An Efficient Signature-based Strategy for Supporting Inexact Filtering in Information Filtering Systems

Ye-In Chang, Lee-Wen Huang, Jun-Hong Shen and Yi-Siang Wang

Dept. of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C
{E-mail: changyi@cse.nsysu.edu.tw}
{Tel: 886-7-5252000 (ext. 4334)}
{Fax: 886-7-5254301}

Abstract

To help users find the right information from a great quantity of data, Information Filtering (IF) sends information from servers to passive users through broadcast mediums, rather than being searched by them. To efficiently store many user profiles in servers and filter irrelevant users, many signature-based index techniques are applied in IF systems. By using signatures, IF does not need to compare each item of profiles to filter out irrelevant ones. However, because signatures are incomplete information of profiles, it is very hard to answer the complex queries by using only the signatures. Therefore, a critical issue of the signature-based IF service is how to index the signatures of user profiles for an efficient filtering process. The *inexact filtering* query is one kind of queries in the signature-based IF systems which is used to filter out the non-qualified data compared with queries. In this paper, we propose an *ID-tree index strategy*, which indexes signatures of user profiles by partitioning them into subgroups using a binary tree structure according to all of the different items among them. In an ID-tree, each path from the root to a leaf node is the signature of the profile pointed by the leaf node. Because each profile is pointed by only one leaf node of the ID-tree, there will be no collision in the structure. Moreover, only the different items among subgroups of profiles will be checked at one time to filter out irrelevant profiles for queries. Therefore, our strategy can answer the inexact filtering query with less number of accessed profiles as compared to Chen's signature tree strategy. From our simulation results, we have shown that our strategy can access less number of profiles to answer the queries than Chen's signature tree strategy for the inexact filtering.

(*Keywords*: Data partition, Inexact filtering, Information filtering, Signature, Similarity search)

1 Introduction

A tremendous amount of information is created and delivered over World Wide Web. This has made it increasingly difficult for individuals to control and effectively manage the potentially infinite flow of information. Ironically, just as more and more users are getting online, it is getting increasingly difficult to find information unless one knows exactly where to get it from and how to get it. Information filtering has made considerable progress in recent years, which denotes a family of techniques that help users find the right information items while filtering out undesired ones. In order to dispatch relevant information to users with respect to their specific information need, a user expresses his (her) interests in a number of long-term [18], continuously evaluated queries, called *profiles*. The users will then passively receive information filtered according to their profiles. In a wide range of applications, such as spam email filtering [8], film filtering [15], news filtering, and recommender systems for products, information filtering is playing an increasingly important role.

Information Filtering (IF) is an area of research that develops tools for discriminating between relevant and irrelevant information [14]. It deals with the delivery of items selected from a large collection that the user is likely to find interesting or useful and can be seen as a classification or a cluster task. Unlike information retrieval (IR) technique, IF deals with users who have long-term interests (information needs) that are expressed by means of user profiles, rather than with casual users whose needs are expressed as ad-hoc queries [2, 11]. To date, a large number of filtering techniques have been developed.

A signature is an access strategy for partial-match retrieval which meets many requirements of an office environment [4]. To apply on IR or IF systems, signatures can be used to save the storage of database and speed up the query performance. Signatures are hash-coded binary words derived from objects stored in the database. They serve as a filter for retrieval in order to discard a large number of non-qualified objects. Generally, all bits of the signature are cleared to *zero*, then a hash transformation is applied to the object's values to determine which bits are set to *one*.

Inexact filtering is one kind of searching strategies which are usually applied on the signature-based IF systems. An inexact filtering strategy is often used to filter out the

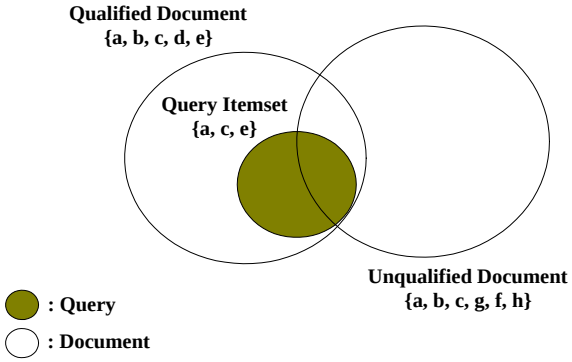


Figure 1: An example of the relationship between a query and database documents in the inexact match

non-qualified data compared with the queries. It can retrieve the database very quickly, but still needs a stage to refine the searched answers. The inexact filtering problem is extended from the inexact match. In the inexact match, to answer query q, we need to find all objects in the database which contain each item of q. However, in the inexact filtering, we need to filter out all objects in the database which are not contained in q. Therefore, the inexact filtering can be considered as the inverse problem of the inexact match. An example of the relationship between a query user and database documents in the inexact match is shown in Figure 1. In this example, there are 3 items, *i.e.,* a, c and e, contained in the query itemset. Because the document with items a, b, c, d and e contains the items of the query entirely, this document is qualified to be sent to the query user. On the other hand, the document with items a, b, c, g, f and h does not contain the items of the query entirely. Therefore, this document will not be sent to the query user.

The inexact filtering strategies are often applied on the content-based filtering systems. In these systems, the query is an incoming document which will be sent to database users according to their profiles. The formal definition of the inexact filtering problem is described as follow:

Definition 1. *Giving an incoming document, I_Doc, the inexact filtering problem is to find all database user profiles whose interest items are all contained in I_Doc. That is,* $P_i \subseteq I\_Doc$*, where* P_i *is the i-th profile in the database.*

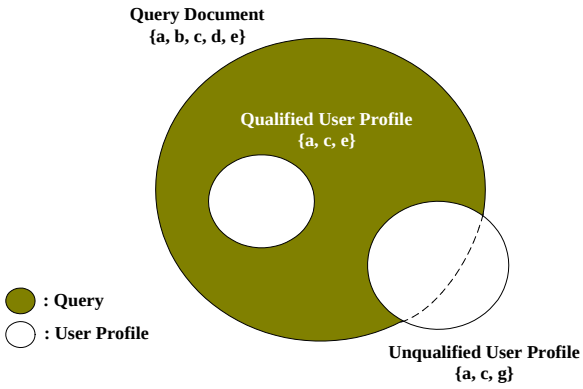


Figure 2: An example of the relationship between a query document and profiles in the inexact filtering

An example of the relationship between a query document and profiles in the inexact filtering is shown in Figure 2. In this example, there are 5 items, *i.e.*, *a, b, c, d* and *e*, contained in the query document. Because the profile with items *a, c* and *e* is entirely contained in the incoming document, this profile is qualified to receive the incoming document. On the other hand, the profile with items *a, c* and *g* is not entirely contained in the incoming document. Therefore, this profile will not receive the incoming document.

Different strategies for the inexact match have been proposed, such as the *Signature Files* [5, 10], the *Bit-Slice Files* [9], the *S-trees* [4], and the *Signature Trees* [3]. All of these strategies can be revised for the inexact filtering by changing the query process very easily. However, the inexact match means that all database objects containing the query need to be found. Chen has proposed *signature tree* strategy [3] to solve the inexact match problem efficiently. This strategy can also be revised to solve the inexact filtering. The signature tree is built by partitioning database profiles into groups according to the different items among them. Therefore, when a query is received, it needs only to check the existence of these different items in the query to filter irrelevant database profiles. However, because it extracts only one different item among each group at a time, the filtering effect may become poor when the query contains all of these different items.

Therefore, instead of using the signature tree to store user profiles, in this paper, we

propose a data structure, called the *Identifier Tree* (ID-tree), by extracting more number of different items among profiles to index them. Similar to the signature tree, each internal node of the ID-tree can be considered as a partition of profiles. However, there could be more than one item used to partition profiles in an ID-tree. Therefore, the profiles will be partitioned into groups more exactly. Moreover, because all profiles in the database are indexed globally, we only consider constructing the index structure for storing profile signatures in the case of the off-line processing. From our simulation, we have shown that the ID-tree strategy can reduce the most number of accessed profiles in the inexact filtering as compared with the signature tree strategy [3].

The rest of this paper is organized as follows. Section 2 gives a survey of several signature-based index strategies under the inexact filtering. Section 3 presents the proposed strategy, called the *ID-tree index strategy*. In Section 4, we study the performance and make a comparison of the proposed strategy with the signature tree strategy. Finally, we give the conclusion.

2 A Survey

In this section, we present a survey of some well-known signature-based index strategies which are often applied in information filtering systems. First, we will introduce some index strategies of the inexact filtering, including the *Signature File* [5, 10], the *Bit-Slice File* [9], the *S-Tree* [4] and the *Signature Tree* [3].

