represented the candidate itemset in the } n \text{'th round}. \] We use an example given in Figure 3 to illustrate the way to decide the support. In Figure 3, the minimum support for \( C_1 \) is calculated as follows: Support = \( \frac{(1+3+3+3+3)}{5} = 2.6 \). Therefore, we have \( L_1 = \{ \{ B \}, \{ C \}, \{ D \}, \{ E \} \} \).

In the Apriori algorithm, if the minimum support is too small, there may be too many candidate itemsets becoming the large itemsets, which takes much time to get the large itemsets. Therefore, we let the minimum support be dynamically decided, since we just want to get the large itemset that has the largest support among all large itemsets. Moreover, instead of using the itemset with the largest length \( k \) in our method, the final result is called the Largest-Common-Keywords (LCK). The policies for getting the LCK are as follows:

1. If the maximum support of elements in \( L_n \) is smaller than that in \( L_{n-1} \), we choose the element with the maximum support in \( L_{n-1} \) to be the LCK. For example, in Figure 3, because the maximum support of elements in \( L_2 \) is smaller than the maximum support of elements in \( L_1 \), we can randomly choose one element from \( L_1 \). Therefore, in this case, we do not need to generate \( C_3 \) from \( L_2 \).

2. If the maximum support of elements in \( L_n \) is equal to that in \( L_{n-1} \), we choose the element with the maximum support in \( L_n \) to be the LCK.

By using our revised Apriori algorithm (denoted by function \textit{RevisedAlg}), we can get the LCK from all profiles first. After getting the LCK, we divide those profiles into two parts according to the result of the LCK. One part contains the LCK and the other part does not contain the LCK. Next, those profiles in the two parts keep on getting the LCK by using function \textit{RevisedAlg}, respectively. We can use each result of the LCK to construct the tree. This process is repeated until no keywords are in the profiles. Take the profile shown in Table 1 as an example, it stores 24 keywords as shown in Figure 4.

Next, our complete algorithm is shown in Figure 5. Form the root of the tree, our input data is \( PSet \) which contains those profiles \( \{ P_i, i = 1 \cdots n \} \). The root of the tree is a pseudo node. If the size of \( PSet \) is larger than 1, we use function \textit{RevisedAlg} (line 8) in procedure \textit{Count-Major}. On the other hand, if the size of \( PSet \) is equal to 1, we add the keywords in the profile \( (kSet) \) and the identifier of profile \( P_i \) to the tree (lines 31 to 37). After getting the LCK of the input data \( PSet \), we divide those profiles into two parts. One part that is named
newPSet1 contains the LCK, and the other part that is named newPSet2 does not contain the LCK. For the newPSet1 part, we will remove LCK from those profiles which contain LCK. If newPSet1 and newPSet2 are not φ, each of them will do the same thing as PSet until all profiles are inserted into the tree.

Let’s use an example shown in Table 1 to illustrate those steps for constructing the count-major large itemset tree. By using our function RevisedAlg, we can get the LCK = {c} at the first time, since we have L1 = {c} with support = 5 and L2 = {cd} with support = 4.

After getting the LCK {c}, we add {c} to the tree by using procedure Insert (line 10). Next, we check each of the profiles in PSet, whether it contains the LCK {c} (lines 11 to 21). Since all the profiles contain the

Figure 3: An example to get the LCK

Figure 4: The data structure for the count-major large itemset method

Figure 5: Procedure Count-Major for the count-major large itemset method
Fig. 6: The third step: (a) the profile \( P_2 \) does not contain the \( LCK = \{d\} \); (b) those profiles contain the \( LCK = \{d\} \).

Table 3: Parameters and their settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>500, 1000, 1500, 2000</td>
</tr>
<tr>
<td>( D )</td>
<td>50, 100, 150, 200</td>
</tr>
<tr>
<td>( K )</td>
<td>3, 4, 5, 6, 7</td>
</tr>
<tr>
<td>( Q )</td>
<td>0%, 20%, 40%, 60%, 80%, 100%</td>
</tr>
</tbody>
</table>

\( LCK \) \{c\}, all the profiles remove the keyword \{c\} as shown in the left part of Fig. 6 and are recorded in the newPSet1 part. Moreover, the newPSet2 part is \( \phi \).

Because newPSet1 is not \( \phi \), we call procedure Count - Major again. For this time, we get the \( LCK = \{d\} \), since we have \( L_1 = \{d\} \) with support = 4 and \( L_2 = \{ai\} \) with support = 3. Similar to the previous step, we insert the \( LCK \) \{d\} to the tree by using procedure Insert and divide those profiles into two parts, one part does not contain the \( LCK \) \{d\} as shown in Fig. 6-(a), and the other part contains the \( LCK \) \{d\} as shown in Fig. 6-(b).

