BFilter: Efficient XML Message Filtering and Matching in Publish/Subscribe Systems

Liang Dai[1], Chung-Horng Lung[2*], Shikharesh Majumdar[2]

[1] School of Computer Science, Carleton University, Ottawa, Ontario, Canada.

[2] Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada.

* Corresponding author. Tel.: +1-613-520-2600; email: chlung@sce.carleton.ca

Manuscript submitted October 1, 2015; accepted January 20, 2016.
doi: 10.17706/jsw.11.4.376-402

Abstract: XML message filtering and matching are important operations for the application layer XML message multicast. As a publish/subscribe system and a specific case of content-based multicast in the application layer, XML message multicast depends highly on the data filtering and matching processes. As the XML applications emerge, efficient XML message filtering and matching become more desirable. Many XML filtering techniques have been proposed in the literature. Most of those techniques do not address complex queries with predicates, twig patterns or branches; some require post-processing or a special coding scheme, which is either time consuming or becomes difficult for management for dynamic changes of user queries. This paper addresses the existing gap in the literature and proposes a new technique called BFilter which performs the XML message filtering and matching operation by leveraging branch points in both the XML publication document and user requests or queries. BFilter evaluates user queries that use backward matching branch points to delay further matching processes until branch points match in the XML publication document and the user query. Using the backward branch point matching technique, XML message filtering can be performed more efficiently as the probability of mismatching in the matching process is reduced. A number of experiments have been conducted and the results demonstrate that for complex queries, BFilter has a better performance than the well-known YFilter.

Key words: Publish/subscribe systems, XML, XML filtering and matching, Bfilter.

1. Introduction

Filtering and matching of published documents or messages, and application layer multicasting are two critical operations in Web services-based publish/subscribe (pub/sub) systems. Efficient techniques for performing these operations are crucial for achieving high system performance, especially as the volume, variety, and velocity of data has been increasing quickly. Filtering and matching operations are needed to identify registered or interested subscribers for the published messages, whereas application layer multicast is needed for data dissemination to subscribers. To effectively support pub/sub systems, the construction and maintenance of the overlay structure are the main issues in application layer multicasting.

Multicasting can be realized at different layers of the Internet architecture. In network layer multicast techniques, packets are replicated and forwarded by network routers. On the other hand, application layer multicast uses an overlay on top of the physical network for communication between hosts. Application layer multicast has some advantages over network layer multicast. First, it does not need to make changes at the router level, which makes implementation more flexible. Second, it can arbitrarily group receivers

from any location in the network. Network layer multicast, however, uses IP addresses to restrict receivers to certain subnets. This means that the receivers within a subnet are usually grouped geographically, which is not appropriate in the case of pub/sub systems, in which the receivers (subscribers) with a particular interest can be located anywhere in the network.

Application layer multicast has a few disadvantages. First, the packets are sent along the application overlay layer from a source to a destination, instead of following the shortest path at the network layer. Thus, the path traversed by a packet may be longer in comparison to network layer multicast. Second, duplicated packets may occur at some links. Thus, the main challenge in application layer multicast is for end systems to construct effective overlay structures.

In pub/sub systems, a subscriber registers a subscription to the pub/sub service and receives published messages that match the subscription. Intuitively, the sources (publishers) can allow their subscribers to retain whatever they want, and send all the data to all subscribers. This approach is definitely not efficient because there are too many duplicated data packets that reduce system throughput and waste network resources as well as increase the processing overhead for the intermediate nodes.

Many research efforts have discussed multicast in the context of pub/sub systems, e.g., [1]–[17]. Generally speaking, there are two ways to carry out multicast in this area. The first is to identify the subscriber by using the subscription information, and then send appropriate data to these subscribers. Data matching can be performed either at the source or at some centralized brokers. The second method is to perform data matching on the fly. In this way, the source simply pushes the data into the network that has a multicast tree composed of routers or called brokers in this context. The application-layer routers or brokers on the tree have a filtering mechanism to dispatch proper subsets of the messages received to their children. The children in turn perform data matching and dispatching and forward the matched data to their children. This continues until the filtered data reaches the subscribers.

The first aforementioned approach may use keyword-based multicast or distributed hash table-based multicast. Keyword-based multicast groups subscribers using the keywords in their subscriptions [2], [7] [18]-[22]. Distributed hash table-based multicast uses hash functions to assign keys to subscribers by using their subscriptions [23]. These methods are efficient in terms of delivery speed. However, the keyword-based approach is less expressive because the subscriptions contain only keywords. The distributed hash table approach is not content-aware. In both of these methods, data matching is based on the key or keywords but not the content. The second approach delivers data according to the content. The subscription description is used to perform the matching. The subscription can be presented either in an n-tuple containing n information spaces, or in XPath expressions [8], [24]–[26]. An XPath expression is used for addressing portions of a XML file. XPath is more expressive than n-tuple [24]–[26] and has been used in this research.

A XML file is a tree-based structure for describing information. The data content is available between a start tag and an end tag. The pair of tags not only scopes the data it contains, but also describes the data, possibly with some constraints on the tags. One XML document has one root tag pair. The root tag pair can have child tag pairs and the children can have their own child tag pairs, and so on. This structure forms a tree with one single root. As an XML file is semi-structured, it naturally applies filters in the hierarchy to perform data matching and delivery. XML-based multicast can properly match and deliver messages to subscribers. However, because it is more difficult to index and identify the elements in the XML file, compared to the content-based message format, which can be considered to be an n-dimensional array containing keywords, the filtering process in each node is time consuming. Hence, the performance of XML-based multicast depends heavily on the approach used to process the XML message.

A preliminary report described the basic idea of a novel XML message filtering algorithm—BFilter [27].

(B represents branch points.) This paper examines various filtering approaches and discusses BFilter in details. BFilter realizes the tree structure in both XML documents and user requests with nested paths. A user request or query with nested paths is called complex query which is resource intensive for the filtering and matching operation. To mitigate the performance challenge for complex queries, BFilter conducts the XML message filtering and matching process by identifying branch points in both XML documents and user requests using a bottom-up approach. The main advantage of such an approach is that the filtering evaluation of user requests uses backward matching branch points to delay further matching, so that the probability of a mismatch is reduced and XML message filtering can be performed more efficiently.

The measurement results from the current implementation show that BFilter performs significantly better than YFilter while handling complex queries. The performance improvement for BFilter obtained from a number of experiments can be as high as about 70% for some scenarios using complex queries. This is observed when the values of the five parameters (Query Depth, Number of branches, Probability of wildcard, Probability of ancestor/descendant relation and Number of predicates) used in the experiments are varied. In summary, the main contributions of the paper include:

- A novel approach to handle complex queries without decomposition, which allows the matching of complex queries to be processed more efficiently. See Section 2 for more detailed description and comparison with existing approaches.
- Emphasis on branch point matching during the bottom-up process to reduce the possibility of mismatch for performance improvement.