In the content-based information filtering systems, an incoming document is represented by a collection of keywords. Moreover, a user profile in the database is also represented by a sequence of distinct keywords. If an incoming document *matches* a profile, it must contain each keyword of the profile [6]. To save the storage of user profiles and filter documents much efficiently, profiles and documents are often represented as bitmaps in the filtering process. The *Signature File* [5, 10], the *Bit-Slice File* [9], the *S-Tree* [4] and the *Signature Tree* [3] are introduced.

In information filtering systems, a user profile may contain several keywords to filter incoming documents. Thus, a profile signature is formed by superimposing the signatures for all its keywords. Figure 3 depicts the signature generation and compression process of

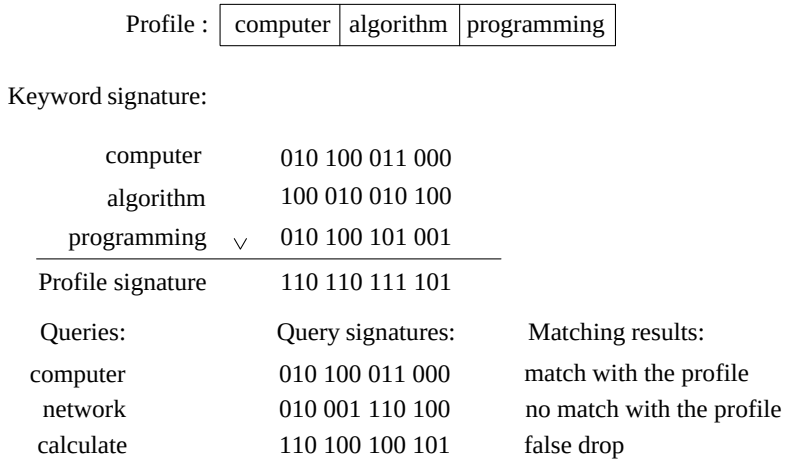


Figure 3: Signature generation and comparison

a profile having three keywords: "computer," "algorithm," and "programming." When a new document arrives, the profile signatures are scanned and many non-qualified profiles are discarded. Then, the rest profiles are checked in the refinement step to eliminate the *false drops*. After the refinement step, the matched document will finally return to the remaining users. A document containing certain keywords to be searched will be transformed into a document signature $Sig(D)$ in the same hash function. The document signature is then compared sequentially with every profile signature $Sig(P_i)$ in signature files. Three possible outcomes of the comparison are exemplified in Figure 3: (1) the profile matches the query (*i.e.*, $Sig(D) \wedge Sig(P_i) = Sig(P_i)$); (2) the profile does not match the query (*i.e.*, $Sig(D) \wedge SigP_i \neq Sig(P_i)$); (3) the signature comparison indicates a match but the profile in fact does not match the search criteria; this case is also called a *false drop* [5].

In a signature file, a set of profile signatures is sequentially stored, which is easy to implement and requires low storage space and low update cost. However, when a new document arrives, a full scan of the signature file is required. Therefore, it is generally slow in retrieval. Thus, each profile has to be checked when a new document arrives. In order to improve the matching process, the profile can be stored in a column-wise manner. That is, the signatures in the file are vertically stored in a set of files [9]. Concretely, if the length of the signatures is m, then all the signatures will be stored in m files. Thus, the profiles

in the signature files are separated by each bit. Therefore, the search cost of a bit-slice file is lower than that of a sequential one. However, the update cost in this data structure becomes larger than that of a sequential one.

The S-tree is a kind of the multi-level signature-based data structure, and is wildly applied on IR and IF systems. S-trees are first purposed by Deppisch [4]. Similar to B^+-trees [1] and R-trees [7], they are height-balanced trees having all leaves at the same level [4]. The advantage of this index strategy is that the scanning of a whole signature file is replaced by searching several paths in an S-tree. However, the storage overhead is almost doubled because of the space used by internal nodes. Furthermore, due to the OR operation among child nodes, the node near the root tends to have many "1" bits, resulting in low selectivity. The disadvantage was improved by Tousidou *et al* [16, 17]. They propose several strategies for node splitting and partial-tree restructuring to improve query-response time. However, this kind of improvement is achieved at cost of time to update data, because it is very time consuming to determine seeds in splitting nodes.

A signature tree that works for a signature file is just like a *trie* [13] for a text. But in a signature tree, each path is a *signature identifier* which is not a continuous piece of bits. A signature tree is quite different from a trie in which the bits labeling a path are consecutive. The signature identifiers can be considered as a generalization of the concept of position identifiers [3] extended to handle inexact matching.

The signature tree replaces the slice checking in the bit-slice strategy with a single bit checking. Thus, it takes less time to perform the insertion and deletion operations on signatures as compared with the bit-slice strategy. Moreover, because of the binary tree structure, the storage requirement of signature trees is much less than that of S-trees. However, because the signature tree extracts only one different item among each group at a time, the filtering effect may become poor when the query contains all of these different items.

3 The ID Tree Index Strategy

In this section, we present a signature-based tree structure to index database transactions, which can efficiently support the inexact filtering. The structure of the ID-tree is similar

Table 1: The example user profiles

Profile	Items	Signature
P1	1 2 3 4	1111000000
P2	1 3 5 6	1010110000
P3	2 3 4 5 7	0111101000
P4	2 4 6 8 9	0101010110
P5	2 4 6 7 8	0101011100
P6	1 2 3 9 10	1110000011
P7	1 7 8 9	1000001110
P8	1 2 6 7 8	1100011100
P9	1 2 3	1110000000

to a *binary trie* [13] for a text. But unlike the consecutive bits of each path in a trie, the paths in an ID-tree are discontinuous. The concept of the ID-tree is similar to the decision tree, which can partition data into subgroups according to the characteristic of each profile. However, the results of the ID-tree are profiles which have some relationship to the queries according to the query types; on the other hand, the results of the decision tree are some classifications of profiles which turn a complex data representation into a much easier structure. However, the ID-tree partitions data by finding items which can evenly divide user profiles into subgroups, but the decision tree partitions data by calculating the entropy value of each attribute in the profiles.

3.1 Construction of the ID Tree

The construction process of the ID-tree is composed of two steps: (1) preprocessing, and (2) extension. In the preprocessing step of constructing an ID-tree from database profiles, a count-based method is used to preprocess the database profiles. Figure 4 illustrates the ID-tree generated from the profiles in Table 1 after the preprocessing step. The preprocess steps of the profiles by using counting tables are described in procedure *Construct_IDtree* shown in Figure 5.

Initially, all profiles in the database are recorded in $PSet$ which will be added to the *root.prof* (line 3). Next, if the size of a group is greater than 1, we will partition the profiles into subgroups (line 4). Function *Generate_Key* is used to find the key for partitioning. This

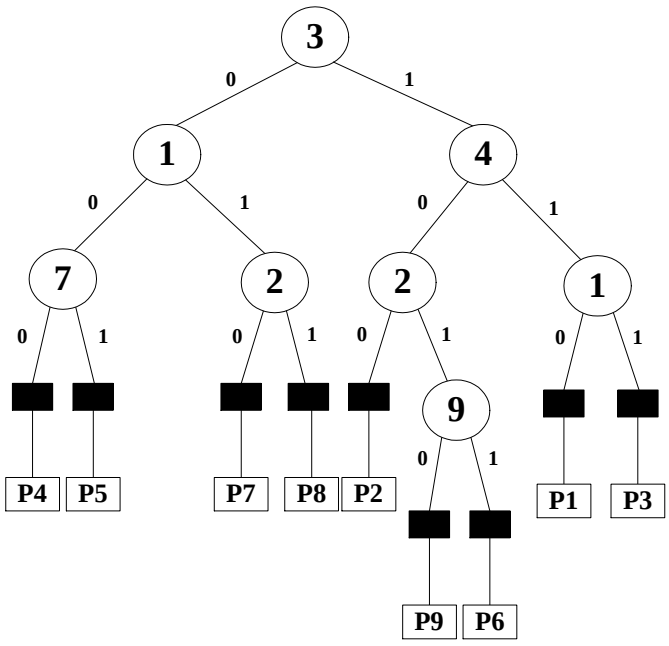


Figure 4: An example of the ID-tree after the preprocessing step

key will be set as an ID-key of identifier *root* (line 6). After finding the key for partitioning, this key will be removed from domain itemset D (line 7). Next, we will allocate all profiles into two subgroups according to whether they contain the key or not (lines 8-12). If the key is contained in a profile, this profile will be partitioned into the right subgroup, which is recorded in $root.rc.prof$ (lines 9-10). Otherwise, the profile will be partitioned into the left subgroup, which is recorded in $root.lc.prof$ (lines 11-12). Therefore, the first ID-key will exist only in one subgroup of profiles partitioned by an identifier. The partition of profiles will be continued recursively until the size of each subgroup is equal to 1 (line 13). When both the left and right subgroups partitioned by an identifier contains only 1 profile, we will set the profile as I_key of the left and right children of this identifier (lines 14-17). The value of I_key will be used in the extension step. Finally, we will check the size of the left and right subgroups partitioned by the identifier. If the size of a subgroup is greater than 1, this subgroup will be partitioned recursively (lines 19-28).