Because there is only one profile \( P_3 \) in newPSet2, we insert the keywords in \( P_3 \) and the identifier of \( P_2 \) to the tree. After that, newPSet2 is empty (lines 22 to 29). Next, we continue to handle the profiles that are represented by newPSet1 as shown in the lower part of Fig. 6-(b). Similar to the previous step, we get the \( LCK \) \{e, h\}.

5. PERFORMANCE

In this section, we compare our proposed methods with the tree method by simulation.

The Simulation Model

In this section, we generated synthetic profiles to evaluate the performance [9]. Four parameters are used in the generation of the synthetic profiles: \( N \), \( D \), \( K \), and \( Q \), where \( N \) is the number of profiles, \( D \) is the number of keywords, \( K \) is the length of the profile, and \( Q \) is the probability of the similarity among profiles. To simplify the study of the effect of the profile size on performance, all profiles have the same length; that is, \( K \) is fixed for all profiles. The keywords that all profiles choose are composed of the set of \( D \) keywords. Therefore, keywords in the first profile are chosen randomly from the set of \( D \) keywords. The first profile is called the “base profile”. In our assumption, the users with similarity interests are clustered into the same group. Therefore, in order to model the similarity among profiles, the similarity parameter \( Q \) controls how similar the new profile and the base profile are. That is, for each word in the new profile, there is a probability \( Q \) that it is the same as the corresponding word in the base profile. If it is not, then the keyword in the new profile is picked at random from the set of \( D \) keywords. There are no duplicated keywords in the profile. Hence, by varying \( Q \) from 0 to 1, we can control the similarity among the profiles. If \( Q \) is 0, the keywords in all profiles are randomly chosen from the set of \( D \) keywords.

Simulation Results

In our simulation, four parameters and their default settings are listed in Table 3. We will change only one of four parameters at one time. First, we define a base case with \( N = 1000 \), \( D = 100 \), \( K = 5 \), and \( Q = 80\% \). According to those parameters in the base case, a comparison of the number of keywords in the tree method and our methods for 100 executions on the average is shown in Table 4. From this result, we show that the tree method [9] needs more storage space to store the keywords than our methods. In the base case, the count-major large itemset method uses the smallest storage space to store the keywords among those methods. On the average, our methods can reduce about 40\% number of keywords as compared with the tree method.
Table 6: A comparison of the number of keywords (under the change of $D$)

<table>
<thead>
<tr>
<th>Methods / D</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>50</td>
<td>100</td>
<td>150</td>
<td>200</td>
</tr>
<tr>
<td>count-based tree</td>
<td>1112</td>
<td>151</td>
<td>185</td>
<td>1719</td>
</tr>
<tr>
<td>count-major large itemset</td>
<td>1112</td>
<td>151</td>
<td>185</td>
<td>1719</td>
</tr>
</tbody>
</table>

Table 7: A comparison of the number of keywords (under the change of $K$)

<table>
<thead>
<tr>
<th>Methods / K</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>523</td>
<td>845</td>
<td>151</td>
<td>185</td>
<td>2601</td>
</tr>
<tr>
<td>count-based tree</td>
<td>306</td>
<td>622</td>
<td>540</td>
<td>1085</td>
<td>1312</td>
</tr>
<tr>
<td>count-major large itemset</td>
<td>306</td>
<td>622</td>
<td>540</td>
<td>1085</td>
<td>1312</td>
</tr>
</tbody>
</table>

Next, we study the impact of four parameters, $N$, $D$, $K$, and $Q$, on the performance. The results are shown in Tables 5, 6, 7, and 8, respectively. In all the cases, the count-major large itemset method needs the smallest storage space among these methods. As $N$, $D$, or $K$ is increased, the number of keywords is increased in all the methods. As $Q$ is increased, the number of keywords is decreased in all the methods. That is, the larger the similarity among the profiles is, the more the common keywords among the profiles are. So, the storage space to store the keywords is reduced when the similarity is large. In particular, when $Q = 100\%$ in which each of the profiles is the same as each other, all those methods store 6 keywords.

6. CONCLUSIONS

In this paper, we have proposed the count-based tree method and the large-itemset-based method to improve the storage space as needed by Yan and Garcia-Molina’s tree method for indexing uses’ profiles. We adopt the idea of the Apriori algorithm to get the large itemset. However, we modify the way to decide the minimum support and the goal in the Apriori algorithm. After getting the large itemset by using our revised Apriori algorithm, we can construct the index structure. From our simulation, we have shown that our methods need smaller storage space to store the keywords than the tree method. How to construct the index structure in the case of the on-line processing is a future research topic.

References


