WFilter: Efficient XML Filtering for Large Scale Publish/Subscribe Systems

Raul Martins and João Pereira

Instituto Superior Técnico,
Avenida Professor Cavaco Silva, 2780-990 Porto Salvo, Portugal
{raul.martins,joao.d.pereira}@ist.utl.pt
http://www.ist.utl.pt

Abstract. Currently, there is a very high amount of dynamic information on the web, bringing the need to find efficient ways to help users select the information they are interested in as soon as it is published. A common solution to cope with this problem is to employ publish/subscribe systems. However, in order to be efficient, these systems must solve the matching problem: compute the subscriptions matched by a given piece of information. Usually, using XML as a way to represent published information entails the usage of XPath as a subscription language. Due to XPath's high expressiveness, solving the matching problem in an XML context efficiently becomes more complex, and as such, common solutions usually reduce the expressiveness of the subscription language to a small subset of XPath in order to achieve higher efficiency. In this paper we propose an algorithm that, based on a value-based matching algorithm, can efficiently solve the filtering problem in a XML publish/subscribe environment, while supporting a significantly expressive subset of the XPath language.

Keywords: Information Filtering, XML Filtering, Publish/Subscribe, Information Dissemination, Algorithm

1 Introduction

The Internet acts a world-scale global distributed system of interconnected computers storing and providing information to users all over the world. Due to the magnitude of the Internet, we can consider it as an astronomical ever-growing repository of new information. At the same time, the number of users interested in some of that information is also growing. Because of this, the accessibility of the information and the easiness of publishing new information has increased. This has brought attention upon ways of selecting which information concerns which users. One possible way is: having a system as an intermediary between users and information. This allows users to specify their interests and, as the information flows in, automatically deliver the desired information to the interested user, providing a way to help users filter which information is relevant to them.

The systems that provide the aforementioned filtering are called Information filtering systems [1]. These systems apply automated methods to filter information streams determining which information is relevant to which users. Their main concern is to prevent information overload and to spare interested users of having to find information by themselves. In order for this to happen, users must explicitly specify their interests, through the use of a specific language, or the language might be determined implicitly based on the users behavior. The main focus in this paper is on the first situation where the user interests are stored in the system as a set of subscriptions. As information streams in, the verified subscriptions are processed and the filtering system can send the relevant information to the users.

These filtering systems work based on a publish/subscribe paradigm. In the context of a publish and subscribe system a new piece of information is designated as an event. In a context where the event's information is represented in XML [2], the subscription language used is always an XML query language, the XML querying languages usually address a path or a group of paths in an XML document. Currently, the most common form of path querying language for XML is XPath [3].

Taking into account that these systems can deal with millions of users, millions of subscriptions and hundreds of events per second, it's critical that they offer an efficient performance when determining the subscriptions matched by an event. This is called the matching problem, being able to determine which subscriptions are matched by an event in acceptable time. A key issue when using the aforementioned subscription language, XPath, is its high expressiveness, that is, its ability to accurately represent user interests. While this gives the users a more powerful mean to express their interests it also brings a drawback: the matching problem becomes more complex, and thus, harder to solve in acceptable time.

In order to be able to determine which events match which subscriptions, many solutions have followed the approach of restraining the expressiveness of the language to a pre-defined subset. In this paper we propose an algorithm that based on the techniques used by LeSubscribe [4], can efficiently solve the matching problem, while handling a considerably complete subset of the XPath language. The key contributions of this paper can be summarized as follows:

- We present a new algorithm to perform XML Filtering called WFilter. This algorithm combines structural matching and attribute expressions matching in a single strategy
- Our algorithm explores commonality in both structural query steps and attribute expression query steps
- By storing subscriptions into clusters, WFilter is able to greatly reduce the universe of possible solutions for each incoming event
- Experimental results show that our solution significantly outperforms YFilter [6], a state-of-the-art algorithm, regardless of the subscription-set composition

The remainder of this paper is organized as follows. Section 2 provides an overview of current XML filtering solutions. Section 3 describes, the WFilter solution. Section 4 presents our experimental results. Finally, Section 5 concludes our work.

2 Related Work

There have been several proposed solutions for the matching problem in a XML Filtering context. Most of these solutions can be grouped in the following types: automata-based, sequence-based, stack-based and pushdown approaches.