Function $Generate\_Key$ shown in Figure 6 is to generate the ID-keys of each identifier by its counting table. To find the key for partitioning, function $ItemCount$ is used to count

```
1: procedure Construct_IDtree(D, PSet, root)
2: begin
3:     root.prof := PSet;
4:     if (|root.prof| > 1) then    /* Create a branch. */
5:     begin
6:         root.key := Generate_Key(D, PSet);    /* Find the key for partition. */
7:         D := (D − root.key_set);    /* Remove the used key from domain D. */
8:         for each (profile t ∈ root.prof) do
9:             if (root.key ∈ t) then
10:                root.rc.prof := root.rc.prof ∪ t
11:            else
12:                root.lc.prof := root.lc.prof ∪ t;
13:        if (|root.lc.prof| = 1 and |root.rc.prof| = 1) then
14:        begin
15:            root.lc.I_key := root.lc.prof;
16:            root.rc.I_key := root.rc.prof;
17:        end;
18:    end;
19:    if (|root.lc.prof| > 1) then    /* Partition root.lc.prof recursively. */
20:    begin
21:        root := root.lc;
22:        Construct_IDtree(D, root.prof, root);
23:    end;
24:    if (|root.rc.prof| > 1) then    /* Partition root.rc.prof recursively. */
25:    begin
26:        root := root.rc;
27:        Construct_IDtree(D, root.prof, root);
28:    end;
29: end;
```

Figure 5: Procedure $Construct\_IDtree$

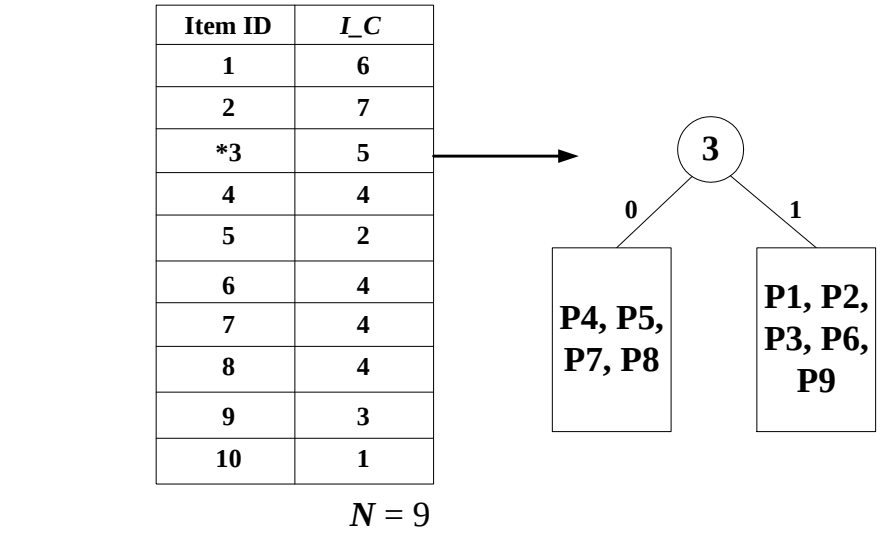
```
1: function Generate_Key(D, PSet): Integer
   /* Return the first ID-key of each identifier by its counting table. */
2: begin
3:     first_key := 0;
4:     I_C := ItemCount(PSet);
5:     mid := |PSet|/2;
6:     first_key = minI_C(I_C, mid);
7: end;
```

Figure 6: Function $Generate\_Key$

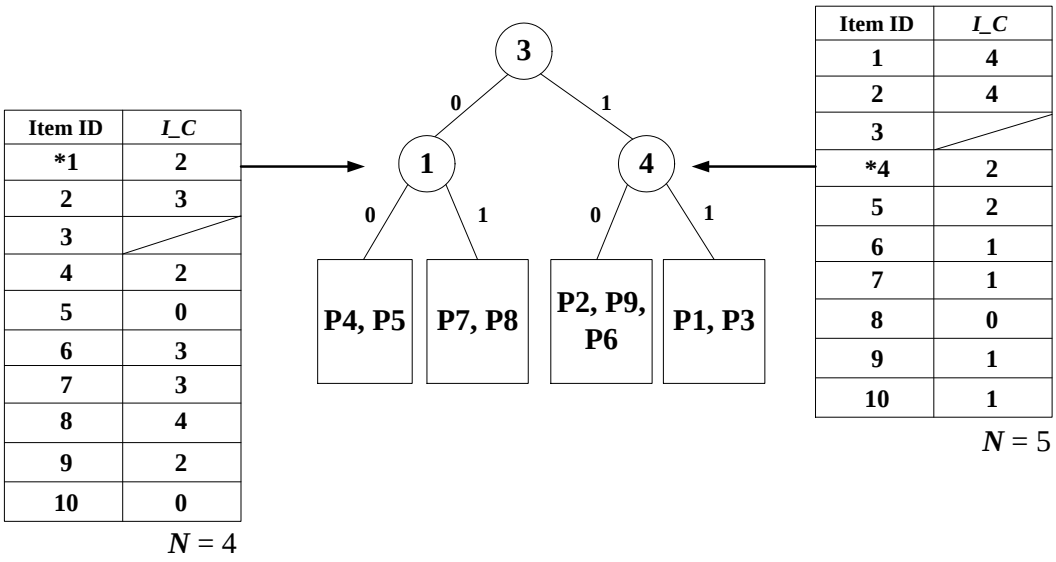
the number of each item in domain itemset D according to $PSet$. This function will return a counting table of $PSet$ and recorded as I_C (line 4). Next, the value of $\frac{|PSet|}{2}$ is calculated as mid to find the half number of profiles in the group (line 5). The value of $PSet$ and mid will be used in function $minI\_C$ to find the item which has the minimum value of $|I\_C[i] - mid|$, where $1 \leq i \leq |I\_C|$ (line 6). After the preprocessing step, the database profiles will be partitioned into subgroups with no collision in all leaves of the ID-tree.

Take the profiles shown in Table 1 as an example. Figure 7 illustrates the generation process of the first ID-key in each identifier using the counting tables. At the beginning, a counting table with all domain items is built in Function $Generate\_Key$ shown in Figure 6 to record the number of items in the database profiles. Next, the item j is chosen from the counting table such that $|I\_C[j] - \frac{1}{2}N|$ is minimal, $1 \leq j \leq N$, where $I\_C[j]$ is the number of item j appearing in all database profiles and N is the number of profiles in this group (or subgroup). This item will then be used as the first ID-key of the identifier. In Figure 7-(a), because there are 9 profiles in the database before the first partition, the first item in the counting table with the minimum value of $|I\_C[j] - \frac{1}{2} \times 9|$, $1 \leq j \leq 9$, *i.e.,* item 3, will be used as the first ID-key of the root identifier (*i.e.,* level 1). After the first ID-key is generated, the corresponding item will be removed from the domain, because this item will no longer be used as the ID-key of its descendant identifiers. Then, the database profiles can be partitioned into two subgroups according to whether they contain item 3 or not. In this case, item 3 is not contained in each profile of the left subgroup, *i.e.,* P_4, P_5, P_7 and P_8, and is contained in each profile of the right subgroup, *i.e.,* P_1, P_2, P_3, P_6 and P_9. After the profiles are partitioned, a path bit "0" will be assigned to the left link of the identifier and a bit "1" will be assigned to the right one.

