XML Filtering Using Dynamic Hierarchical Clustering of User Profiles

Panagiotis Antonellis and Christos Makris

Computer Engineering and Informatics Department, University of Patras, Rio 26500, Greece
adonel@ceid.upatras.gr makri@ceid.upatras.gr

Abstract. Information filtering systems constitute a critical component in modern information seeking applications. As the number of users grows and the information available becomes even bigger it is crucial to employ scalable and efficient representation and filtering techniques. In this paper we propose an innovative XML filtering system that utilizes clustering of user profiles in order to reduce the filtering space and achieves sub-linear filtering time. The proposed system employs a unique sequence representation for user profiles and XML documents based on the depth-first traversal of the XML tree and an appropriate distance metric in order to compare and cluster the user profiles and filter the incoming XML documents. Experimental results depict that the proposed system outperforms the previous approaches in XML filtering and achieves sub-linear filtering time.

1 Introduction

Information filtering systems [1] are systems that provide two main services: document selection (i.e., determining which documents match which users) and document delivery (i.e., routing matching documents from data sources to users). In order to implement efficiently these services, information filtering systems rely upon representations of user profiles, that are generated either explicitly by asking the users to state their interests, or implicitly by mechanisms that track the user behaviour and use it as a guide to construct his/her profile. Initial attempts to construct such profiles typically used "bag of words" representations and keyword similarity techniques to represent user profiles and match them against new data items. These techniques, however, often suffer from limited ability to express user interests, being unable to fully capture the semantics of the user behaviour and user interests. As an attempt to face this lack of expressibility, there have appeared lately a number of systems that use XML representations for both documents and user profiles and that employ various filtering techniques to match the XML representations of user documents with the provided profiles. The process of filtering XML documents is the reverse of searching XML documents for specific structural and value information. An XML document filtering system stores user profiles along with additional information (e.g. personal information of the user, email address). When an XML document arrives, the system filters it through the stored profiles to identify with which of them the document fits. After the filtering process has finished, the document can be sent to the corresponding users with matching profiles.

1.1 Existing approaches

The existing XML filtering systems can be categorized as follows:

Automata-based Systems. The prominent examples of automata-based systems are XFilter [2] and YFilter[7]. Systems in this category incorporate Finite State Automata (FSA) to quickly match the document with the user profiles. In these systems, each data node causes a state transition in the underlying finite state automata representation of the filters.

Sequence-based Systems. Systems in this category represent both the user profiles and the XML documents as string sequences and then perform subsequence matching between the document and profile sequences. FiST [12] employs a novel holistic matching approach, that instead of breaking the twig pattern into separate root to leaf paths, transforms (through the use of the Prüfer sequence representation) the matching problem into a subsequence matching problem. In order to provide more efficient filtering, user profiles sequences are indexed using hash structures. XFIS [4] represents XML documents and user profiles using a novel sequence representation based on post order traversal and Prüfer sequences. XFIS supports on-line filtering of XML documents in only one pass, thus it is ideal for on-line applications and filtering systems.

Stack-based Systems. The representative system of this category is AFilter [5]. AFilter utilizes a stack structure while filtering the XML document against user profiles. Its novel filtering mechanism leverages both prefix and suffix commonalities across filter statements, avoids unnecessarily eager result/state enumerations (such as NFA enumerations of active states) and decouples the memory management task from result enumeration to ensure correct results even when the memory is tight.

Push Down Approaches. XPush [9] translates the collection of filter statements into a single deterministic pushdown automaton. The XPush machine uses a SAX parser that simulates a bottom up computation and hence doesn't require the main memory representation of the document. XSQ [13] utilizes a hierarchical arrangement of pushdown transducers augmented with buffers.

Suitable clustering algorithms for semistructured documents were extensively studied in [11]. XML document clustering was based in modeling the XML documents as trees, calculating the tree edit distance between them and applying a modified hierarchical clustering algorithm [8]. The tree edit distance is computed as the minimum-cost sequence of operations required to convert one given tree to another [10] [14]. In [6] the authors suggest the usage of tree structural summaries to improve the performance of the distance calculation and at the same time to maintain or even improve its quality. In [3] the authors introduce a novel compact representation of XML documents based on edge summaries. The proposed structure is utilized along with a suitable distance metric to efficiently cluster homogeneous and heterogeneous XML documents.