The rest of the paper is organized as follows. Section 2 presents the background and related work. Section 3 discusses the backward matching branch point algorithm. Section 4 demonstrates the some experimental results. Finally, Section 5 concludes the paper.

2. Background and Related Work

There are two important operations performed in a XML pub/sub system: XML message filtering and matching, and XML message multicast. This paper focuses on techniques for XML message filtering and matching. A representative set of related work that focuses on XML filtering and matching is presented in this section.

XFilter [24] is based on deterministic finite automata (DFA), which uses linked lists to store user requests and handle each request individually. It is capable of handling XPath relationship notations, such as ancestor/descendant (represented by a '//' in XPath) as well as a wildcard '*'. Because it stores requests separately, the same segments of a different request cannot share storage space and have to be matched individually. XQRL [9] and XScan [11] are also DFA-based approaches.

DFA has a potential state explosion problem due to its exponential growth in the number of XPath expressions. One approach to overcome this issue is to use lazy DFA [10]. With the lazy DFA technique, states and transitions are calculated at run time, not at the compile time. In other words, there is no need to calculate all the states in the beginning, hence, the number of states can be reduced. However, for complicated XML data, the number of states still grows exponentially [10]. To reduce the number of states, Zhao and Xia [38] proposed an approach that uses stream index and hash table to evaluate and skip invalid elements in XPath expressions for performance improvement.

Another solution to mitigate the state explosion problem is to use nondeterministic finite automata (NFA). YFilter [29] uses NFA to emphasize prefix sharing for performance improvement. However, the ancestor/descendant relationship in complex queries introduces more matching states, which may also result in the number of active states increasing exponentially [30]. Post-processing is another feature of YFilter. To deal with queries with nested paths (complex queries), YFilter decomposes a query into simple

queries and matches them separately. For example, a complex query /a/*[c]/*/c will be split into /a/*/c (Q1) and /a/*/*/c (Q2). After NFA execution has been completed for the entire document, post-processing starts to verify if Q1 and Q2 are both matched and whether the first "*" is matched in the same place in the document.

In contrast to YFilter, AFilter emphasizes suffix commonality to reduce mismatches [31]. The objective of AFilter is to overcome the disadvantage of YFilter by constructing user queries or requests as a directed graph (AxisView). The transition between two states is conditional for each request. It depends on the type of relationship between the corresponding element names for each state. In this way, the ancestor/descendant relationship can be treated as a self transition. The outgoing state is simply a waiting state. Thus the increment in the number of active states is not significant.

AFilter uses a triggering mechanism to delay the matching process until a trigger condition is met. Each node in the directed graph corresponds to a label and each edge corresponds to a set of axis tests. Each edge is annotated with a set of axis assertions for verification. Unlike YFilter, which uses a stack to store matched active states, AFilter uses a set of named stacks to store matched states. The states are called objects because they also maintain pointers that allow tracing of previous matching states. Therefore, the state stacks actually store the matching paths. AFilter implements these data structures to backtrack the matching path whenever an end of request is triggered during the process of matching.

Optionally, AFilter uses PRLabel-tree to leverage *prefix* sharing by caching path expressions based on the commonalities in their prefixes, and uses SFLabel-tree to enhance *suffix* sharing by caching path expressions based on their overlapping suffixes. AFilter does not explicitly deal with requests with nested paths, nor does it address attribute-based expressions and predicates.

The hybrid algorithm [32] enhances performance from the viewpoint of engineering. It is intended to speed up the filtering process by taking advantage of both Yfilter and Afilter. The hybrid algorithm pre-processes the user requests and separates them into two groups. One group contains the so-called simple requests and the other contains the complex requests. A simple request here means that the request has no wildcard or ancestor/descendant relationship; otherwise, it is a complex request. The complex requests are sent to Afilter. Neither Afilter nor the hybrid algorithm explicitly deals with nested paths. The performance of the hybrid algorithm depends on the percentage of simple requests in the total user requests. However, the cost of switching the two underlying structures is unknown. Moreover, the cost of expanding its capability to handle a nested path is not predictable. This is because the decomposed requests may be either simple or complex.

Chen *et al*. proposed a novel XML message filtering algorithm called GFilter [33]. In Gfilter, post-processing of complex queries is improved using Tree-of-Path (TOP) coding scheme [34] and the query is represented by the Generalized-Tree-Pattern (GTP) introduced by Chen el al. [35]. GFilter focuses on optimizing the path matching performance via a bottom-up approach, benefiting from the heuristics that the probability of mismatching at the leaf is higher than at higher levels, which results in faster filtering decision. However, the decomposition and post-processing is needed for handling complex queries.

Sun *et al.* [36] proposed HFilter, a hybrid stream filtering method, that combines the lazy DFA and NFA techniques. The motivation of HFilter is based on the observations that NFA based approaches are more space efficient than lazy DFA based techniques, but lazy DFA based techniques could be more time efficient than NFA based methods. The main target of HFilter is deep and recursive XML data with low memory limitation. However, the large space consumption problem still exists, resulting in memory overflow and degradation of efficiency [35].

SFilter [1] made use of query guide and simple integer stacks (as opposed to states stack used in YFilter) to achieve better performance in comparison to YFilter. A query guide represents all the path trees for the

path queries. However, SFilter focuses on linear paths with child and descendant axes against XML stream; predicates, twig queries (queries with twig patterns or branches), or nested paths are not considered in SFilter, which are recognized important features for XML documents.

WFilter [30] utilizes a coding scheme that converts each query node into to a special internal representation. The approach can efficiently store and identify query nodes and their positions for the matching process. A comparison with YFilter showed significant performance gains. However, similar to SFilter, the approach only focuses on linear parent-child and ancestor-descendant relationships. Nest paths and queries with branches are beyond the scope of WFilter.

iFiST [13], extended from FiST [12], encodes the XML document and XPath queries into Prüfer sequences. Similar to GFilter, iFiST also handles the matching via the bottom-up approach by using a subsequence matching algorithm for twig patterns. A subsequence is a linear pre-order path from the root to a leaf node. After all the subsequences of a twig pattern are matched, the second phase of iFiST is to perform post-processing to discard false matches by verifying the correctness at the branch nodes in the twig patterns of the document and the queries, as filtering subsequences alone may lead to false positives [13]. Adopting the Prüfer coding scheme is time efficient. However, a special coding scheme becomes difficult to manage for dynamic changes of queries.

PFilter [15] is also a sequence-based filtering technique. It encodes XPath expressions into value-based sequences based on pre-order traversing of the query tree (from the root to leaf nodes). The objective is to identify related profiles for structure and content matching. The generated sequences are grouped into hash-based indices that can be processed concurrently. The results for a few experiments showed that PFilter could be up to 18% faster than YFilter.

QFilter [37] was proposed to improve secure database access control. QFilter ensures that only authorized users have access to the authorized XML data. QFilter captures access control rules as NFA and performs pre-processing for each query expressed in XPath to determine if the query can be accepted or needs to be rewritten. The results show that QFilter has better performance than approach that has no access control, uses YFilter for post-processing, or handles static analysis.