The first solution to arise was XFilter [5], an automata based solution that handles almost the entire subset of the XPath language. It consists on converting each subscription into a finite state machine (FSM) where each location step is converted into a state. The state machines are then indexed in a hash table. A problem with this approach was that it did not take into account commonalities between subscriptions and therefore each subscription is matched against each incoming event completely isolated from the other subscriptions.

YFilter, deals with this problem by using a single non-finite-automata (NFA) resulting of the combination of all subscriptions into a single automata. This approach fully explores commonalities between subscription's prefixes, common subscription steps are only tested once, but, with an increase in distinct subscriptions, the states will proportionally increase, causing the search space to contain deep recursions.

In an attempt to avoid the state growth in automata based solutions, AFilter [8] attempted to use a simple stack based approach supporting itself on several indexing structures that would reduce the size and enhance the speed of the matching process. Although it's faster than the previous automata-based approaches, it does not support attribute expressions in their queries.

Other solutions are, FiST [7] and XFIS [10], which focus on improving the performance of processing a certain type of subscriptions called twig expressions or nested path expressions. They convert the subscriptions into a special type of string and then use techniques usually employed in efficient string matching matching to achieve efficiency in the matching process. Yet, none of these solutions support attribute expressions. FiST however, has an upgraded version called pFiST [9], which is able to handle attribute expressions, although its structural matching component still sacrifices processing speed on regular subscriptions in order to optimize the matching of the twig expressions.

There is also Predicate-based Filter [12], which follows a very different approach. Instead of attempting to explore the commonality between subscriptions, it converts each of them into a set of different predicates and then indexes them. It breaks the matching process into two phases, where in the first the commonality between those predicates is fully explored and a part of the matching process is computed severely simplifying the second phase. In this second phase no commonality is explored, it still determines matches efficiently due to a cluster indexing strategy and the work already done in the first phase. Predicate-based Filter

also handles attribute-based expressions by also converting them into predicates and matching them together with the structural expression steps.

3 Le Subscribe

3.1 Summary

LeSubscribe, is a matching algorithm designed for value-based publish/subscribe systems. In these systems, the information is always represented as a set of $\{attribute, value\}$ value pairs. Due to this, the publication language allows the description of attributes and their respective values. The subscription language offered by LeSubscribe system supports the specification of interests as a conjunction of predicates. A predicate defines a condition over an attribute. This algorithm works by indexing the subscriptions of the system in a clever manner by using a combination of data structures, and accessing them based on the received events.

3.2 Subscription Language and Event Representation

As previously stated, LeSubscribe is suited for matching processing in publish/subscribe value-based systems. Each event consists in a set of pairs containing a name and a value, in the form of $\{(atrribute_1, value_1)\ ;\ ..\ ;\ (attribute_n, attribute_n)\}$. An example of this could be the event $\{(a,\ 5)\ ;\ (b,\ 6)\ \}$, it represents an event with two attributes, a and b, where attribute a has the value 5 and b has the value 6. The subscription language supported by this matching algorithm will consist of a set of predicates applied to the event attributes. Each subscription consists in a set of predicate expressions in the form $\{(atrribute_1\ operator_1\ value_1)\ ;\ ...;\ (atrribute_n\ operator_n\ value_n)\}$. The available operators that can be present in a predicate are *equality*, *less-than*, *greater-than*, *less-than-equals* and *greater-than-equals*.An example of a subscription could be $S = \{(a = 5)\ ;\ (b > 7);\ (c < 7)\ \}$.

3.3 The Algorithm

The algorithm works by assigning a distinct identifier, called predicate id, to each unique predicate and indexing it in a **predicate index**. There are several predicate indexes, called **predicate families**. They group together all the predicates that have the same attribute and operator. Le Subscribe also indexes the result of evaluating those predicates in a vector, the **predicate vector**. For any event, the **predicate vector** stores the results of predicate evaluation for that event. The entry of the predicate vector that corresponds to the result of evaluating a predicate of id n is the position n of the vector. This entry is set to 1 if the predicate was satisfied and set to 0 otherwise. When a predicate is verified it is added to the **access predicate vector**. This way, after each event, there is a vector containing all the matched predicates.