1.2 Motivation and contribution

Existing XML filtering approaches are not always effective in filtering XML documents against a rapidly growing number of stored user profiles for the following reasons:

- Usually, filtering systems cover a wide range of user interests and topics, thus each incoming XML document is relevant to a small portion of stored user profiles. However this fact is ignored by most filtering systems and the XML documents are checked against all user profiles.
- Systems considering similarities between stored user profiles, e.g. AFilter, utilize those similarities only to reduce extra checks, thus they keep checking every XML document against all user profiles.

In this research work we propose a filtering system that:

- Utilizes a unique sequence representation for both user profiles and XML documents based on the preorder traversal of XML tree.
- Measures similarity between two given user profiles or between an XML document and a user profile based on an innovative metric that utilizes a modification of the Levenshtein distance between the corresponding string representations.
- Creates an hierarchical structure of clusters using a hybrid hierarchical clustering algorithm based on the above mentioned metric.
- Applies a dynamic hierarchical filtering approach for each incoming XML document, based on the formed structure of clusters. The number of different levels of filtering depends on the number of previous matches in each level of clusters

The rest of the paper is structured as follows. Section 2 introduces the utilized sequence representation and describes the distance metrics adopted; section 3 discusses analytically the clustering and filtering processes; section 4 discusses the experimental results and section 5 presents our conclusions.

2 Sequence Representation and Distance Metrics

2.1 Sequence Representation of XML trees

In this work, we use a unique sequence representation of XML documents and user profiles, based on the preorder traversal of XML trees.

Every XML document can be easily modeled as an XML tree, where every enclosed element or attribute is modeled as a child in the XML tree. On the other hand, tree modeling of user profiles (expressed in XPath[19]) is not straightforward, as they may contain special relations (such as `//`, etc). In this paper we consider only parent/child (`/`) and ancestor/ descendant relations (`//`), which is the most used relation in user profiles. In order to model such a relation, we add an extra node in the XML tree, labeled with `*`. Figure 1 depicts an example of modeling a user profile (expressed in XPath) as an XML tree.

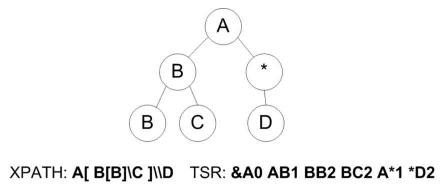

Fig. 1. Modeling a user profile as XML tree

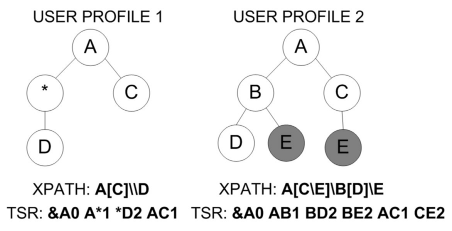

Fig. 2. Modified edit distance between two user profiles

In order to construct the sequence representation of an XML tree, we need to replace each distinct tag label with a single unique char. For this reason, we utilize a dictionary structure that assigns each distinct tag with a single char label and keeps track of the correspondence between tag labels and char labels. In the rest of this paper, for simplicity reasons, we will refer to the XML nodes directly with their char labels.

Based on the above observations, we introduce a novel tree sequence representation (TSR) of XML trees, based on preorder traversal, with the property that every XML tree is represented by a unique sequence representation. Each node of the XML tree is encoded by the pair `<Parent><Node><Depth>`, where `<Parent>` represents the parent's char label, `<Node>` represents the current node's char label and `<Depth>` represents the current's node depth. If the current node is the root node, we replace `<Parent>` with `&`. For example, let us consider the XML tree in Figure 1. The encoding of the `C` node is `BC2`, where `B` is the char label of node's parent, `C` is the node's char label and `2` is its depth in the XML tree. The TSR of an XML tree is calculated by preorder traversing the XML tree and appending in each step the string encoding of every node reached. For example, the TSR of the XML tree presented in Figure 1 is `&A0 AB1 BB2 BC2 A*1 *D2`. The depth information is stored in order to avoid ambiguity in cases of nested nodes with the same label. For example, in Figure 1, if the TSR didn't store the depth for every node, the node `D` could be child of either the two `B` nodes.