As the volume, variety, and velocity of data have been increasing significantly, an efficient XML filtering and matching technique has to consider all of these challenges. In summary, the main approaches reported in the literature use DFA, NFA, or a hybrid approach (i.e., lazy DFA + NFA, or NFA + a special coding scheme). DFA based approaches, including the hybrid method using lazy DFA, have the space problem which become even more challenging as the volume of data increases dramatically. The performance of NFA based methods, e.g., YFilter, suffer due to the deep and recursive structure of complex queries. Some approaches improve the performance of YFilter, but they either do not address complex queries (e.g., predicates or queries with twig patterns or branches) or require a special coding scheme. As the variety of data also changes quickly, it becomes essential to address complex queries. On the other hand, using a special coding scheme has yet to demonstrate the effectiveness of high velocity of data changes, as recalculations need to be conducted with the coding scheme for changes. For NFA based approaches, either top-down, e.g., YFilter, or bottom-up techniques, e.g., GFilter and iFiST, have been proposed. As demonstrated in GFilter and iFiST, using a bottom-up technique is more efficient than a top-down method.

This paper presents a new filtering technique called BFilter that mitigates some of the limitations of the existing work described in this section. The primary objective of BFilter is to facilitate matching of complex queries (with twig patterns or nested paths). Similar to iFiST, BFilter adopts the holistic matching principle, i.e., a twig pattern of a complex document is not divided into multiple linear paths and processed for matching separately. But iFiST requires the post-processing step for branch node verification. PFilter removes the post-processing operation. Moreover, iFiST and PFilter either use a coding scheme or

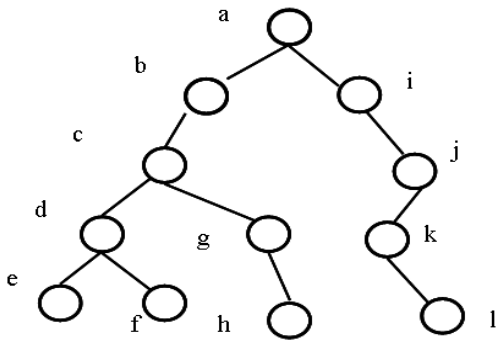
hash-based index for sequences of paths from the root to leaf nodes. While the performance of iFiST and PFilter may be improved using those coding schemes, the cost of keeping track of dynamic changes of queries could be high. BFilter also eliminates the post-processing phase of YFilter. BFilter differs from PFilter that adopts a top-down approach for matching evaluation, whereas BFilter, similar to GFilter and iFiST, uses a bottom-up approach for the matching evaluation. Furthermore, BFilter starts the matching evaluation from the lowest-level branch nodes instead of leaf nodes used in GFilter and iFiST, and the approach does not decompose a complex query into simple queries for the matching process. As a result, BFilter can efficiently perform the filtering and matching task. Section 3 explains the rationale behind the filtering and matching process, and discusses BFilter in details.

3. Bfilter — Backward Matching Using Branch Point

BFilter is a novel XML message filtering algorithm for XML message filtering and matching. BFilter effectively deals with user requests with complex queries by leveraging branch points in both XML document and user requests. It uses backward matching of branch points to evaluate user requests. A matching process is delayed until branch points in both the XML document and the user request match. Thus, XML message filtering can be performed more efficiently as the probability of mismatching is reduced. BFilter not only matches queries backwards, but also matches branch points backwards. The match of branch points is preconditioned for further matching of a complex query. Since BFilter treats a complex query as a whole, no post-processing is needed for complex queries.

3.1. Technique Overview

BFilter treats a complex request as a unit without having to decompose it to simple queries. Similar to YFilter, BFilter utilizes the tree structure for XML documents and user queries. In BFilter, a tree is composed of three parts, as illustrated in Fig. 1: branch points, including the root and nodes that have more than one child node; branches connecting two branch points; and branches that have only one end attached at a branch point while the other end is free. A branch that connects two branch points is known as a *Transit Branch*; other branches are called *Tangling Branches*.



Branch Points: a, c and d
Transit Branches: a/b/c and c/d
Tangling Branches: d/e, d/f, c/g/h and a/i/j/k/l

Fig. 1. A tree structure of a XML document or user query [27].

In Fig. 1, there are two transit branches and four tangling branches. The two transit branches are *a/b/c* and *c/d*. The transit branch *a/b/c* connects branch points *a* and *c*. The transit branch *c/d* connects branch points c and d. The four tangling branches are *d/e*, *d/f*, *c/g/h* and *a/i/j/k/l*. If a XML query matches a document, the branch points in the request must match the corresponding branch points in the XML

document.

BFilter matches queries from the end to the front of a request. The matching process starts only when both a document branch point and a request branch point have been matched. Only when the branch point is matched will its tangling branches and the transit branches (if they exist) are checked. In other words, the matching of branches is delayed until the branch point they attach to is matched.

The following description explains why BFilter is more efficient using branch point matching operations than other filtering techniques. Given a user query Q with a nested path and a XML publication document, let the probability of matching of Q be $P(Q)$. Because YFilter matches a query from the front to the end, we have:

$$P(Q) = P(\text{front}) \times P(\text{rest} \mid \text{front}),$$

where P(front) is the probability of matching of the front part of Q and P(rest | front) is the probability of matching of the rest of Q when the front is matched. Similarly, AFilter matches a query from the end to the front:

$$P(Q) = P(\text{end}) \times P(\text{rest} \mid \text{end}),$$

where P(end) is the probability of matching of the end part of Q, and P(rest | end) is the probability of matching of the rest of Q when the end is matched.

Because BFilter matches a query from the last branch point to the front of the query, we have

$$P(Q) = P(\text{last branch}) \times P(\text{rest} \mid \text{last branch}),$$

where P (last branch) is the probability of matching of the last branch point of Q and P(rest | last branch) is the probability of matching of the rest of Q when the last branch is matched.