The subscriptions are indexed together with their predicates. They are grouped in clusters based on their size and a single common-predicate[1] which we will call the *accesspredicate*.

Since subscriptions are stored in clusters based on heir size and an *accesspredicate*, one can also characterize the clusters based on those two factors. So, clusters of different sizes but that share the same *accesspredicate* are grouped into cluster families and indexed in a vector called **cluster vector** based on their *accesspredicate*.

As an incoming event arrives, the algorithm processes it in two phases:

In the first phase it starts by initializing the contents of the ***predicate vector*** to 0 in all positions. Then, given an input event $e = \{(attribute_1, value_1) ; (attribute_2, value_2)\}$, this phase computes the predicates satisfied by that event e. This is done by accessing each of the predicate *families* for $attribute_1$ and $attribute_2$ and using $value_1$ and $value_2$ as keys on the respective indexes. In each of the indexes the algorithm will search for the predicates satisfied by the respective $value_n$. Then, all of those predicates will have their entries in the **predicate vector** set to 1.

The second phase consists in traversing the **access predicate vector** and for each predicate, access the **Cluster Vector** and verify if there are any clusters indexed by that *accesspredicate*. In case there is, at least, one cluster indexed by that *accesspredicate*, it is considered a candidate cluster and must have all its subscriptions evaluated. This is done by traversing each of the predicates in the subscription and checking its value in the predicate vector. If it is 0 then the subscription is not matched and the algorithm does not need to consider the remaining predicates of the subscription. When it is 1, the algorithm needs to evaluate the next predicate. If this was the last predicate, then the subscription is matched by the event. Note that, the *accesspredicate* is not indexed with the subscriptions. This is due to the fact that we previously know when verifying a cluster, that, that cluster's *accesspredicate* is already verified.

Figure 1 shows the main data structures used by this algorithm and how they relate.

4 WFilter

4.1 Summary

WFilter is an algorithm suited to perform XML filtering in a publish/subscribe XML context. In this context, events are represented in the form of XML Documents and the subscriptions are expressed in the form of a XML path query language. In this case, we will be using the commonly used query language XPath.

This algorithm is based on the LeSubscribe unidimensional solution presented in Section 3. Our main goal was to take advantage of the efficient processing capacity of the LeSubscribe indexing techniques. Due to a higher expressiveness

[1] This is because we are only considering the uni-dimensional solution of Le Subscribe

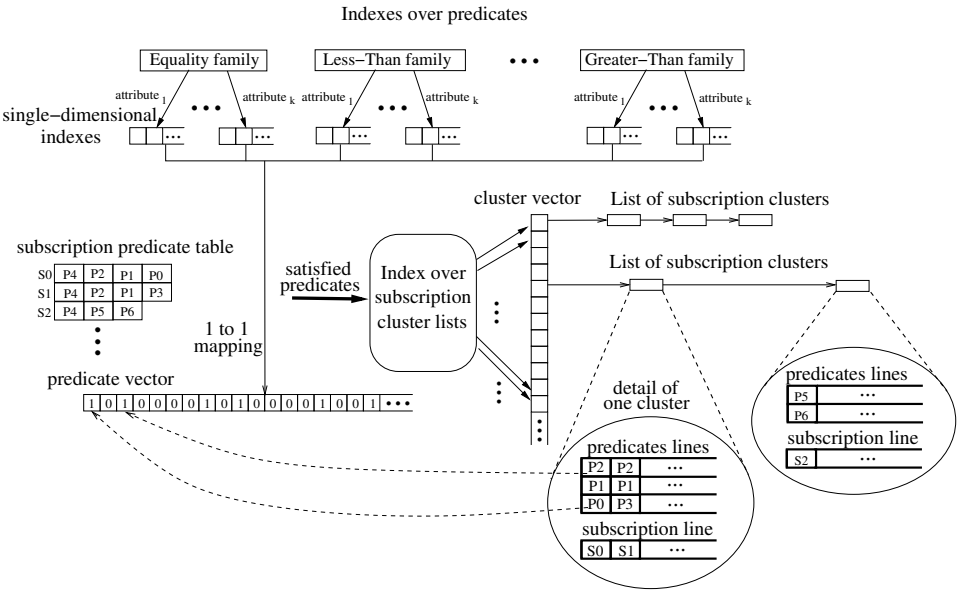

Fig. 1. LeSubscribe data structures.