2.2 Distance between user profiles

In order to construct clusters of similar user profiles, we need to define a measure of the distance between two given user profiles. Previous works in this area propose the tree edit distance as a measure of the distance between any two labeled trees. The tree edit distance counts the cost of the total number of simple edit operations required to transform one tree to another. Initially, each edit operation costs 1, but someone can assign different costs in every edit operation.

The following simple edit operations are allowed:

- *Delete*. Deletion of a single node.
- *Insert*. Insertion of a single node.
- *Replace*. Replacement of an existing node with another one.

The semantics of user profiles require some modification to the above measure. For this reason, we make the following assumptions:

- Delete operations are allowed only in leaf nodes or in ancestor/descendant cases and cost 1.
- Insert operations are allowed only in leaf nodes or in ancestor/descendant cases and cost 1.
- Replace operations are allowed everywhere, but cost as much as the minimum weight of the two corresponding nodes (replaced and replacement node). The intuition behind this is that the more descendants a tree node has, the more important is for the corresponding user profile semantics. However, if the replaced node has a `*` label, then the replacement cost is 0.
- The weight of a node v, denoted as $w(v)$ is the total number of nodes in the subtree rooted at v.

Ancestor/descendant cases correspond to the presence of a `*`-label. In those cases, it is allowed to delete/insert a parent/child node of a `*`-labeled node. Figure 2 depicts an example of the modified edit distance between two user profiles. In this case, in order to transform User Profile 1 into User Profile 2, the following edit operations are required:

- Insertion of the `E`-labeled node under the `*`-labeled node.
- Insertion of the `E`-labeled node under the `C`-labeled node.
- Replacement of the `*`-labeled node with the `B`-labeled node.

Following the previously mentioned assumptions, each of the first two edit operations cost 1, while the third edit operation costs 0. Hence the modified edit distance between the two user profiles is 2.

In order to calculate our modified tree edit distance between two user profiles, we need a distance metric that reduces the problem of calculating tree edit operations into that of calculating the sequence edit distance between user profiles TSRs. For this reason we employ a modification of the Levenshtein distance between the TSRs of user profiles. The original Levenshtein algorithm [16] (also called Edit-Distance) calculates the least number of edit operations that are necessary to modify one string to obtain another string. The most common way of calculating this is by the dynamic programming approach. A tableau is initialized measuring in the (m, n)-cell the Levenshtein distance between the m-character prefix of one with the n-prefix of the other word [20]. The tableau can be filled from the upper left to the lower right corner. Each jump horizontally or vertically corresponds to an insert or a delete, respectively. The cost is normally set to 1 for each of the operations. The diagonal jump can cost either one, if the two characters in the row and column do not match, or 0, if they do. Each cell always minimizes the cost locally. This way the number in the lower right corner is the Levenshtein distance between the words.

However, in TSR, every node is represented as a pair of char labels and an integer representing its depth, thus we modify the Levenshtein distance to consider pair of chars instead of single chars. The depth information is used

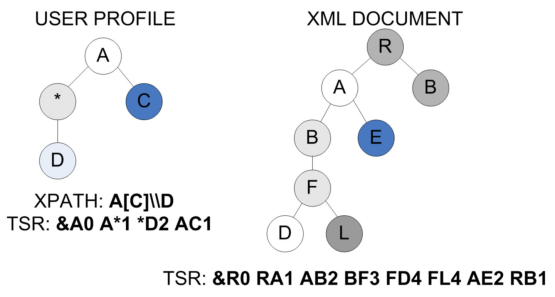

Fig. 3. Modified edit distance between a user profile and an XML document

only in the case of nested nodes with the same tag, in order to distinguish between them. Moreover, in order for the user profiles semantics to be fulfilled, we apply the previously mentioned assumptions to the modified Levenshtein distance algorithm. Table 1 presents the modified Levenshtein algorithm applied for the two user profiles in Figure 2.

Table 1. Modified Levenshtein Algorithm

		&A0	AB1	BD2	BE2	AC1	CE2
	0	1	2	3	4	5	6
&A0	1	0	1	2	3	4	5
A*1	2	1	0	1	2	3	4
*D2	3	2	1	0	1	2	3
AC1	4	3	2	1	2	1	2

2.3 Distance between an XML document and a user profile

The distance between an XML document and a user profile is measured in a similar manner with the distance between two user profiles. This fact is critical, as the filtering algorithm should be able to compare the distance between two user profiles and the distance between an XML document and a user profile. The only differences are the following:

– Delete operations in the side of the XML document cost 0.
– Replace operations cost as much as the weight of the user profile's corresponding node.