In real XML messages, a tag name has a higher probability of appearance at a higher level than at a lower level in a query. In other words, the probability of a match of the front part of a query is higher than that of its end part, which indicates that a match for the front part of a query (which is more generic) may not lead to a successful match for the end part (typically more specific). This observation motivates a backward matching (bottom-up) approach used in XML filters, including AFilter, GFilter as well as BFilter. It implies that, for a particular query, if the match of its last element(s) is successful during backward matching, the probability of a match of its remaining part is higher than the case in which the match of its front element(s) is successful during forward matching. This heuristic can be explained through the following example, in which a user is requesting information about the Systems and Computer Engineering department at Carleton University in the domain of Canadian secondary schools: Canada/University/Carleton/Systems and Computer Engineering. A simplified example of a XML document of Canadian Post-Secondary Schools is shown below:

```
<Canada>
    <University>
        <Carleton>
            < Systems and Computer Engineering/>
        </Carleton>
        <Toronto>
            <Electrical and Computer Engineering/>
        </Toronto>
    </University>
```

```
<College>
    <Algonquin>
        <ESL Language School/>
    </Algonquin>
</College>
</Canada>
```

Because the top level tag of the message "Canada" is the most generic element and will appear in all messages, the probability of a match of the first element of the request is 100%. The second level tag "University" is likely to appear in about half of the messages in the domain because, unlike in the example, where the message includes both "University" and "College" at the second level, some messages may contain only "University" or "College". So the probability of a match of the second element of the request is about 50%. As the third tag of the message, "Carleton" will appear in fewer messages than the tag "University". Thus the probability of a match of the third element of the request will be lower than the element "University". Similarly, the probability of a match of the fourth element "Systems and Computer Engineering" will be lower than that of the element "Carleton". Thus, the probability of a match of an element at the higher level is usually higher than that at the lower level. So *P* (front) is greater than *P* (end). Furthermore,

$$P(\text{last branch}) = P(\text{branch point matched and it is a branch point in XML document}) \quad (1)$$

$$= P(\text{branch point matched}) \times P(\text{it is a branch point in XML document}) \quad (2)$$

Note that the two events (branch point matched) and (it is a branch point in XML document) are independent. Further, *P* (branch point matched) is approximately the same as *P* (end) because the branch point is the last one which is close to the end element of the request. On the other hand, the number of branch points in any XML document is at most half of the total number of tags, so *P* (it is a branch point in XML document) is at most 0.5. Thus *P* (last branch) will be less than *P* (end). Therefore,

$$P(\text{rest} \mid \text{last branch}) > P(\text{rest} \mid \text{end}) > P(\text{rest} \mid \text{front}) \quad (3)$$

The three types of matching approaches, branch point backward matching (BFilter), backward matching (AFilter and GFilter), and forward matching (YFilter), start matching from the last branch point, the last element, and the first element of a query, respectively. After the first matching step succeeds, the probabilities of matching for the three approaches correspond to $P(\text{rest} \mid \text{last branch})$, $P(\text{rest} \mid \text{end})$ and $P(\text{rest} \mid \text{front})$, respectively. Mismatching is a scenario where the matching fails after the first step(s) succeeds. Hence, the probabilities of mismatching for the three types of approaches–branch point backward matching, backward matching, and forward matching–are given by $1 - P(\text{rest} \mid \text{last branch})$, $1 - P(\text{rest} \mid \text{end})$ and $1 - P(\text{rest} \mid \text{front})$, respectively. From equation (3), we can see that the probability of *mismatching* of branch point backward matching, e.g., $1 - P(\text{rest} \mid \text{last branch})$, proposed in BFilter is less than that in backward matching, e.g., $1 - P(\text{rest} \mid \text{end})$, adopted by AFilter and GFilter), and the probability of mismatching of backward matching in turn is less than that in forward matching, e.g., $1 - P(\text{rest} \mid \text{front})$, used in YFilter. In other words, the probability that mismatching that will occur in BFilter is expected to be the lowest of the three.

Mismatching in the XML message matching process causes the filter to spend time on evaluating queries that will ultimately fail. The lower the probability of mismatching, the greater the likelihood that processing of unmatched queries will be finished earlier in the matching process. Reducing the mismatching

probability is supposed to increase efficiency of filtering.

3.2. Query Representations

In BFilter, a complex query is represented with sub-queries that are separated by branch points. Fig. 2 illustrates an example with three queries, Q1, Q2, and Q3, in BFilter. Each query is composed of one of more sub-queries. Each sub-query starts at a branch point and has one or more paths. A sub-query is named as Q{*m* , *n* }, where m is the index of the complex query that the sub-query belongs to, and n is the index of the sub-query.

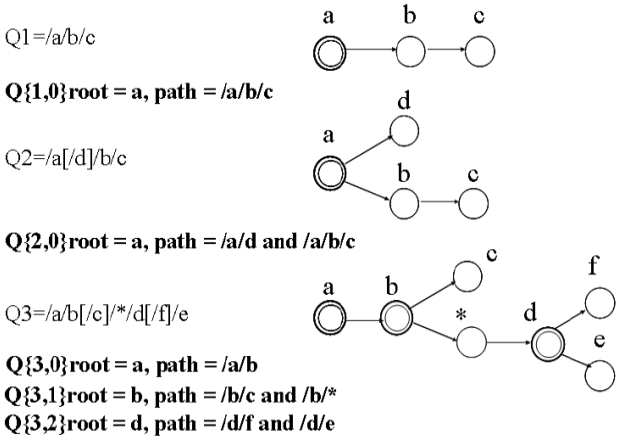


Fig. 2. Example query [27].

For instance, in Fig. 2, Q1 is a simple query; its root is element *a* which is also a branch point by definition, see Section 3.1, and is presented with a double circle. Q1 has a single path */a/b/c*. Q2 is a complex query that has one branch point (element *a* which is the root) and two paths */a/d* and */a/b/c*. Q3 is also a complex query with three branch points, *a*, *b*, and *d*. Q3 is represented by three sub-queries, Q{3,0}, Q{3.1} and Q{3,2}.

A sub-query has two pointers, one pointing to its parent and another to its child. If a sub-query is the first one in a complex query (with index zero), its parent is null; if a sub-query is the last one, its child is null.

3.3. Architecture of BFilter

Fig. 3 shows the high-level architecture of BFilter. When the user queries are read in, the *XPath Parser* is responsible for parsing them into query objects and sending these queries to the *Query Index Tree*. *Query Index Tree* stores these queries in a data structure that uses a hash table-based approach to implement NFA.

The *SAX XML Parser* is event-based XML parser [31] and a component reused from YFilter. The *XPath Parser* is derived from YFilter. In YFilter, a complex query is decomposed into simple queries. The *XPath Parser* reconstructs the decomposed queries into one unit so that the *Query Index Tree* can index them as a whole. The *Query Index Tree* is derived from YFilter's Query Index by adding additional functionalities to index complex queries and store branch point information.

When parsing a XML document, the "start document" and "end document" events are generated at the beginning and the end of the document respectively. A "start element" event is generated when a start tag is read. This event contains the tag's name and attributes. An "end element" event is generated when an end tag is read. This event is related to the "start element" event generated by the start tag corresponding to this end tag. It contains the name of the corresponding start tag and closes the segment marked by the start and end tag pair in the document. A "characters" event is generated when content between a pair of tags is read. This event carries the content as a string.

BFilter leverages branch points in both the XML document and the user request or query. It uses

backward matching branch points to evaluate user requests. The XML document *Branch Point Detector* is responsible for reporting a branch point when a document is read, by using the algorithm described in the next section. When a branch point occurs in a document, it is pushed into the *Document Branch Stack*. The *Queries Branch Points Stack* holds the branch points of queries in the form of a *Query Index Tree*. The *Query Index Tree* has a *Runtime Stack* to store the states of NFA. During the matching process, the *Query Index Tree* will push the current active states for all queries onto the *Run Time Stack*. This operation is the same as performed in the YFilter. The difference is that the *Query Index Tree* also stores the branch point information of all queries in the hash table-based structure, so the queries' branch points can be identified from the current active states stored in the hash table-based structure. The branch points found will be pushed onto the *Queries Branch Points Stack*.