and complexity of this kind of systems, it's necessary to simplify the matching problem by decreasing the complexity of each event. This way, each subscription in our system is a XPath expression, that can only represent exactly a path or a sub-path in a XML Document. In order to do this, we broke down each event, which we will call external event from now on, into several others smaller events, which we will call internal events. Each internal event represents a unique path from the root node to a leaf node of the XML document represented by the external event. This means, there will be as much internal events as there are leafs in the XML Document. Our algorithm processes those generated internal events.

However, reducing the complexity of the events is not enough, as, LeSubscribe works with subscriptions consisting of conjunctions of predicates. Therefore, there is also a need to convert the subscriptions expressed in XPath into a set of predicates. An example of this can be seen on Example 1.

Example 1. *A predicate based representation could consist of predicates representing the position of the nodes in the path. Each node would generate a predicate expression comparing its position to a value or another predicate. Let's assume the following subscription represented in XPath,* s_o *= [/ a // b / * / d]. We could represent it as* $s_c = [\, pos_a = 0 \;\&\&\; pos_b > pos_a \;\&\&\; pos_d = pos_b + 2]$.

As shown in Example 1, this subscription representation would lead to a very high amount of predicates and some of the predicates would not be easily indexable since they involve a condition over two attributes. This would imply

a high cost to determine the predicates satisfied by an event. Another problem, is that in XML paths, the same node can appear more than once, in the same path which would render the previously presented approach ineffective and would make this approach even more complex, requiring some way to keep track of which node satisfied which predicate. Thus, we decided to take a different approach.

This approach was similar to LeSubscribe. The key differences are the different kinds of predicates, the way the subscription steps are all indexed in a single index instead of in a family, the additional index to handle attribute expressions ,the *predicate vector* represent for each predicates the positions in the path being processed where the element associated to the predicate appears, instead of just 0 or 1, and the technique to determine if a subscription is actually matched, all of this differences will be detailed further in this paper. For simplicity sake, we will first explain our solution assuming subscriptions containing only wildcards("*"), parent-child axis("/") and ancestor-descendant axis("//"). Section 4.2 addresses how we convert the subscription location steps into special predicates. Section 4.3 will introduce the data structures changes from LeSubscribe. This will be followed by an explanation of the algorithm itself on section 4.4. Finally, section 4.5 will detail how we incorporate XPath attribute expressions("[]") into our solution.

4.2 Internal Subscription Representation

Every node addressed in a XPath expression is called a location step. WFilter, converts each location step into a special internal representation that we will call predicate (despite the fact that they are not actually predicates). Basically, the *root predicate* represents the location step addressing the root node and is set to 1 in case that node appears in the current event path. The special *wildcard predicate* is represented by a special predicate whose entry in the predicate vector will have the first n bits set 1, where n equals the current event's *path depth*. Finally, the *predicate position* which represents the location steps following the root and addressed with parent-child("/") axis and ancestor-descendant axis("//") and consists of a 64-bits number, and each time that location step appears in the event path in the position n will have its *nth* bit set to 1. When stored, the *position predicate* will have either a negative or a positive identifier. This works as a flag indicating wether it refers to the parent-child("/") axis or to the ancestor-descendant axis("//").

4.3 Data Structures

The WFilter algorithm is composed of the same key structures as LeSubscribe. However, some of them are implemented in a different way.

The index over the predicates is used to efficiently determine the *predicate identifier* of all predicate types that correspond to a node of the matching internal event . For example, consider the subscriptions s_1 = *[/ a / b]* ; s_2 = *[/ a // b]* ; s_3 = [// b / a] and event e_1 = *(a -> b)*. When the algorithm processes

each of the XML nodes in e_1, node a maps to more than one predicate type as it appears as root in subscription s_1 and s_2 and as a regular step in s_3. In this case node a would map to a *root predicate* and a *position predicate*, while b will only map to a *predicate position*. Basically, this index stores for each element at most two predicate identifiers: one corresponding to a root predicate, if there is at least one subscription containing that predicate. The other corresponds to a position predicate if there is at least a subscription containing the position predicate that concerns that element.