The above rules ensure that a user profile's distance from an XML document is 0 iff its tree representation is a subtree of the XML document's representation.

Figure 3 depicts an example of the modified edit distance between a user profile and an XML document. In this case, in order to transform the XML document into the user profile, the following edit operations are required:

- Deletion of the `R`-labeled node.
- Deletion of the `B`-labeled node (under the `R`-labeled node).
- Deletion of the `L`-labeled node.
- Replacement of the `E`-labeled node with the `C`-labeled node.
- Replacement of the `B`-labeled node and the `F`-labeled node with the `*`-labeled node.

Following the assumptions made before, the first three edit operations cost 0. The fourth edit operation costs as much as the weight of the `C`-labeled node, e.g. 1. Finally the last edit operation costs 0. Hence the modified edit distance between the user profile and the XML document is 1.

The distance is calculated again utilizing a modified Levenshtein distance algorithm based on the previously presented assumptions.

One crucial property of the use of the two previously described metrics is expressed in the following lemma:

Lemma 1. *Given two user profiles P_1, P_2 and an XML document D, suppose that $distance(D, P_2) = 0$. Then $distance(D, P_1) \leq distance(P_2, P_1)$*

Proof. Since $distance(D, P_2) = 0$, then a segment of the XML document matches exactly with P_2. Let us denote by S that segment and S' the rest of the XML document (excluding S). Let us consider $distance(S', P_1)$. There are two cases:

- $distance(S', P_1) \leq distance(P_2, P_1)$
- $distance(S', P_1) > distance(P_2, P_1)$

In the first case, we can delete S from D (deletion costs 0), and thus:
$distance(D, P_1) = distance(S', P_1) \Leftrightarrow distance(D, P_1) \leq distance(P_2, P_1)$
In the second case, we can delete S' from D (deletion costs 0). Thus,
$distance(D, P_1) = distance(S, P_1)$.
However, because S matches with P_2, we have:
$distance(S, P_1) = distance(P_2, P_1) \Leftrightarrow distance(D, P_1) = distance(P_2, P_1)$.
Thus in every case we have proved that: $distance(D, P_1) \leq distance(P_2, P_1)$.$\square$

The above lemma allows us to apply a clustering technique in order to reduce the filtering space and thus create an hierarchical filtering scheme as explained in the next sections. In particular, consider a cluster of user profiles, C, and its centroid profile P. The centroid profile is the profile that has the minimum average distance from the rest of the user profiles. In addition, consider that the most distant profile from the centroid is the profile O and its distance from P is d. Finally, consider an XML document D whose distance from P is r. If $r \geq d$, then based on Lemma 1, we can assume that there is no profile in the cluster C whose distance from D is 0. If there was such a profile Z, then its distance from P should be greater than r, based on Lemma 1. However, the most distant profile's distance from P is $d \leq r$, thus there is no such a profile as Z in the cluster C.

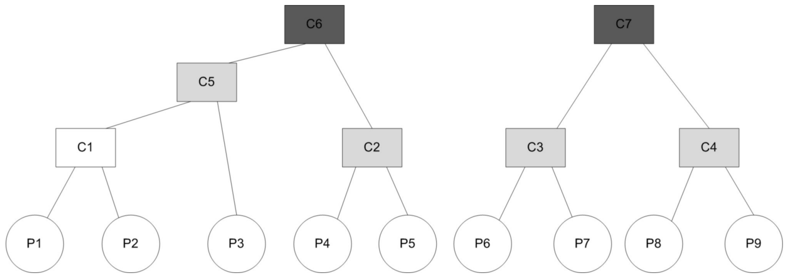

Fig. 4. Example of a cluster hierarchy forest

3 Filtering System

Our filtering system consists of two subcomponents: *User profile clustering* and *Filtering algorithm*. The user profile clustering process is activated once, when the system is initialized. When an XML document arrives, the filtering algorithm is invoked to find those user profiles that match with the XML document. The filtering algorithm utilizes the hierarchical structure of clusters formed at the clustering phase in order to find those user profiles that match with each incoming XML document.