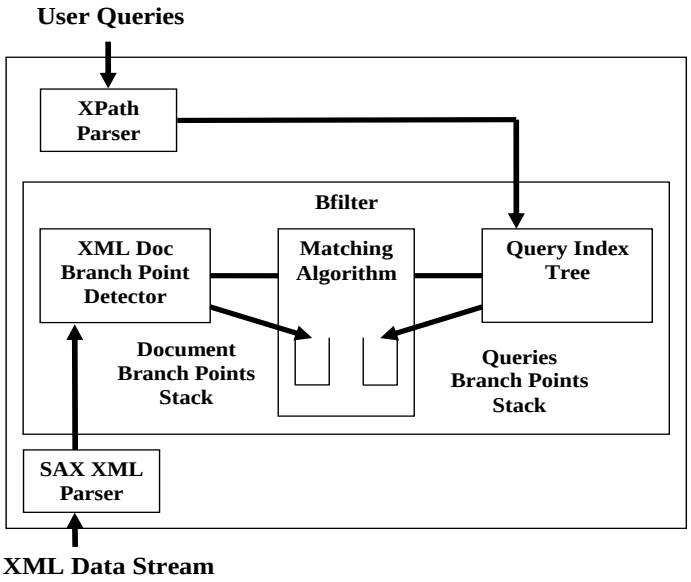


Fig. 3. Architecture of BFilter.

The main functionalities of BFilter that differ from YFilter are listed and described below.

3.3.1. Handle the stacks

As each tag is read in, it is pushed onto (start tag) or popped from (end tag) the *run Time Branch Stack*. The elements in the *run Time Branch Stack* are used to find branches of the document and match the tangling branches.

As each branch point is detected in either the document or the requests/queries index, it is pushed onto the corresponding *Document Branch Stack* or *Queries Branch Points Stack*. When an element is popped out from *the run Time Branch Stack* (as the end tag of the element is read in), if a branch point that is associated with this element exists in the two stacks, the branch point is popped out from them, and the matching is performed for this branch point.

3.3.2. Detect branch points in document

By using the stacks, the identification of branch points of queries in *Query Index Tree* is processed by using preorder traversal in the document tree (when a start tag is read); the identification of branch points in a XML document is performed during an inorder traversal of the document tree; the matching of branch points is processed during a postorder traversal of the document tree (when end tags are read).

If a sequence of end tags follows a start tag that is at the same level as the previous one, the current top element on the *RunTimeBranch Stack* is a branch point. In Fig. 4, the sample document stream is **a/b/c/d/d/c/e/e/b/f/g/g/f/a**. Branch points are represented with double circles. A tag name with an

underscore means that the tag is an end tag. When a start tag is read in, a node representing this tag is pushed onto the runtime stack. When an end tag is read, the corresponding node is popped out from the runtime stack. At the bottom of the figure, it shows the status changes of the runtime stack from left to right. When the first tag, *a*, is read in, node *a* is pushed onto the runtime stack. Now the runtime stack has one node – *a*. When tag *b* is read, node *b* is pushed onto the top of the stack. After tag *c* and *d* are read in and node *c* and *d* are pushed onto the stack, the end tag of *d* is read in, so the *d* node is popped out from the stack. Then node *c* is popped out when end tag *c* is read in. When start tag *e* is read in, the branch point *b* is detected because *e* immediately follows a pop-out of *c*. In the same way, the root *a* is detected when start tag *f* is read in after node *b* is popped out from the stack.

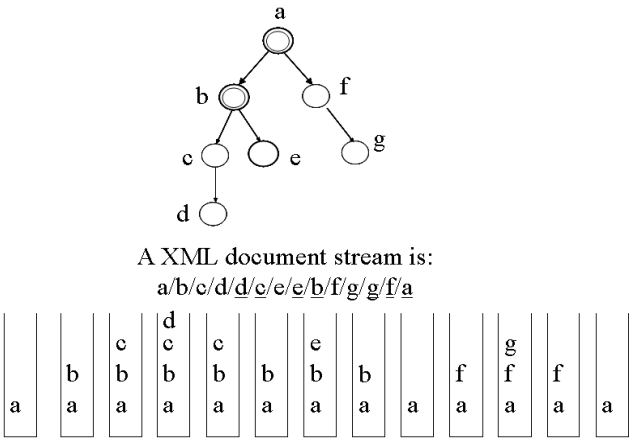


Fig. 4. Branch point identification – An example.

3.3.3. Process branch point matching

BFilter treats a complex query as a whole during the matching process. In BFilter, a complex query is represented by using sub-queries that are separated by branch points of the complex query. Each sub-query is rooted at a branch point and has one or more paths. A sub-query has only one branch point which is its root. So a complex query is represented as a linked list of sub-queries in which the first sub-query rooted at the root of the complex query is at the head of the list. For instance, if a complex query, Q, has three branch points, namely the root, branch point 1 and branch point 2, it will be represented by sub-query 0, sub-query 1 and sub-query 2 rooted at the root, branch point 1 and branch point 2, respectively. Notice that a simple query is represented as one sub-query that has only one path. In BFilter, when we say a query, it usually indicates a sub-query not a complex query.

When a branch point is popped out from the *Document Branches Stack* and the *QueryBranches Stack*, (see Fig. 3), if the branch point exists in both stacks, it means that this branch point is a match between the document and the queries that have this branch point. Therefore, the matching operation is performed for these queries. Otherwise, there are two possibilities: the branch point exists only in the *Document Branches Stack* or in the *Query Branches* Stack, but not both. In the first scenario, the element popped out is not a branch point in the queries; therefore nothing needs to be done. In the second case, the queries that have this branch point as root are simply marked as unmatched and no further matching is needed. When processing a match for a query that has this matched branch point, if its descendants are not matched, the query is marked as unmatched. Otherwise, the matching for all its branches will be performed.

As discussed earlier, BFilter processes the matching operation for a query only when its branch point finds a match in the document. The order of branch point matching is conducted from the lower level to the higher level, but the determination of a match is always made at the higher level. If the result is unmatched, all the descendants of the query are marked as unmatched and no further matching operation is needed.