Next, to generate the identifiers of the subgroups, the counting tables of the profiles in each subgroup are recursively built to find the ID-keys. Figure 7-(b) shows the process of the identifier generation in the second level of the ID-tree. Because item 3 has been removed from the domain at the first partition, this item will not be recorded in both counting tables used in level 2. In the left subgroup partitioned by the root identifier, there are 4 profiles in the subgroup. Thus, the counting table of this subgroup will be rebuilt by counting the number of items in these 4 profiles. In the counting table of the left subgroup,



(a)



(b)

Figure 7: The generation process of the first ID-key in each identifier by the counting tables: (a) level 1; (b) level 2.

item 1 can be chosen as the ID-key because that $|I\_C[1] - \frac{1}{2} \times 4|$ is minimum as compared with other items, where $N/2 = 2$. After ID-key 1 is determined in the left subgroup (*i.e.,* level 2), profiles P_4 and P_5 are partitioned into the left subgroup because that item 1 is not contained in them. Moreover, P_7 and P_8 are partitioned into the right subgroup because that item 1 is contained in them.

Similar to the processing of the left subgroup, the first ID-key of the right subgroup can be determined by the counting table of the profiles in it. Thus, item 4 with $I\_C = 2$ is set as the first ID-key of the right child identifier since $N/2 = 2.5$. Profiles P_2, P_6 and P_9 are partitioned into the left subgroup for excluding item 4, and P_1 and P_3 are partitioned into the right subgroup for including item 4. The partition of the profiles will be continued recursively until the number of profiles in each subgroup is decreased to 1.

Because there is only one ID-key in each identifier after the preprocessing step, the filtering effect of subgroups using ID-keys may be inefficient. To improve the filtering efficiency of each subgroup, a method for extending an identifier with more than one ID-key is used in our strategy. Because there may be more than one different item between two subgroups partitioned by an identifier, the basic idea of extending the ID-keys of each identifier is to find all the different items between two subgroups to partition them at one time. In other words, we increase the number of ID-keys in each identifier to partition each group more precisely. Figures 8-(a) and 8-(b) show the tracing paths of the ID-tree with inexact filtering query $q = (1, 2, 3, 5, 8)$ before and after the extension steps, respectively. We can observe that the number of ID-keys in each identifier increases after extension. For example, in Figure 8-(a), the root identifier of the ID-tree only contains 1 ID-key (*i.e.,* 3). In contrast, it contains 2 ID-keys (*i.e.,* 3, 8) after the extension step shown in Figure 8-(b). This is because these database profiles can be partitioned up by the absence or occurrence of 2 ID-keys at one time, which results in filtering more number of non-qualified profiles than the version of a single ID-key shown in Figure 8-(a). In the example, the tracing paths of query q are drawn in bold lines. If the path is connected to a profile, this profile will be checked with the query physically. That is, each bit of the profile will need to be checked with q. In the example, we can observe that 5 profiles are needed to be accessed physically before the extension (*i.e.,* P_2, P_4, P_7, P_8 and P_9) and only 1 profile is needed to be accessed

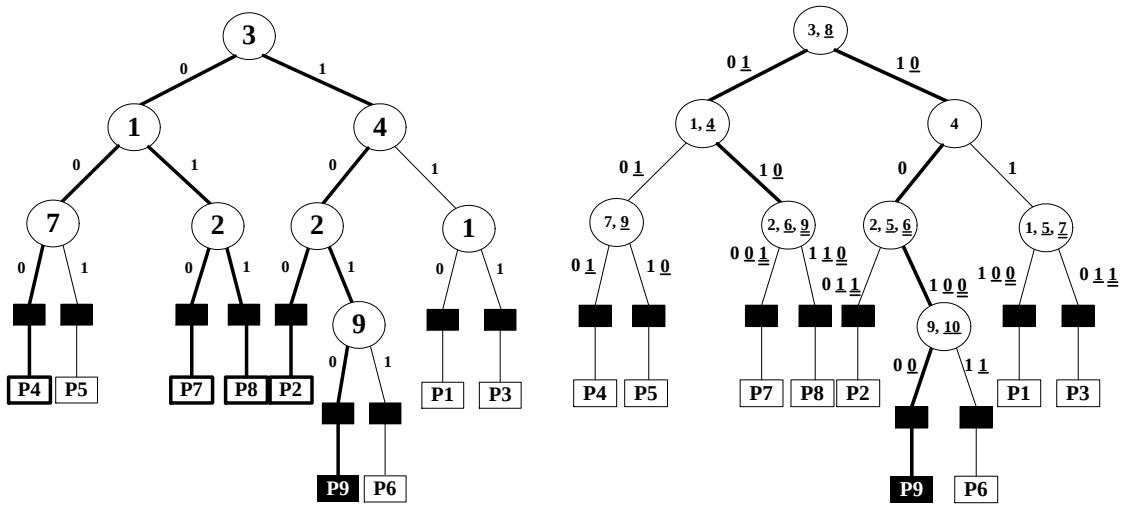


Figure 8: The tracing examples of ID-trees with the inexact filtering query $q = (1, 2, 3, 5, 8)$: (a) the ID-tree before the extension step; (b) the ID-tree after the extension step.

physically after the extension (*i.e.,* P_9). Therefore, it will filter out more irrelevant profiles to answer a query after the extension step.

The extension process of the identifiers in the ID-tree is described in procedure *Extend_IDkey* as shown in Figure 9. In this procedure, three arrays key_set, U_key and I_key are used in each identifier to record the information of subgroups partitioned by it. The information recorded in these arrays are described as follows:

1. key_set is used to record the ID-keys in an identifier.

2. U_key is used to record the ID-keys which have been used by the descendant of an identifier.

3. I_key is used to record the intersection of all profiles in each subgroup partitioned by an identifier.

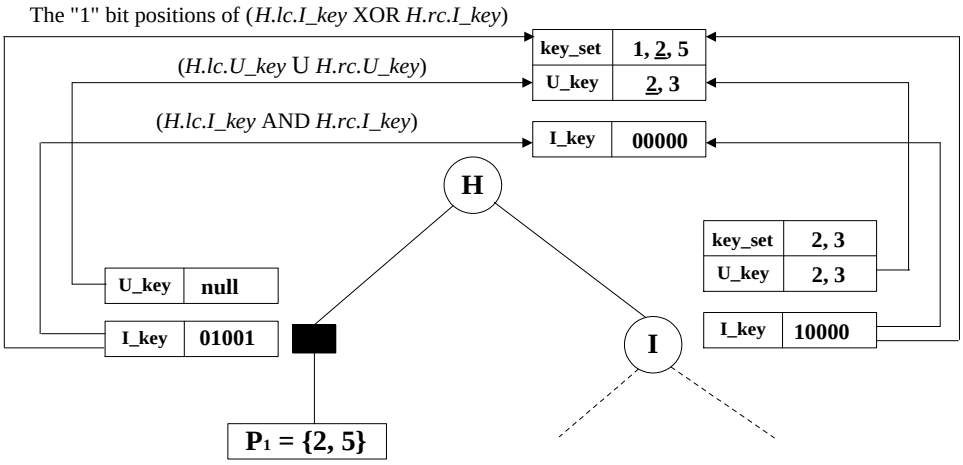
To extend the ID-keys of each identifier, the procedure first traces the ID-tree by its identifiers in the post-order manner (lines 3-6). After an identifier is searched (*i.e.,* a leaf node), the union of U_key in its left and right children are calculated to get these ID-keys which have been used (line 7). Then, in line 8, the exclusive OR (XOR) of I_key in these two

```
1: procedure Extend_IDkey(root)
     /* Extend ID-keys of each identifier in the ID-tree. */
2: begin
3:     if (root.lc is not a leaf) then
4:         Extend_IDkey(root.lc);
5:     if (root.rc is not a leaf) then
6:         Extend_IDkey(root.rc);
7:     root.U_key := (root.lc.U_key ∪ root.rc.U_key);
8:     root.key_set:=(the "1" bit positions of
           (root.lc.I_key XOR root.rc.I_key)−root.U_key);
9:     root.U_key := (root.U_key ∪ root.key_set);
10:    root.I_key := (root.lc.I_key AND root.rc.I_key);
11:    Assign_Path(root);
12: end;
```