3.1 User Profile Clustering

Our XML filtering system utilizes a modified hierarchical clustering algorithm, in order to form a cluster hierarchy. The proposed clustering algorithm is a classical hierarchical clustering algorithm which utilizes the previous described distance metric between two user profiles. Our clustering algorithm works as follows:

At first, every user profile is considered as a single cluster. In every step, the algorithm finds the two closest clusters and merges them in one cluster. For every newly formed cluster, the algorithm calculates the cluster centroid, which is that user profile which minimizes the average distance with the rest user profiles in that cluster. In addition, the algorithm calculates the `max distance`, which is the distance between the cluster's centroid and the most distant user profile inside the cluster. The `max distance` will be utilized during the filtering process, described in Section 3.3. Finally, the clustering algorithm keeps track of the cluster hierarchy, thus every formed cluster points to the two clusters it was formed by. The clustering algorithm stops until only two top-level clusters have remained. The result of our clustering algorithm is a cluster hierarchy forest (with two root nodes) in which every node has exactly two children nodes (expect of the leaf nodes). Every node u of that tree stores pointers to every-leaf node (e.g. user profile) contained in the subtree rooted at node u, a pointer to its centroid profile and its `max distance`.

Figure 4 shows an example of a such a cluster hierarchy forest. The clusters `C6` and `C7` are the two root nodes of the forest. As it can be seen, every cluster in the forest has exaclty two children, except of the low-level clusters (`P1` through `P9`) which represent the stored user profiles. Every cluster in the forest stores pointers to its low-level user profiles, thus, for example, cluster `C5` has pointers to user profiles `P1`, `P2`, and `P3`. Due to space limitations, the `max distance` and centroid profile for every cluster are not shown into the figure.

3.2 Filtering Algorithm

The filtering algorithm is used in order to filter an incoming XML document through the previously described cluster hierarchy forest. The result of the filtering process is a list of user profiles that match with the incoming XML document. The incoming XML document is then forwarded to the corresponding users of the matched profiles.

Before describing the filtering algorithm, we give two important definitions which will be used through out the filtering algorithm.

Definition 1. *A user profile p matches with an incoming XML document D iff $distance(p, D) = 0$.*

The distance between a user profile and an XML document is measured as described in Section 2.3, thus the above definition ensures that a user profile matches an XML document iff its corresponding tree representation is a subtree of the XML document's tree representation.

Definition 2. *An XML document D matches with a cluster of user profiles C iff $distance(D, c) \leq m$, where c is the cluster's centroid and m is the cluster's `max distance`.*

The above definition is based on Lemma 1. So, if $distance(D, c) \leq m$, then it is possible that the cluster C contains a user profile that matches with D. On the other hand, if $distance(D, c) > m$, it is not possible that the cluster C contains a user profile that matches with D. Thus, if an XML document D matches with a cluster C, then it should be filtered through all the user profiles contained in C. Otherwise, we can ignone the unmatched cluster and all its user profiles. This notion is exploited by our filtering algorithm in order to dramatically decrease the filtering space, thus achieving much better filtering time than the other filtering algorithms.

The proposed algorithm utilizes a list of active clusters, called `activeList`, which at any moment contains all the clusters that should be checked against the next incoming XML document. This list is updated after an XML document has been filtered as described later. In addition, the filtering algorithm adds a counter called `matchCnt` in every cluster of the hiearchy tree. This counter counts how many XML documents have been matched with the corresponding cluster. Finally, the filtering algorithm initializes an extra global counter, called `totCnt`, that counts the total number of XML documents that have been filtered.The filtering process is as follows:

When first initialized, the `activeList` contains only the two top clusters in the hierarchy forest. As a result, the first incoming XML document will be checked against the two clusters in the `activeList`. At any step of the filtering process, every incoming XML document is checked only against the clusters of the `activeList`. For every matched cluster, the filtering algorithm increases its `matchCnt` and then filters the XML document against all the user profiles contained in that cluster, by simply calculating the distance between the XML document and every user profile as described in Section 2.3. Every profile that matches with the XML document is added to the output resultset of profiles. However, the process of filtering an XML document with all the user profiles within a cluster is the bottleneck of the filtering algorithm because it requires checking the XML document with all the user profiles. Based on this notion, we propose a dynamic filtering process which takes into consideration the number of matchings per cluster and updates accordingly the `activeList`. The intuition is that if a cluster has a lot of matchings, then the matched XML documents are always checked against all its user profiles, thus if we want to reduce that cost, we should split that cluster. In the same manner, if a cluster has very few matchings, then we can eliminate the cost of checking every incoming XML document with its centroid by merging that cluster with its sibling cluster. Thus, after an XML document has been filtered, the filtering algorithm checks the value of $\frac{\texttt{matchCnt}}{\texttt{totCnt}}$ of all the clusters contained in the `activeList` and compares them with two thresholds: `topThr` and `bottomThr`. If the $\frac{\texttt{matchCnt}}{\texttt{totCnt}}$ for a cluster C is greater than `topThr`, then we remove C from the `activeList` and insert into the `activeList` the two children of C. On the other hand, if the $\frac{\texttt{matchCnt}}{\texttt{totCnt}}$ for a cluster C is less than `bottomThr`, we remove C and its sibling from the `activeList` and insert into the `activeList` the parent cluster of C. Thus, in every step of the filtering process, we try to eliminate the cost of checking an XML document with the centroids of the clusters in the `activeList` and the cost of filtering an XML document with all the user profiles within a matched cluster.

For example, consider the cluster hierarchy tree presented in Figure 4. Initially, the `activeList` contains the clusters `C6` and `C7`. Thus every incoming XML document is checked against those clusters and if it matches with one or both of them, it is filtered through the user profiles of the matched cluster(s). After a few XML documents have been filtered, suppose that the value of $\frac{\texttt{matchCnt}}{\texttt{totCnt}}$ for the cluster `C6` has exceeded the `topThr`. In such a case, the filtering algorithm removes `C6` from the `activeList` and inserts the clusters `C5` and `C2` into the `activeList`. Thus, every incoming XML document is now checked against clusters `C5`, `C2` and `C7`.

The values of `topThr` and `bottomThr` vary between 0 and 1 and are not strictly defined and can be adjusted accordingly to the needs of every system. We propose the following indicative values: `topThr` $= 0.3$ and `bottomThr` $= 0.05$.

4 Experiments

We tested our filtering system against FiST[12], which is one of the state-of-the-art algorithms for filtering XML documents against twig pattern user profiles. We chose FiST because it supports twig pattern user profiles, unlike other systems (e.g. AFilter, XFilter) which support only linear path expressions. Our

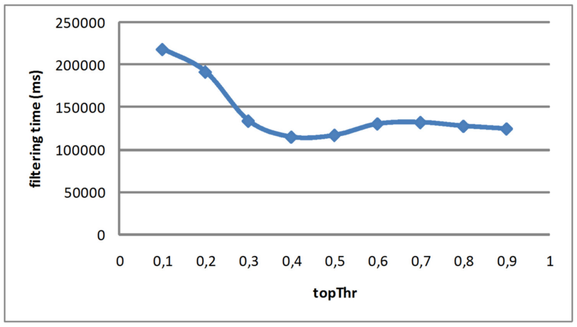

Fig. 5. Filtering time in relation with topThr threshold

filtering system was implemented in Java using the freeware Eclipse IDE[18]. In order to obtain comparable and reliable results, we also implemented a FiST-like algorithm in Java using Eclipse.

In our experiments we used three different datasets: the DBLP dataset[15], the Shakespeare's plays dataset[17] and the Sigmod Record dataset[21]. For each of those datasets, we also generated a random number of user profiles with arbitrary depth and fan-out.