3.4. Algorithm

BFilter consists of two main procedures. The first procedure, as depicted in Fig. 5, is to handle the start tag event during the filtering process. The second procedure, as presented in Fig. 6, is used to identify matching branches when a matched branch point is found. Fig, 7 depicts the filtering algorithm [27]. The two stacks, *documentBranches* and *queryBranches*, are used to hold the branch points of documents and queries, respectively. The other stack, called *runTimeBranch*, is used to hold the current elements' branch of the document.

```
Procedure: HandlingStartTag(Document tag e)

        //get current state in gIndex for e
        Send e to gIndex for states lookup;
        For each current state c
                If c is branch point for query q in Q
                        push (e, q) onto queryBranches;
                If c is accepting state for query q in Q
                        If q has no predicates on this path
                                Make the path as matched;
                        Else
                                //delay the matching until
                                //the branch point is matched.
                                Save branch in the node;
                        End if;
                End if;
        End for;
End Procedure;
```

Fig. 5. Handling start tag [27].

```
Procedure MatchingBranches()

        For each matched query q, do
                If q's child is not matched
                        //no matching is needed
                        Set q as not matched;
                Else
                        Process matching on all branches of q;

                        If q is failed in matching
                                //reset states of all sub-queries
                                Set sub-queries as unmatched;
                        End if;
                End if;
        End for;
End Procedure;
```

Fig. 6. Matching branches [27].

```
Filtering Algorithm
Input:  QueryIndex gIndex
        Stack runTimeBranch
        Stack documentBranches
        Stack queryBranches
        Incoming element e
        List CurrentQueries Q
Init:
        gIndex and Q is populated by user requests

        runTimeBranch = Empty;
        documentBranches = Empty;
        queryBranches = Empty;

        While incoming element e is not the end of document
                If e is start element then
                        Push e onto runTimeBranch;
                        If document branch point is detected
                                Push the top of runTimeBranch onto documentBranches;
                        End if;
                        Call Procedure HandlingStartTag(e);
                Else
                        Pop e from runTimeBranch;

                        //a branch point in queries matches a branch point in document
                        If e is a branch point in queryBranches and documentBranches
                                Call Procedure MatchingBranches();
                                Remove from documentBranches and queryBranches;
                        Else if e is a branch point in queryBranches
                                //no matched branch point in document, simply set
                                //unmatched
                                For each matched query q for this branch point, do
                                        Set q and its sub-queries as unmatched;
                                End for;
                                Remove from queryBranches;
                        Else if e is a branch point in documentBranches
                                // no matched branch point in queries, do nothing
                                Remove from documentBranches;
                        End if;
                End if;
        End while;
End Algorithm;
```

Fig. 7. Filtering algorithm [27].

The first if statement shown in Fig. 7, tests if a start element or tag is read in. When a start tag is read in, the element is pushed onto the *runTimeBranch* stack. If a branch point is also detected, the top of the *runTimeBranch* stack will be copied to the *documentBranches* Stack and all the queries associated with the branch point will be pushed onto the queryBranches stack.

When the element's end tag arrives, the element is popped out from the *runTimeBranch* Stack. When an element sent to the *queryIndex* Tree reaches an accepting state for a path of a sub-query, the portion of the current elements' branch, which is from the element that triggers the root of the sub-query to the current arriving element, will be saved to the sub-query as a candidate for a match of its path.

The following concrete example demonstrates how the BFilter algorithm works. This example consists of five queries, Q1 to Q5, and it also shows the sub-queries and associated paths for each query. For instance, Q2 is composed of two sub-queries, Q{2,0} and Q{2,1} rooted at element *a* and element *b*, respectively. Fig. 8 illustrates an aggregated query tree for those five queries along with the query index and the path information. The aggregated query tree is represented as a state machine, i.e., each node in the tree represents a state and each element of a XML query is an input.

- Q1: /a/b/e
  - Q{1,0}root=a, p0=a/b/e
- Q2: /a/b[/c]/e
  - Q{2,0}root=a, p0=a/b
  - Q{2,1}root=b, p0=/b/e, p1=/b/c
- Q3: /a//c[/e]/d
  - Q{3,0}root=a, p0=/a//c
  - Q{3,1}root=c, p0=/c/d, p1=/c/e
- Q4: /a/*[/b]/c
  - Q{4,0}root=a, p0=/a/*
  - Q{4,1}root = *, p0=/*/c, p1=/*/b
- Q5: /b/f/g
  - Q{5,0}root=b, p0= b/f/g

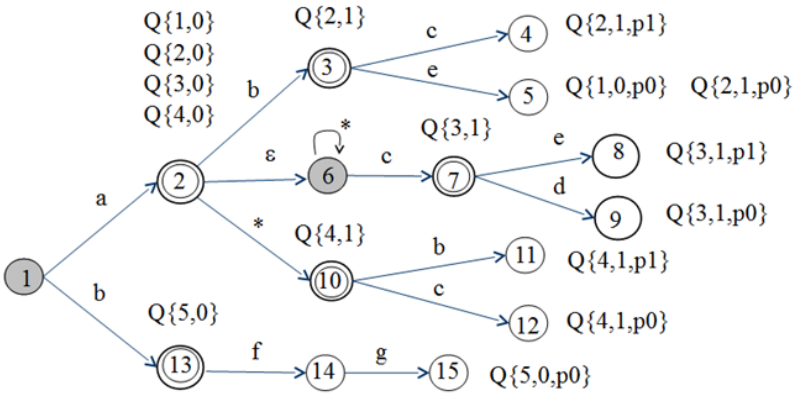


Fig. 8. An aggregated query tree and query index for Q1–Q5 (adapted from [27]).

Query Q1 is simple and is represented as only one sub-query: Q{1,0}. The indexing starts from its root element *a* which reaches state 2, as shown in Fig. 8. State 2 is then marked as a branch point, represented with a double circle, of Query 1, in particular, the root of Q{1,0}. (The root is a branch point; see the definition of branch points in Section 3.1.) Q{1,0} has only one path, e.g., p0. After the last element of the path reaches state 5, this state is marked as Q{1,0,p0} which is the accepting state for path p0 of Q{1,0}.

Query Q2 consists of two sub-queriesQ{2,0} and Q{2,1}. Q{2,1} has two branches starting from state 3. State 2 is marked as the branch point of Q{2.0} and Q{2,0} is the parent of Q{2,1}. State 4 and state 5 are marked as accepting states for the two paths of Q{2,0}, e.g., Q{2,1,p0} and Q{2,1,p1}, respectively.

Similarly, Query Q3 is composed of two sub-queries, Q{3,0} and Q{3,1}. The indexing starts from Q{3,0} which is the parent of Q{3,1}. Because Q{3,0} has no tangling branch, its transit branch is indexed right away at the beginning. The transit branch contains an ancestor/descendant relationship between elements *a* and *c*; hence state 6 with a self-loop is created for this relationship. The symbol "ε" means that the transition needs no input to enter state 6. After the transit branch is indexed, state 7, which is the state of the last element of the transit branch *c*, becomes the start state for indexing Q{3,1}. When indexing for Q{3,1} begins, the start state is marked as a branch point for Q{3,1}. Because Q{3,1} has two tangling branches, they are indexed one by one. State 8 and state 9 are then marked as accepting states for the two branches, respectively. Query Q4 is also a complex query with a wildcard * and is indexed in the same way as Query Q3. Q4 has two branch points, e.g., state 2 and state 10.