The subscriptions are also organized in different clusters. These clusters work exactly the same way as the ones used in LeSubscribe except for two key changes. One is that when the id corresponds to a predicate of type *predicate position* and addresses an ancestor-descendant relationship, its id is stored as an negative id in the cluster. As stated before, this works as flag, for the matching algorithm to detect that the predicate needs to be handled differently. The second difference is that each predicate, represent more than one condition. One is explicit and indicates that a node exists in the current internal events. The other is implicit and concerns the position in which a node appeared relative to other nodes. This implicit condition can be verified by a set of operations that are performed between all the predicates in a subscription. If an access predicate verifies the explicit condition, that cluster will have to be verified. Thus, because the implicit condition between the predicates still need to be checked, all the predicates are required to take part in the matching process, including the access predicate. So, for a match to be determined, we need to compute the results of each predicate relative to the previous one[2].

4.4 The Algorithm

WFilter processes each of the aforementioned internal events in three phases: predicate matching phase, cluster selection phase and matching phase.

Predicate Matching Phase In this phase WFilter verifies which of the predicates are matched by the current internal event. So, every node that appears in at least one subscription will be converted to a *predicateid* using the predicate index. Recall that an event consists of a $(nodename, nodedepth)$ pair. So, we must add a match representing that the *predicateid* corresponding to a *node* appeared at depth *nodedepth*. This is done by setting the bit number *nodedepth* in the predicate vector entry *predicateid*.

Cluster Selection Phase Now that all matched predicates and respective depths are stored on the predicate vector, now the clusters that may have subscriptions that can verify the current event must be selected. This is done like in Le Subscribe. In this case, any predicate id that has at least one 1 in its bit-chain will be considered as a matched predicate.

[2] This obviously doesn't apply to the root node.

Subscription Verification Phase Then, we must evaluate each of the subscriptions contained in the previously selected candidate clusters. All the subscriptions in each candidate cluster must be checked, either until they are matched or until one of their predicates does not match the respective operator, at which point we can skip the remaining predicates of that subscriptions and proceed to the next subscription in the cluster. The specific algorithm to check each subscription of each cluster takes care of determining which operators it should apply between the predicates of a subscription based on the previously set flags. We implemented this operators as the following set of bitwise operations:

pos_b parent pos_c $(../b/c) \rightarrow$ This operation must return the positions where predicate pos_b was verified right before predicate pos_c. So, we implemented it as $pos_b << 1$ & pos_c.

pos_b ancestor pos_c $(../b//c) \rightarrow$ This operation must return the positions where predicate pos_c was verified at any position after the predicate pos_b. So, we implemented it as pos_c & $BinaryMask(pos_b)$, where binary mask returns a 64-bit number that has the most significant bits set to 1 until the first set position in pos_b and the rest of bits are set to 0. This mask basically removes all the occurrences of node c before node b and leaves unchanged all the occurrences of node c after node b.

pos_a verifyAttribute $pos_i d_a$ $(a[@id = 7]) \rightarrow$ In this case we just need to verify if the attribute was matched at the same depth as the node, so it's implemented as pos_a & $pos_i d_a$.

4.5 Handling Attributes

So far, we have presented our solution considering only the XPath subset containing { / , // , * }. However, one of the key goals of our algorithm was to be able to encompass a large subset of the XPath language which would also include the attribute expressions ("[@]"). This kind of expressions are in fact very similar with the predicates used in LeSubscribe. This approach allowed us to adapt the techniques used by LeSubscribe directly. So, we use the same concept of predicate families.

Exploring commonality in this kind of predicate is more complicated, as a unique attribute expression is composed of several factors: the attribute being addressed, the structural node to which it belongs, the operator and the comparison value. To handle this, we consider that a unique attribute expression is composed by the tuple {*structural node,addressed attribute*}. Each of this unique tuples is assigned a unique identifier called **attribute id**. This identifier is then used on each *family* to obtain the matched predicate ids The comparison values are stored alongside with the predicate identifier that represents the predicate composed of that particular combination {*structural node, addressed attribute, operator, comparison value*}.