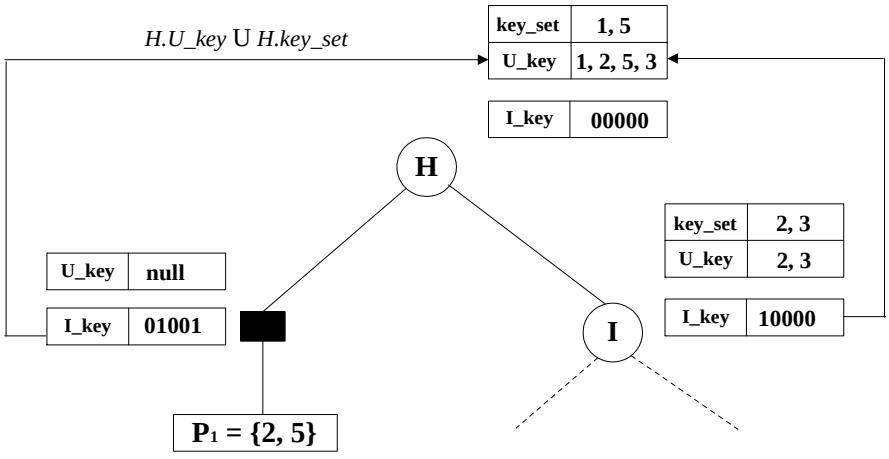
Figure 9: Procedure $Extend\_IDkey$

children ($root.lc.I\_key$ XOR $root.rc.I\_key$) will be calculated to get all of different items between two subgroups partitioned by the identifier. Therefore, each extended ID-key will also exist only in one subgroup of profiles partitioned by an identifier. Because some of these items may have existed in descendants of the identifier, these duplicated items have to be eliminated according to U_key to prevent false calculation in the query stage. This is the reason of the operation (($root.lc.I\_key$ XOR $root.rc.I\_key$)$-root.U\_key$)). After duplicated items are eliminated, the remaining items will be used as new ID-keys and recorded in key_set of the identifier (line 8). That is, the "1" bit positions of (($root.lc.I\_key$ XOR $root.rc.I\_key$)$-root.U\_key$)). Because new ID-keys are added to the identifier, the information of used ID-keys need to be updated to U_key (line 9). Then, the intersection of I_key in the left and right children of the identifier will be calculated and recorded in I_key of it (line 10). The information will be used in the extension of its father identifier. Finally, Function $Assign\_Path$ will be used to assign new bit paths of the left and right subgroups partitioned by the identifier (line 11).

Take the identifiers shown in Figure 10 as an example. In the example, identifier I is the right child of identifier H. To generate the information of H, *i.e.,* $H.key\_set$, $H.U\_key$ and $H.I\_key$, the information of left and right children of H (*i.e.,* $H.lc$ and $H.rc$) will be used. In Figure 10-(a), first, $H.I\_key$ is generated by ($H.lc.I\_key$ AND $H.rc.I\_key$), and $H.U\_key$ is generated by ($H.lc.U\_key \cup H.rc.U\_key$). Next, the ID-keys of H is generated



(a)



(b)

Figure 10: An example of the extension step: (a) before duplication being eliminated; (b) after duplication being eliminated.

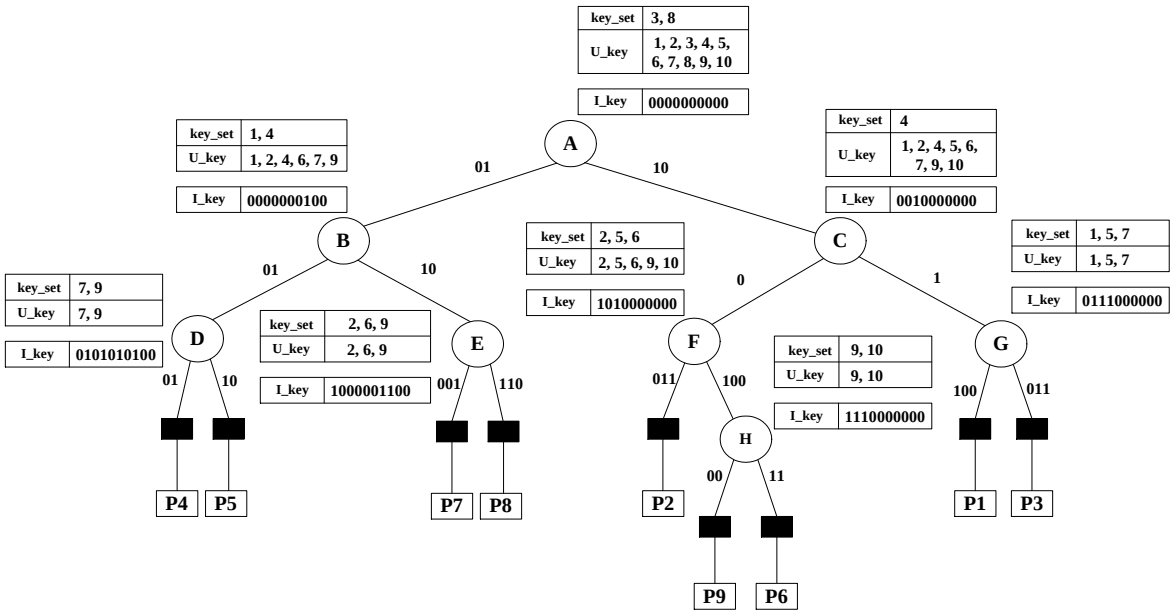


Figure 11: An example of ID-tree after the extension step

by the "1" bit positions of ($H.lc.I\_key$ XOR $H.rc.I\_key$) and recorded in $H.key\_set$. Then, $H.U\_key$ is used to check whether these ID-keys have already been used by the descendants of H or not. Item 2 of $H.key\_set$, which is in underline in Figure 10-(a), is contained in $H.U\_key$; therefore, this item needs to be removed from $H.key\_set$. It means that item 2 certainly has been used by the descendants of H. Thus, the ID-keys of H contain only 1 and 5. The result of $H.key\_set$ after the duplication is eliminated is shown in Figure 10-(b). After the duplication is eliminated, $H.U\_key$ is reassigned by ($H.U\_key \cup H.key\_set$) to guarantee that it contains all of the used ID-keys.

After the generation of new ID-keys, function $Assign\_Path$ is used to assign the bits to the left and right bit paths (*i.e.*, lp and rp) of the identifier according to whether the subgroups contain the ID-keys or not. The extension step will be continued recursively until the ID-keys of each identifier are extended. Figure 11 illustrates the ID-tree generated from the profiles in Table 1 after the extension step. The filtering efficiency of each identifier in the ID-tree will be better after the extension process. Thus, when queries are received, it needs shorter time to search their similar profiles by using the ID-tree.

```
1: procedure Inexact_Filter(q, root)
 /* Filter out the profiles in the signature tree which are not contained in the query q.
    */
2: begin
3:     if (root is not a leaf node) then
4:         if (root.key ∈ q) then
           /* q contains the key item. */
5:         begin
6:             Inexact_Filter(q, root.lc);
7:             Inexact_Filter(q, root.rc);
8:         end
9:         else
10:            Inexact_Filter(q, root.lc)
   /* Trace the left child of root if q does not contain the key item. */
11:    else
12:        if (root.pro ⊆ q) then
13:            Add root.prof to the answer set;
14: end;
```

Figure 12: Procedure *Inexact_Filter* (based on the signature tree)

3.2 Inexact Filtering in the ID Tree

Basically, the query process for the inexact filtering in our ID-tree strategy is similar to the *signature tree* strategy which retrieves the inexact match information [3]. However, for inexact filtering, we need to revise Chen's signature tree strategy [3] by a simple modification since we want to filter out irrelevant information, instead of retrieving it. The modified strategy based on the signature tree is described in the procedure *Inexact_Filter* shown in Figure 12. In procedure *Inexact_Filter*, when a query is received, it will trace the signature tree in the pre-order manner. In the inexact filtering, we will trace both the left and right children of a node if query q contains the key item recorded in this node (line 4). (However, in the inexact match, it will traverse both children of a node if query q does not contain the key item.) Otherwise, we will only trace the left child if query q does not contain the key item. This is because all the database profiles in the right subtree are not contained in the query (lines 9-10). (However, in the inexact match, it will only trace the right child if query q contains the key item.)