Query Q5 is a simple query, but it starts with element *b*. The purpose is to illustrate that the query tree potentially could have many branches even from the beginning. Further, the same element appears at different positions in a query, e.g., element *b* as the 2nd node in Q1 and Q2 in comparison to its 1st position in Q5, could result in very dissimilar structure.

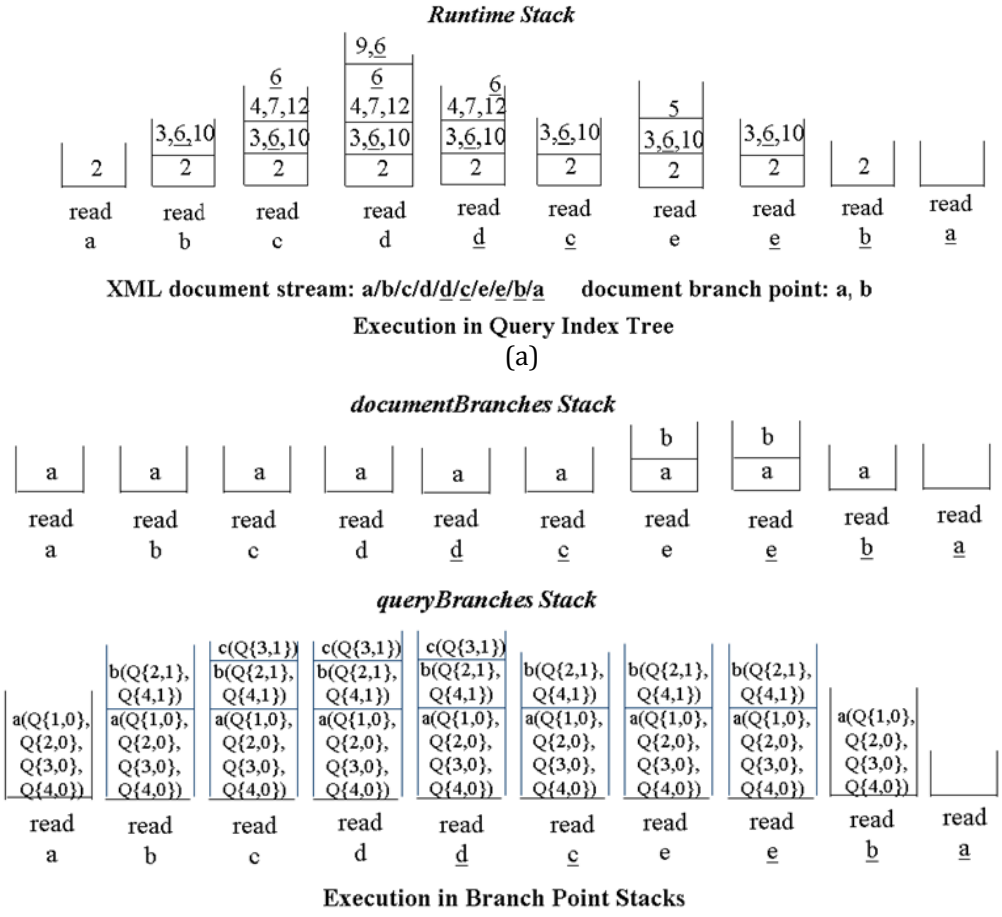


Fig. 9. Execution in Bilter: an illustration [27].

To demonstrate the stack operations for Bfilter, we assume that the incoming XML document stream is *a/b/c/d/d/c/e/e/b/a*, where the underscored letters are the end tags. Note that the branch representing Q5

in Fig. 8 is not considered in this example, since the XML document stream starts with element *a*, but Q5 starts with element *b*, which results in a mismatch. Fig. 9(a) depicts the execution in the query index tree and the changes of the status of the Runt*ime Stack* which is used to hold active states during matching process. Fig. 9(b) shows the changes of the statuses of *documentBranches* and *queryBranches*. Notice that a dummy element should be pushed onto both stacks if no branch point corresponding to a start tag is found. This is to make the branch point matching in the two stacks easy by simply looking at the top elements in both stacks. The process of pushing dummy elements onto the stacks is not shown in Fig. 9(b) for brevity. Fig. 9(a) depicts the changes of the status of the Runtime Stack. This Runtime Stack is a part of the query index tree; it is used to hold active states during the matching process.

At the beginning the *Runtime Stack* is empty. When the first element, *a*, is read in, state 2 is reached in the *queryIndex* Tree and is pushed onto the *Runtime* Stack. State 2 matches four branch points associated with element *a* of the queries previously aggregated in the tree. These four branch points are the first sub-queries of Q1, Q2, Q3 and Q4 and are represented as Q{1,0}, Q{2,0}, Q{3,0} and Q{4,0}, respectively. These branch points are wrapped as a unit which is associated with the element a so that they can be found when an end tag of element *a* is read. Because the first element is also the root of the XML document, a node associated with element *a* is pushed onto the *documentBranches* Stack.

When element *b* arrives, three states are active. First, there is a transition of *b* from state 2 to state 3, after which state 3 becomes active. Second, because state 6 is the next state of state 2 that requires no input, state 6 is also active. Finally, the transition of "*" from state 2 to state 10 matches any tag name so that state 10 is active as well. Thus, state 3, state 6 and state 10 are pushed onto the *Runtime Sta*ck based on the *queryIndex* Tree. Because states 3 and 10 are branch points of Q{2,1} and Q{4,1}, the two branch points are pushed onto the *queryBranches Stack* that are associated with element *b*. At this stage element b has not been detected; thus we do not know whether or not it is a branch point in the document. However, it will be detected before the end tag of *b* is read [27].

When element *c* is read, states 4, 7, 12 and 6 are reached from previous states 3, 10 and 6. These states are pushed onto the *Runtime Stack*. State 6 is automatically added because it needs no input. States 4 and 12 are accepting states of Q{2,1,p1} and Q{4,1,p0}, thus, the current branch in the *runTimeBranch* Stack is saved in the query object for Q{2,1} and Q{4,1} as a candidate of match for the path 1 of Q{2,1} and path 0 for Q{4,1}, respectively. Because state 7 is a branch point of Q{3,1}, Q{3,1} is pushed onto the queryBranches Stack. After element *d* is read in, states 9 and 6 are pushed onto the Runtime Stack. The current branch is saved in Q{3,1}, because state 9 is the accepting state of Q{3,1,p0}.

When the end tag of *d*, e.g., *d*, is read in the top states (9 and 6) are simply popped out from the *Runtime Stack*. When the end tag *c* arrives, top states 4,7,12 and 6 are popped out from the *Runtime Stack*. Because element *c* has a corresponding branch point in *queryBranches* Stack but not in *documentBranches* Stack, the corresponding branch point is popped out from *queryBranches* and Q{3,1} is marked as unmatched. No matching will be performed for the sub-query Q{3,1} at this point because element c is not a branch point in the document.