Predicate identifiers representing attribute expressions are generated based on the combination {*structural node, addressed attribute, operator, comparison value*} and are stored on subscriptions in the same way as the others. However, the bit 64 of their number is set to 1 to work as a flag as they also need special treatment. They need to have the same position set to 1 as the structural node they belong to. This means we just need to perform a *binary AND* between the attribute predicate and the last

predicate. here is a sample the algorithm[3] to process the subscription clusters for all cases:

```
1: for(j=0;j < clustersize; j++) {
2:   res = predicatevector[cluster[0][j]];
3:   for (i=1; res != 0 && i < subsize; i++) {
4:    predid = cluster[i][j];
5:    if (predid > 0) {
6:       next = predicatevector[predid];
7:       if (next > 0)
8:         res = (res << 1) & next; //regular case
9:      else
10:       res = res & -next; //attribute case
11:   }
12:  else { //ancestor-descendant
13:     next = predicatevector[-predid];
14:     res = next & PredicateConstants.getMask(res);
15:        }
16:   }
17: }
```

Code Sample 1. The algorithm for processing a subscription cluster

4.6 Optimization: Reducing amount of events

As previously shown, in order to adapt the events of an XML context we had to break each external event into several internal events. Furthermore, the whole matching process has applied to each event. Considering this, reducing the number of internal events would result in noticeable performance improvement. Unfortunately, every internal event contains a part of unique information and a part of redundant information. This means that, each internal event contains a part of indispensable information and as such none of the events can be discarded. Moreover, the redundant information is also useful. It allows a completely independent handling of each internal event.

The only remaining solution was to combine the information contained in different events. However, this would go against the previous purpose of breaking them down. Despite this, there's a case where the matching process remains the same. When dealing with events that only differ on the leaf node, we can combine them into one, as long as the leaf nodes have different tag names. In the case where you have multiple nodes on the same depth, the matching process only becomes more complicated because it generates false positives that require verifying if the node that matched as one level above other in terms of depth, also has the same breadth. The filtering of this false positives is costly and is not needed for leaf nodes as it is guaranteed that there will be no nodes after them and as such there will be no false positives.

5 Performance Analysis

In this section, we will provide the result of our experimental study with a performance comparison with the algorithm YFilter. We considered YFilter an adequate choice for

[3] Certain optimizations were left out for the sake of a better understanding of the code

the comparison analysis as it is state-of-the-art when regarding such a large subset of the XPath language, and its authors made available the implementation in java.

We implemented WFilter in Java(Java 6) and conducted all experiments in a 2.3 GHz Intel Core i5 processor with a L2 cache of 256 KB and a L3 cache of 3 MB and 4 GB of 1333 MHz DD3 ram memory. For all the experiments we used the nitf dataset included in the YFilter code package and the respective query generator. In this environment, we performed several experiments of which we will present the most relevant results. First we started by measuring our performance in sets of subscriptions containing only the parent-child("/") and the ancestor-descendant("//") operators.