Take the profiles in Table 1 as the input data. Figure 13-(a) illustrates the tracing

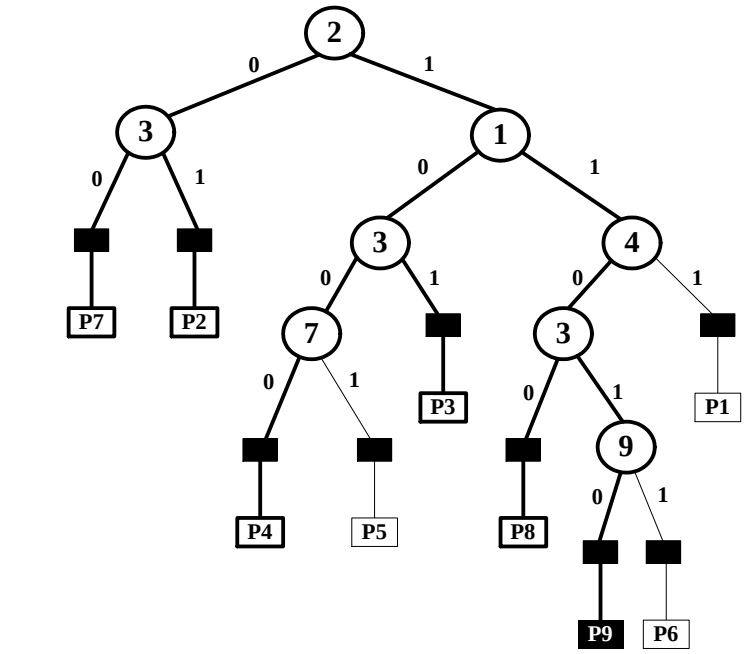
Table 2: A comparison of the number of accessed data with query $q = (1, 2, 3, 5, 8)$ in inexact filtering

Strategies	**The number of accessed profiles**	**The number of traversed bits in the index**
Signature tree	6 (60 bits)	8 bits
ID-tree	1 (10 bits)	13 bits

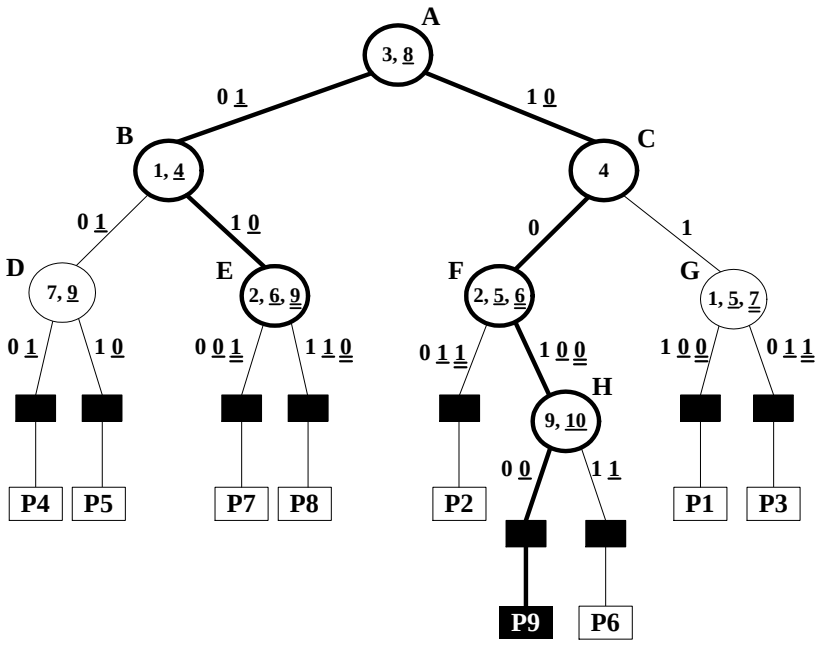
example by using Chen's signature tree [3] with the query $q = (1, 2, 3, 5, 8)$. The tracing paths of the query are drawn in bold lines. Therefore, to answer the inexact filtering result with query q, 8 internal nodes and 6 profiles with query q are checked in the example.

Instead of using Chen's signature tree [3], we use the ID-tree structure to index the database profiles. The query steps using the ID-tree are described in procedure $Mod\_Inexact\_Filter$ shown in Figure 14. In the procedure, it traverses the subtrees according to the bit paths of each identifier. That is, it traces the subtree of the identifier only if the key items which correspond to the bit paths are contained in the query (lines 4-9). For example, there are three ID-keys 2, 5 and 6 in the identifier F of the tree in Figure 13-(b). Because the left bit path of F is "011," the profiles of the left subgroup partitioned by F certainly contain items 5 and 6, and item 2 must be excluded from them. Thus, the left subtree of F will not be traced unless the query contains both items 5 and 6. Figure 13-(b) illustrates the tracing example by using the ID-tree. Therefore, to search the inexact filtering result of the query, we only have to check 13 ID-keys and 1 profile in the ID-tree.

A comparison of the number of accessed data between the signature tree and the ID-tree with query $q = (1, 2, 3, 5, 8)$ in inexact filtering is shown in Table 2. Although we need to traverse more bits in the index by using the ID-tree strategy, the accessed profiles can be reduced a lot by using it. Moreover, the indexes of both strategies will be loaded into memory, and the physical profiles will be stored in the disk. Therefore, it is obvious that our ID-tree strategy can improve the query performance in the inexact filtering database systems as compared with the signature tree strategy.



(a)



(b)

Figure 13: A tracing example with the inexact filtering query $q = (1, 2, 3, 5, 8)$ by using two kinds of signature-based binary trees: (a) the signature tree; (b) the ID-tree.

```
1: procedure Mod_Inexact_Filter(q, root)
  /* Filter out the profiles in the ID-tree which are not contained in the query q. */
2: begin
3:     if (root is not a leaf node) then
4:     begin
5:         if (the left bit path of root is contained in q) then
6:             Mod_Inexact_Filter(q, root.lc);
7:         if (the right bit path of root is contained in q) then
8:             Mod_Inexact_Filter(q, root.rc);
9:     end
10:    else
11:        if (root.prof ⊆ q) then
12:            Add root.prof to the answer set;
13: end;
```

Figure 14: Procedure $Mod\_Inexact\_Filter$ (based on the ID-tree)

Table 3: Parameters used in the generation of synthetic profiles of the simulation

Parameter	**Description**
N	The number of profile signatures
D	The number of items in the domain itemset
W	The number of items in each profile signature
Q	The parameter that controls the similarity among profiles
$query\_w$	The percentage of domain items

4 Performance

In this section, we study the performance of our strategy for inexact filtering query based on the ID-tree. First, the simulation model is presented briefly. Second, we compare the performance of our strategy with that of Chen's signature tree strategy [3].

4.1 The Simulation Model

In this simulation, we use the profile generator which has been applied in [3] to generate synthetic profiles and to evaluate the performance. Each synthetic profile will be transformed into the signature defined in [12] after the generation. The parameters used to generate the synthetic profiles are shown in Table 3. Parameter N is the number of database profiles. Each profile is composed of keywords chosen from the domain item set. Therefore, the

length of each profile signature is set to a fixed number D, which is the number of items in the domain item set. To simplify the study of the effect of the profile size on performance, all synthetic profiles have the same number of items. Parameter W is used to control the number of items in each profile signature, *i.e.,* the number of "1" bits in a signature. Thus, we can generate a *sparse profile*, *i.e.,* a profile which contains few items, by tuning parameter W. The first profile generated from the domain items is called the *base profile*. The base profile is like a seed of all the other synthetic profiles. After the base profile has been generated, other profiles will be generated according to it.

The similarity parameter, Q, controls how similar the new profiles and the base profile are, where $0 \leq Q \leq 1$. That is, for each bit which is set to "1" in the signature of the base profile, there is a probability Q that the corresponding bit of the new profile signature is also set to "1." A random variable r will be used to determine whether the probability is hold or not, where $0 \leq r \leq 1$. If $r \leq Q$, then the corresponding bit of the new profile signature is set to "1." Otherwise, this bit is set to "0." After each "1" bit of the base profile has been scanned, if the number of items in the new profile is less than W, we will turn on the "0" bits of this profile to "1" at random until the number of "1" bits in it is equal to W. Because there is no duplicated profile in the database, we can control the distribution of the cluster by tuning the similarity parameter Q from 0 to 1. In other words, all the synthetic profile signatures are created randomly using a uniform distribution for the positions that will be set to "1." After the signatures of synthetic profiles are generated, they will be used as the input data of the signature tree [3] and our proposed ID-tree.

For the inexact filtering, parameter $query\_w$ is used to control the number of items in the synthetic query. This parameter is chosen from 0% to 100%. It indicates that what percentage of the domain items will be chosen in the query. If a domain item is chosen by the query, the corresponding bit position in the signature of the query will be set to "1." For example, when $query\_w$ is set to 50%, and the domain contains 100 items ($D = 100$), there will be 50 bits set to "1" in the query signature.