Our first experiment was to investigate the influence of the `topThr` threshold in the performance of our algorithm. For that purpose, we disabled the checking for the `bottomThr` threshold and we used our algorithm to filter 100 documents through 1000 user profiles. Both utilized documents and user profiles were arbitrarly selected from the 3 aforementioned datasets. The initial value of `topThr` was 0.1 and in each step of this experiment we increased `topThr` by 0.1 until it became 0.9. We measured the total filtering time of 100 documents required by our algorithm in each step and we present the results in Figure 5. As we can see, the total filtering time for the 100 XML documents decreases as the value of `topThr` increases, until `topThr` reaches 0.4. At that point, the filtering time has its global minimum value (approximately 115000ms). After that point, the filtering time starts to increase again as the value of `topThr` increases. This trend of the filtering time can be explained as follows: at first, when `topThr` has a low value (e.g 0.1, 0.2), the filtering algorithm acts aggressively and splits very often the clusters belonging to `activeList`, thus it moves deep in the cluster hieararchy. As a result, the size of the `activeList` becomes very large and each incoming document is always cheched against a large number of low-level clusters. However, as the value of `topThr` increases (e.g 0.3, 0.4), the filtering algorithm becomes less aggressive and splits fewer clusters, resulting in a smaller `activeList`. Every incoming XML document is now checked against a moderate number of top-level clusters, thus the total filtering time is reduced and reaches its minimum value when `topThr` becomes 0.4. However, after that point and as the value of `topThr` continues to increase, the filtering algorithm acts more conservatively and rarely splits the clusters contained in `activeList`. Although,

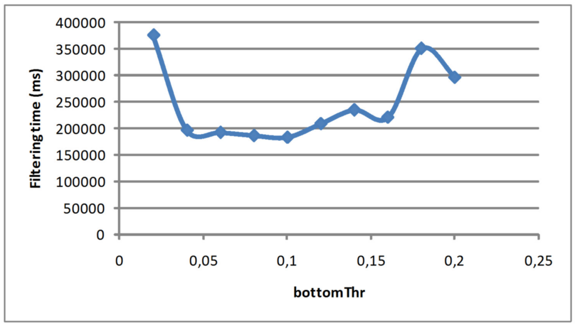

Fig. 6. Filtering time in relation with bottomThr threshold

the size of `activeList` remains very small, the clusters contained in `activeList` are very large (as they are not splitted easily) and whenever an incoming XML document matches with a cluster in `activeList` it is filtered out through all the profiles belonging to the corresponding cluster, thus requiring more time to be filtered.

Our second experiment was to investigate the influence of the `bottomThr` threshold in the performance of our algorithm. For that purpose, we standarized `topThr` to 0.4, while in each step of that experiment we incremented `bottomThr` by 0.02 (starting from 0.02) until it reached 0.2 (half value of `topThr`). We measured the total filtering time of 100 documents required by our algorithm in each step and we present the results in Figure 6. As we can see, the filtering time behaves in a similar manner with the first experiment: it decreases until `bottomThr` becomes 0.1 and increases after that point. The explanation behind this is that low values of `bottomThr` result in a very conservative filtering algorithm which rarely merges two clusters of the `activeList`, thus the size of `activeList` never decreases even if some of its clusters are rarely matched with an incoming XML documents. However, as the value of `bottomThr` increases, the filtering algorithm starts to merge more easily rarely matched clusters, thus the `activeList` contains less but more popular clusters, thus the filtering time decreases. Further increase of `bottomThr` (> 0.1) results in an aggresive merging of clusters contained in `activeList`, thus the `activeList` contains only some few top-level clusters, which in turn results in a lot of cluster matchings for every incoming XML document. Those matchings increase the filtering time, as the XML document has to be filtered out through all the user profiles contained in every matched top-level cluster.

Our third experiment was to compare our proposed algorithm with Fist, one of the state-of-the-art XML filtering algorithms. During this experiment we measured the total time required by the two algorithms for filtering a set of 200 XML documents. We varied the number of stored user profiles between 200 and 1000 in order to investigate the relation between the number of user profiles

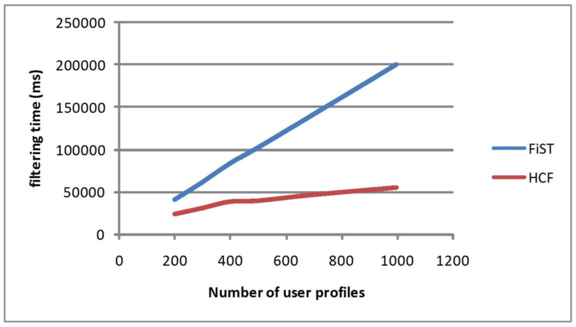

Fig. 7. Filtering time required for 200 documents by FiST and our algorithm (HCF)

and the required filtering time. The results of this experiment are presented in Figure 7.