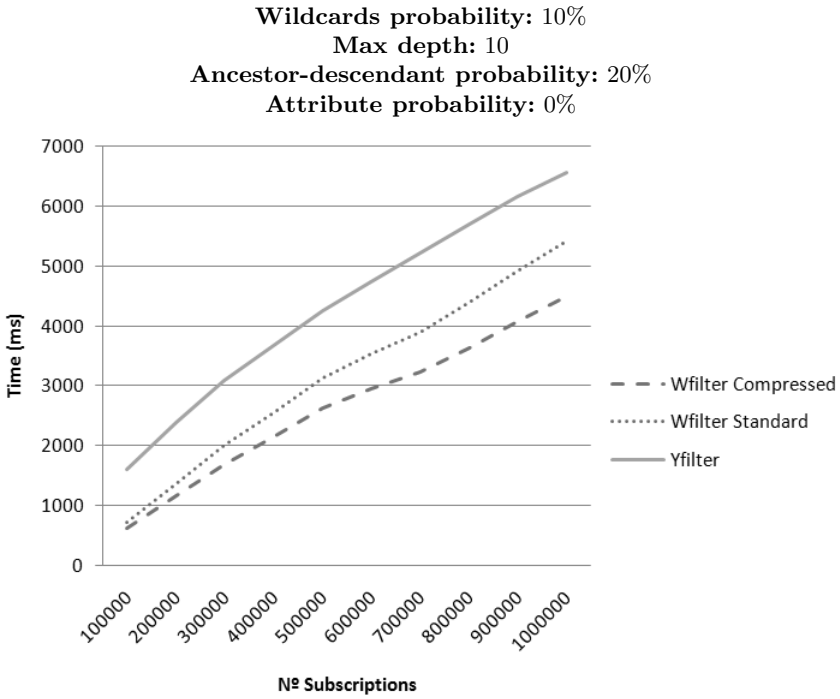

Fig. 2. Performance on increasing amount of subscriptions with no attributes

The case where subscriptions only contain structural is best suited for YFilter, as its NFA is designed with only structural subscriptions in mind. For WFilter, the fact that there are no attributes, reduce the selectivity of the possible access predicates.

In this case, we can observe on Figure 2 that both WFilter and YFilter scale equally well. So, WFilter shows a consistent improve, performing around 60% to 100% faster than YFilter for an increasing number of subscriptions.

Our next experiment includes the entire subset of the language WFilter can handle, it includes attribute expressions combined with the structural expressions. By balancing out the operators occurrence in subscriptions, we tried to create a subscription base as similar to real as possible. It contains all the possible operators combined in a random way on subscriptions with random size. The results are shown in Figure 3.

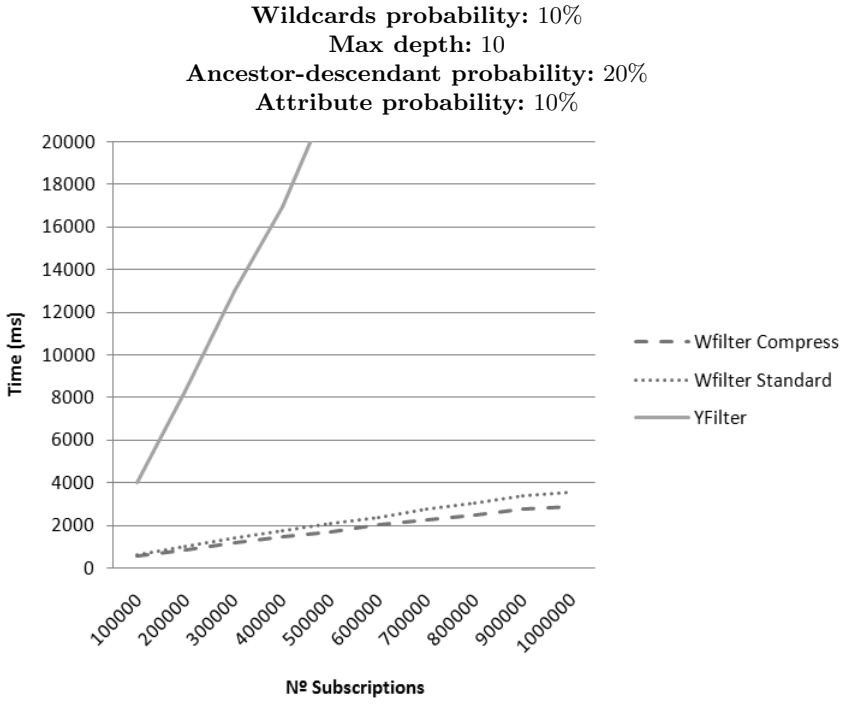

Fig. 3. Performance on increasing amount of subscriptions with attributes

As we can see, just by having a small percentage of attributes in the subscriptions our algorithm severely outperforms YFilter, this is not only because we include the attribute expressions as regular predicates and so make them part of the matching. It happens because we gain significant improvements in selectivity when using attribute predicates as *access predicate* for our solution. While, YFilter requires post-processing and a more complex structural matching in order to store the paths visited in order to verify the attribute expressions later. In this case, WFilter performs over 10 times faster than YFilter.

6 Conclusion

The purpose of this work was to develop an XML filtering algorithm suited for an XML/XPath publish/subscribe environment based on the efficient techniques employed by Le Subscribe for value-based publish/subscribe systems. The main goal was to perform efficient matching between a very large amount of subscriptions and a high rate of incoming events. Moreover, since an XML/XPath publish/subscribe system provides a very high expressiveness in both the events and in the subscriptions, we considered that in any context where it would be employed, that expressiveness would play a significant role. Due to to that, our goal was not only to develop an efficient solution but also one that could handle subscriptions encompassing a significant amount of the XPath language and still perform well.