Table 4: Base values for five parameters used in the simulation

Parameter	**Default value**
N	1000
D	110
W	35
Q	0.5
$query\_w$	80%

Table 5: Comparisons between the signature tree and the ID-tree (under the base case)

Strategies	**The number of accessed profiles**	**The number of traversed bits in the index**
Signature tree	334	854
ID-tree	7 (reduced 98%)	2996

4.2 Simulation Results of Inexact Filtering Strategies

In this section, we make a comparison of our ID-tree strategy and the signature tree strategy [3] in inexact filtering. In our simulation, we define a base case as shown in Table 4. That is, we cluster 1000 user profiles into the same group. All the profile signatures are created randomly by using a uniform distribution with similarity parameter 0.5 (*i.e.,* $Q = 0.5$) for the positions that will be set to "1." The set of domain items is composed of 110 items, *i.e.,* $D = 110$. The length of each profile is fixed, *i.e.,* $W = 35$. In the following, our simulation results are the average of 1000 queries.

According to those parameters in the base case, a comparison of the number of accessed profiles and the number of traversed bits in the index by using our ID-tree strategy and the signature tree strategy for $query\_w = 80\%$ is shown in Table 5. In Table 5, we show that the signature tree strategy [3] needs to access more physical profiles to filter the false answers as compared with the ID-tree strategy. This is because that the signature tree traverses fewer bits in the index stored in the memory. In Table 5, although the ID-tree strategy traverses more bits in index, the traversal needs less time as compared with the access time in physical profiles. For each physical profile, it needs to check D bits (*i.e.,* 110 bits) to filter the false results. Moreover, all physical profiles are stored in the disk.

Therefore, to access the profiles, it needs extra time to load them into the memory. Because the cost of the profile access time is over 200 times more than the index access time, we can ignore the number of traversed bits in the index unless the number of accessed profiles between two strategies is almost the same in the following. In Table 5, on the average, our strategy can reduce about 98% number of accessed profiles as compared to Chen's strategy [3]. The value of the reduced percentage can be calculated by using the formula described as follow:

$$reduced\ percentage = (1 - \frac{the\ number\ of\ accessed\ profiles\ by\ using\ the\ ID{-}tree\ strategy}{the\ number\ of\ accessed\ profiles\ by\ using\ the\ signature\ tree\ strategy}) \times 100\%$$

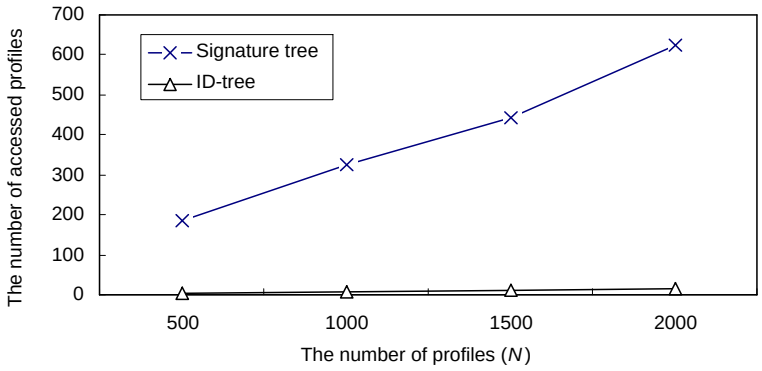


Figure 15: A comparison of the number of accessed profiles (under changing the value of N)

Next, we study the impact of five parameters on the performance. In the first case, we vary the value of N, the number of profiles. The range of N is set to 500, 1000, 1500 and 2000, while the other parameters are kept as their base values. Under changing the value of N, a comparison of the number of accessed profiles by using the ID-tree strategy and the signature tree strategy for $query\_w = 80\%$ is shown in Figure 15. In Figure 15, when the value of N increases, the number of accessed profiles by using the ID-tree and the signature tree also increases. However, our strategy needs to access less profiles to refine the answers as compared to Chen's strategy. This is because that the ID-tree can filter more false profiles in the index as compared to the signature tree.

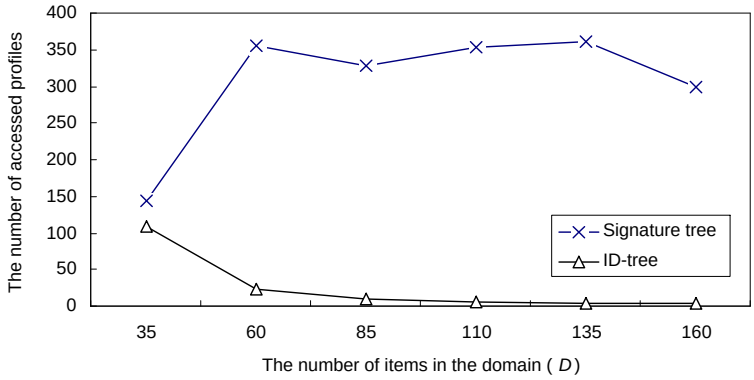


Figure 16: A comparison of the number of accessed profiles (under changing the value of D)

In the second case, we vary the value of D, the number of items in the domain. The range of D is set to 35, 60, 85, 110, 135 and 160, and the base values are used for the other parameters. Under changing the value of D, a comparison of the number of accessed profiles by using the ID-tree strategy and the signature tree strategy for $query\_w = 80\%$ is shown in Figure 16. In Figure 16, when the value of D increases, the number of accessed profiles by using our strategy decreases. However, the number of accessed profiles by using Chen's strategy is unstable. This is because that it only considers one of different items among database profiles in the signature tree. Under the fixed value of W (*i.e.,* the weight of each database profile), the sparsity of each profile will increases as the value of D (*i.e.,* the number of items in the domain) increases. A user profile will become *sparse* if the number of items chosen from his or her domain is very small. Therefore, from this result, we show that our strategy can reduce more accessed profiles as compared to Chen's strategy when database profiles become *sparse*. This is because when the database profiles are sparse, there may be more different bit positions among them. Thus, the probability to generate more ID-keys by using the ID-tree strategy will become greater.

In the third case, we vary the value of W, the number of items in each profile signature. The range of W is set to 15, 35, 55, 75, and 95, and the base values are used for the other parameters. Under changing the value of W, a comparison of the number of accessed

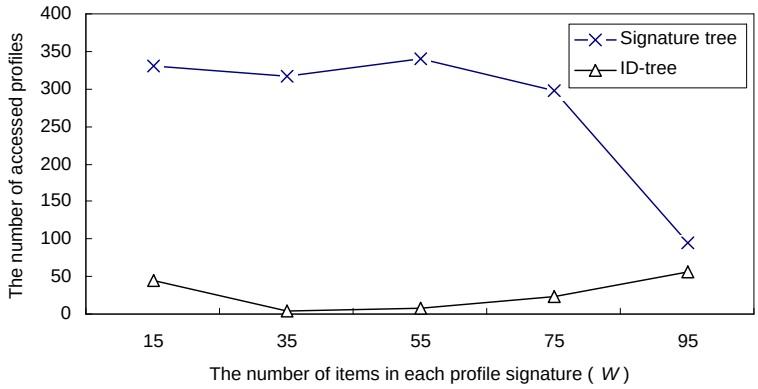


Figure 17: A comparison of the number of accessed profiles (under changing the value of W)

profiles by using the ID-tree strategy and the signature tree strategy for $query\_w = 80\%$ is shown in Figure 17. In Figure 17, when the value of W increases, the number of accessed profiles by using our strategy increases from $W = 35$. Moreover, the number of accessed profiles by using Chen's signature tree strategy is still unstable. This is because that it only considers one of different items among database profiles in the signature tree. Therefore, from this result, we show that our strategy can reduce more the number of accessed profiles as compared to Chen's signature tree strategy, when database profiles become sparse. However, when the value of W is smaller than 35, the number of accessed profiles by using the ID-tree strategy decreases when the value of W increases. This is because the number of different bit positions among all profiles is too less. Therefore, the number of ID-keys in the ID-tree may become less in this condition. If there is only one item in each database profile (*i.e.*, $W = 1$), the index of the ID-tree will become the same as the signature tree.