According to Figure 7, our algorithm (referred as HCF) outperforms FiST in every step of that experiment and the difference in filtering time increases dramatically as the number of user profiles grows. In particular, in the case of 200 user profiles, our algorithm is 41% faster while in the case of 1000 user profiles, our algorithm is 72% faster. Another important notion is that FiST requires linear filtering time, while our algorithm requires sub-linear time. The effectiveness of our algorithm is due to the reduction of the filtering space achieved by employing the cluster hierarchy forest. As a result, the number of user profiles needed to be checked against every incoming XML document is very small related to the total number of stored user profiles.

5 Conclusions and Future Work

In this paper we have presented a new XML filtering system that uses clustering of user profiles in order to scale well as the number of user profiles grows. The proposed system utilizes a unique sequence representation for user profiles and XML documents, based on the preorder traversal. Based on this sequence representation, we proposed a modification of the Levenshtein distance metric in order to calculate the distance between two user profiles or between an XML document and a user profile. The proposed metric reduces the problem of calculating the tree edit distance into that of calculating the modified Levenshtein distance between the sequence representations. Our system applies hierarchical user profile clustering in order to succeed sub-linear filtering time, based on the number of matchings per cluster. Our experimental results showed that the proposed system outperforms the previous algorithms in XML filtering and requires sub-linear time to filter the incoming XML documents.

As future work, we intend to compare our filtering algorithm with more approaches (such as AFilter, YFilter etc) as well as to utilize alternative clustering techiques such as k-Means; moreover, we aim to extend our filtering algorithm in order to additionally support value-predicate user profiles instead of only structural user profiles.

6 Acknowledgements

Panagiotis Antonellis' work was supported in part by the Hellenic State Scholarships Foundation (IKY).

References

1. Aguilera, M. K., Strom, R. E., Stunnan, D. C., AsHey, M. and Chandra, T. D: *Matching Events in a Content-based Subscription System.* PODC 1999, 53–61
2. Altinel, M. and Franklin, M.l J.: *Efficient Filtering of XML Documents for Selective Dissemination of Information.* VLDB 2000, 53–64
3. Antonellis, P., Makris, C. and Tsirakis, N.: *XEdge: Clustering Homogeneous and Heterogeneous XML Documents Using Edge Summaries.* ACM SAC 2008 (to appear)
4. Antonellis P. and Makris, C. "XFIS: An XML Filtering System based on String Representation and Matching", International Journal on Web Engineering and Technology (IJWET) 4(1), 70–94,2008
5. Canadan, K., Hsiung, W., Chen, S., Tatemura, J. and Agrrawal, D.: *AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering.* VLDB 2006, 559–570
6. Dalamagas, T., Cheng, T., Winkel, K. and Sellis, T.: *Clustering XML documents using Structural Summaries.* EDBT Workshop 2004, 547–556
7. Diao, Y., Altinel, M., Franklin, M.l J., Zhang, H. and Fischer, P. *Path sharing and predicate evaluation for high-performance XML filtering.* TODS 2003, 28(4) 467–516.
8. Francesca, F., Gordano, G., Ortale, R.and Tagarelli, A.: *Distance-based Clustering of XML Documents.* MGTS 2003, 75–78
9. Gupta, A.K and Suciu, D *Stream processing of XPath queries with predicates.* SIGMOD 2003, 419–430
10. Isert, C.: *The editing distance between trees.* Technical Report, Ferienakademie, for course 2: Bume: Algorithmik Und Kombinatorik, Italy, 1999
11. Jain, A.K. and Dubes, R.C.: *Algorithms for Clustering Data.* Prentice-Hall, 1988.
12. Kwon, J., Rao, P., Moon, B. and Lee, S.: *FiST: Scalable XML Document Filtering by Sequencing Twig Patterns.* VLDB 2005, 217–228
13. Peng, F. and Chawathe, S.: *XSQ: A streaming XPath Queries.* TODS 2005, 577–623
14. Zhang, K. and Shasha, D.: *Simple fast algorithms for the editing distance between trees and related problems.* SIAM Journal on Computing, 1989, 1245–1262.
15. http://kdl.cs.umass.edu/data/dblp/dblp-info.html
16. http://www.levenshtein.net/
17. http://xml.coverpages.org/bosakShakespeare200.html
18. http://www.eclipse.org
19. http://www.w3.org/TR/xpath
20. http://www.levenshtein.net
21. http://www.dia.uniroma3.it/Araneus/Sigmod/Record/DTD/index.html