In this work we proposed the XML Filtering algorithm WFilter, that is able to perform matching on a structural level as well as on attribute expressions. It does so, by representing both types of subscription steps as an entity of the same kind, a textitpredicate and to a certain level, fully exploring the commonality between those predicates. The predicates are only stored once and are represented by a unique ID. The matching process was basically divided into a two-step process where the first step is very simple and its result simplifies the second step. In the first step each unique predicate is only verified once and the result of this first phase is each predicate containing information indicating if it was matched and at which depth. The second phase verifies which subscriptions are matched by determining if the matched predicates (matched in the first phase) are in the order expressed in the subscription.

Because subscriptions are stored in different clusters based on a common predicate between them (the access predicate). The second phase in the matching verification is performed only on a subset of the subscriptions in the system. As only the clusters whose access predicate is verified need to be further analyzed.

To properly cope with the indexes and techniques of Le Subscribe and with the predicate representation of subscription steps our algorithm simplified the events by parsing each of the paths on a XML document independently, basically, breaking an external *event* into several smaller events called internal events.

In the end, we've achieved the goals expected with this work as WFilter supports both structural expression steps and attribute expression steps and severely outperforms YFilter, a state of the art algorithm, when dealing with subscriptions that contain both attribute expressions and structural expressions in any distribution. Even, when handling with subscriptions regarding only the structural subset of the XPath language, which is the main focus of YFilter, WFilter shown to perform significantly better.

A further development of WFilter is to find an efficient way to fit nested path matching into the current matching paradigm. Currently, it would be quite simple to handle nested path expressions by expanding those expressions into two different expressions and indexing them on an additional index for post-processing. However, not only, this is a standard solution that is already employed in most XML Filtering algorithms, but this would also go against WFilter's goal of including all of matching steps into the same process for the sake of efficiency. WFilter, is implemented in a way that is suitable for parallelization and as such is it recommended.

References

1. Hanani, U. and Shapira, B. and Shoval, P. Information Filtering: Overview of Issues, Research and Systems. Journal User Modeling and User-Adapted Interaction (2001)
2. Clark, James and DeRose, Steve, XML Path Language (Xpath) Version 1.0. W3C Recommendation
3. Berglund, A. and Boag, S. and Chamberlin, D. and Fernandez, Mary F. and Kay, Mand R, Jonathan and Siméon, J..: XML Path Language (XPath) 2.0. W3C Working Draft (2003)
4. Pereira, J..: LeSubscribe. Algorithmes de filtrage efficace pour les systèmes de diffusion dinformation à base de notifications., Université de Versailles Saint-Quentin-en-Yvelines (2002)
5. Altinel, M. and J. Franklin, M..: Grid Efficient Filtering of XML Documents for Selective Dissemination of Information In: 26th VLDB Conference, New York (2000)

6. Yanlei D. and Fischer, P. and Franklin, M.J. and To, R.: FiST: scalable XML document filtering by sequencing twig patterns. Proceedings, 18th International Conference on Data Engineering (2002)
7. Kwon, J. and Rao, P. and Moon, B. and Lee, S.: TYFilter: efficient and scalable filtering of XML documents . Proceedings of the 31st international conference on Very large data bases (2005)
8. Candan, K. Selçuk and Hsiung, W. and Chen, S. and Tatemura, J. and Agrawal, D.: AFilter: adaptable XML filtering with prefix-caching suffix-clustering. Proceedings of the 32nd international conference on Very large data bases (2006)
9. Kwon, J. and R., P. and M., Bongki and Lee, S.: Value-based predicate filtering of XML documents. Data Knowl. Eng. (2008)
10. Antonellis, P. and Makris, C.: Efficient and Scalable Sequence-Based XML Filtering. J. Web Eng. Technol. (2008)
11. Mariam S. and Vassilis J. T.: XFIS an XML filtering system based on string representation and matching. WebDB (2009)
12. Hou, Shuang and Jacobsen, H.-A. .: Predicate-based Filtering of XPath Expressions. ICDE (2006)