In the fourth case, we vary the value of Q, the parameter that controls the similarity among profiles. The range of Q is set to 0.1, 0.3, 0.5, 0.7, and 0.9, and the base values are used for the other parameters. Under changing the value of Q, a comparison of the number of accessed profiles by using the ID-tree strategy and the signature tree strategy for $query\_w = 80\%$ is shown in Figure 18. From this result, we show that when the value of

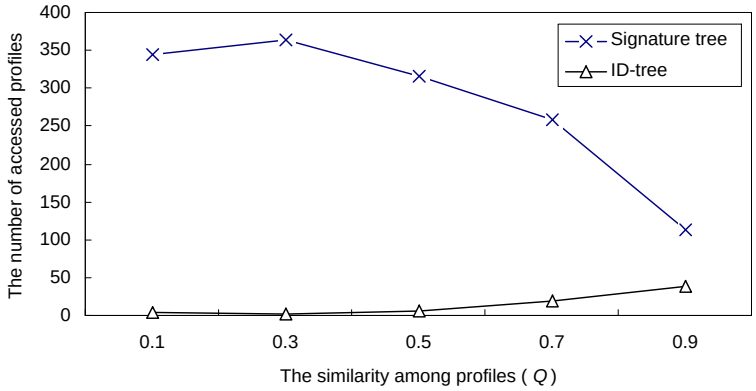


Figure 18: A comparison of the number of accessed profiles (under changing the value of Q)

Q increases, it needs to access more profiles as compared to the small value of Q by using our strategy. This is because when the value of Q increases, the number of identical items among database profiles will also increase. Therefore, the number of ID-keys in the ID-tree may relatively decreases. Moreover, we can observe that the number of accessed profiles by using the signature tree decreases as the value of Q increases. This is because when the database profiles are similar to each other, the number of internal nodes with identical values in the signature tree will increase. Therefore, it will filter a large number of profiles at once by the index nodes of the signature tree in this condition. However, our strategy still reduces over 65% the number of accessed profiles as compared to Chen's strategy even if $Q = 0.9$. When Q is set to nearly 1 and each database profile has only one item different from others, the ID-tree will become the same as the signature tree. Moreover, both of these trees will be very unbalanced in this condition.

In the fifth case, we vary the value of $query\_w$, the parameter that controls the weight of each query. We vary $query\_w$ from 50% to 90% and the base values are used for the other parameters. Under changing the value of $query\_w$, a comparison of the number of accessed profiles by using the ID-tree strategy and the signature tree strategy is shown in Figure 19. From this result, we show that when the value of $query\_w$ increases, both strategies

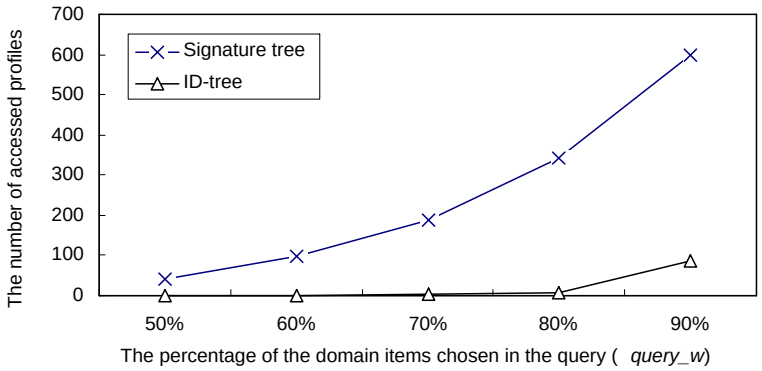


Figure 19: A comparison of the number of accessed profiles (under changing the value of $query\_w$)

need to access more physical database profiles to refine the answers. This is because when the weight of a query is very large, it may pass most of the index nodes in both strategies. However, passing the index nodes of the ID-tree, it needs to check all the different items among database profiles. Therefore, the probability to pass the ID-tree is much less than the signature tree. In other words, our strategy can filter out more false profiles as compared to Chen's strategy. Moreover, if $query\_w$ is set to 100%, both strategies need to traverse all of the index nodes and access all physical profiles. Because the index size of the ID-tree is larger than that of the signature tree, it needs longer access time to respond the query by using our strategy in this condition.

From these simulation results, we have observed that our strategy can really reduce the number of accessed profiles more than Chen's signature tree strategy. Unless the weight of the incoming query is very heavy, our strategy can filter most of false profiles in the index, which is stored in the memory. That is, the ID-tree strategy provides the best results with different kinds of input data sets.

5 Conclusion

As the booming development of web data, information filtering has become an important issue for sending appropriate information to appropriate users. In this paper, we have proposed an efficient signature-based index strategy, the ID-tree index, to support the inexact filtering in information filtering systems. Our proposed ID-tree strategy can reduce the number of accessed profiles as compared with Chen's signature tree strategy. Our ID-tree structure partitions profiles into subgroups globally by considering all of the different items among database profiles at one time. These items will help us filter most irrelevant profiles in the index and answer queries efficiently. Moreover, each profile will be assigned a unique path and can be recognized by the path. After filtering irrelevant profiles, information can be sent to a group of users by using our ID-tree strategy. From our simulation, we have shown that our strategy can reduce up to 98% the number of accessed profiles in the inexact filtering as compared with Chen's signature tree strategy.

6 Acknowledgement

This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-95-2221-E-110-101. The authors also like to thank "Aim for Top University Plan" project of NSYSU and Ministry of Education, Taiwan, for partially supporting the research.

References

[1] R. Bayer and K. Unterrauer, "Prefix B-Tree," *ACM Trans. on Database Systems*, Vol. 2, No. 1, pp. 11–26, March 1977.

[2] Y. I. Chang, J. H. Shen, and T. I. Chen, "A Data Mining-Based Method for the Incremental Update of Supporting Personalized Information Filtering," *Journal of Information Science and Engineering*, Vol. 24, No. 1, pp. 129–142, Jan. 2008.

[3] Y. Chen, "On the Signature Tree and Balanced Signature Trees," *Proc. of the 21st IEEE Int. Conf. on Data Engineering*, pp. 742–753, 2005.

[4] U. Deppisch, "S-Tree: A Dynamic Balanced Signature Index for Office Retrieval," *Proc. of ACM Conf. on Research and Development in Information Retrieval*, pp. 77–87, 1986.

[5] C. Faloutsos, "Access Methods for Text," *ACM Computing Surveys (CSUR)*, Vol. 17, No. 1, pp. 49–74, March 1985.

[6] C. Faloutsos and D. W. Oard, "A Survey of Information Retrieval and Filtering Methods," *Technical Report, University of Maryland*, Aug. 1995.

[7] A. Guttman, "R-Tree: A Dynamic Index Structure for Spatial Searching," *Proc. of 1984 ACM SIGMOD Int. Conf. on Management of Data*, pp. 47–54, 1984.

[8] M. Hammami, Y. Chahir, and L. Chen, "Webguard: A Web Filtering Engine Combining Textual, Structural, and Visual Content-Based Analysis," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 18, No. 2, pp. 272–284, Feb. 2006.

[9] Y. Ishikawa, H. Kitagawa, and N. Ohbo, "Evaluation of Signature Files as Set Access Ficilities in Oodbs," *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pp. 247–256, 1993.

[10] A. J. Kent, R. Sacks-Davis, and K. Ramamohanarao, "A Signature File Scheme Based on Multiple Organizations for Indexing Very Large Text Databases," *Journal of the American Society for Information Science*, Vol. 41, No. 7, pp. 508–534, Oct. 1990.

[11] T. Kuflik and P. Shoval, "User Profile Generation for Intelligent Information Agents-Research in Progress," *Proc. of the Second Int. Workshop Agent-Oriented Information System*, pp. 63–72, 2000.

[12] N. Mamoulis, D. W. Cheung, and L. Wang, "Similarity Search in Sets and Categorical Data Using the Signature Tree," *Proc. of the 19th IEEE Int. Conf. on Data Engineering*, pp. 75–86, 2003.

[13] D. R. Morrison, "Patricia — Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of ACM*, Vol. 15, No. 4, pp. 514–534, Oct. 1968.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd, "The Pagerank Citation Ranking: Bringing Order to the Web," *Technical Report, Stanford University, Stanford, CA*, Jan. 1998.

[15] J. Salter and N. Antonopoulos, "Cinemascreen Recommender Agent: Combining Collaborative and Content-Based Filtering," *IEEE Intelligent Systems*, Vol. 21, No. 1, pp. 35–41, Feb. 2006.

[16] E. Tousidou, P. Bozanis, and Y. Manolopoulos, "Signature-Based Structures for Objects with Set-Valued Attributes," *Information Systems*, Vol. 27, No. 2, pp. 93–121, April 2002.

[17] E. Tousidou, A. Nanopoulos, and Y. Manolopoulos, "Improved Methods for Signature-Tree Construction," *The Computer Journal*, Vol. 43, No. 4, pp. 301–314, June 2000.

[18] D. H. Widyantoro, T. R. Ioerger, and J. Yen, "An Adaptive Algorithm for Learning Changes in User Interests," *Proc. of the 8th Int. Conf. on Information and Knowledge Management*, pp. 405–412, 1999.