When element *e* is read in, state 5 is reached from state 3. The current branch is saved as a match candidate for corresponding paths Q{1,0} and Q{2,1} respectively. Because element e follows a sequence of pop operations (due to end tags), the element *b*, which is on the top of the *runTimeBranch* Stack before the element *e* is pushed, it is detected as a branch point of the document according to the branch point detection algorithm. Thus *b* is pushed onto the *documentBranches* Stack. After the end tag of e is read in, the top of the *Runtime Stack* is popped out.

When the end tag of *b* is read in, the top three states on the *Runtime Stack* are active and popped out first. Because element *b* has associated branch points in both the *documentBranches* Stack and *queryBranches*

Stack, Q{2,1} and Q{4,1} find matches in the document. Thus the matching process starts for Q{2,1} and Q{4,1} after the branch points are popped out from both stacks. Because the two paths of Q{2,1} have match candidates, Q{2,1} is marked as matched. Notice that this example does not include predicates on paths. If a predicate exists, the matching process has to evaluate all predicates; only matching the path is not enough. Because the Q{2,1} is matched, the current branch from the root of Q{2,0} to the root of Q{2,1} is saved in Q{2,0} as a match candidate of the transit branch of Q{2,0}. If for example Q{2,1} is not matched, the algorithm does not save the candidate for Q{2,0}, because a parent query cannot match if its child query is not matched. Q{4,1} is different from Q{2,1}, because its path 1 has no candidate. So Q{4,1} is simply marked as unmatched at this point.

Finally, when the end tag of *a,* e.g., *a,* is read in, BFilter finds a match of branch points in the *documentBranches* Stack and *queryBranches* Stack. The four branch points, namely Q{1,0}, Q{2.0}, Q{3,0} and Q{4,0}, match branch point a in the document. Because the child query Q{4,0} is unmatched, Q{4,0} is simply marked as unmatched without executing the matching process. Q{1,0} is simple because it has only one path and is marked as matched. Because the child query of Q{2,0} is matched, Q{2,0} is marked as matched after the match of its transit branch succeeds. Similarly the Q{3,0} is marked as unmatched because its child is unmatched. At the end of this process, BFilter checks whether a query is matched or not by looking at its root in the sub-query representation. Thus Q1 and Q2 are matched in this example.

As highlighted in [27], the matching process only starts when a root of a sub-query is popped out from the *queryBranches* Stack. Whenever a match is determined for a sub-query, no matter whether the result is matched or not, all match candidates of the sub-query are deleted. If the result is unmatched, all the descendants of the sub-query are marked as unmatched.

This example demonstrates the difference between YFilter and BFilter. BFilter matches queries backwards and the matching process only starts when branch points match in both the document and the queries. For queries Q1, Q2, Q3 and Q4 in the example, YFilter will execute the matching process at each accepting state for the corresponding queries that are decomposed from the four queries. In particular, Q3 will be decomposed into two simple queries Q3.1 (/a//c/d) and Q3.2 (/a//c/e); Q4 will be decomposed into two simple queries Q4.1 (/a/*/c) and Q4.2 (/a/*/b). From the *queryIndex* Tree we can see that states 9 and 12 will be the accepting states for Q3.1 and Q4.1, respectively. In the case of YFilter, when the two states are reached, the matching process occurs. However, no matching is performed for Q3 and Q4 in BFilter, because the branch points are not matched (*a* and *b* in the document stream, *a* and *c* in Q3, and *a* and * in Q4). In the case of a document having more content after the end tag of element *b* is read. For example, the document stream, ***a/b/c/d/d/c/e/e/b/a***, has more content between ***b*** and ***a***, and if this extra part matches Q3.2 or Q4.2, YFilter not only processes matching for Q3.2 or Q4.2, but also conducts post-processing to verify these decomposed queries and will eventually discover that Q3.1 and Q3.2, as well as Q4.1 and Q4.2, are not matched in the same place in the document. The post-processing in cases like these for YFilter is unnecessary and processing time is wasted.

Unlike YFilter, BFilter cleans up all candidates for the current sub-queries as well as their descendants when the corresponding branch point in the document is read over. BFilter only keeps match candidates for the current portion of the XML document. Thus, if a sub-query and its descendants are matched, it needs not do anything when the new portion of the document arrives. Otherwise, the matching of branch point restarts from the root of the last sub-queries in the new portion of the document. From this example we can see that BFilter is more efficient in dealing with complex queries than YFilter.

4. Experimental Results

This section describes the performance measurement results for a prototype implementation of BFilter.

As discussed earlier, BFilter is built on top of YFilter implementation to reuse existing software components. This implementation is suitable for comparing the filtering algorithms that is the main focus of this paper

The BFilter prototype was implemented using Java 1.6, and run on a PC with an AMD 1.6 GHz processor and 1.0 Gb of main memory running the Windows Vista operating system. The Java virtual machine memory size was set to 256Mb. The data type used in the experiments was the same as in YFilter: News Industry Text Format (NITF) described in [29]. ToXGene [39] was used to generate the XML document. A total of ten documents were used in this experiment. The query generator in the YFilter test suite [29] was used to generate the queries. The following provides the definitions of all parameters and performance metrics used in the performance measurement.

- **Filtering Cost** is the total time (ms) needed to filter the input documents. It is the time period that starts from reading in the start tag of the root of the first XML document from the SAX event-based XML parser, and ends when all documents are filtered and the results of the user requests are obtained. In the experiments filtering cost was used to compare the performance of the filtering algorithms.
- **Query Depth (L)** is for a sequence of element names separated by "/" and "//" (which stand for parent/child relationship and ancestor/descendent relationship, respectively). The number of element names determines the query depth. For a complex query, query depth is the depth of its longest path.
- **Number of Predicates (P)** is the total number of predicates in a query. An element name in a user request may contain predicates. When matching such a query, both the element name and its predicates have to find a match in the XML document.
- **Number of Nested Paths (NP)** is the number of nested paths in a complex query. A simple query has no nested path.
- **Probability of ancestor/descendant (//)** is the probability that the ancestor/descendant relationship appears in a query. The ancestor/descendant relationship creates a self-loop state in NFA and can increase the number of active states during matching. This parameter is used to measure how sensitive the performance of an algorithm is to an increase in the number of occurrences of "//".
- **Probability of wildcard (*)** is the probability that "*" appears in a query as an element name. The wildcard "*" matches any element from a XML document being filtered and thus the number of active states can increase with an increase of the probability of "*". This parameter is used to measure how sensitive an algorithm is to an increase of the number of occurrences of "*".

A number of experiments have been conducted and their results are presented in the subsequent subsections.

4.1. Test Case 1 — Branch Point Mismatching

This case demonstrates a scenario in which a complex query cannot achieve a branch point matching in a XML document. The query and XML document are shown in Fig. 10.

This query is complex. Its last branch point is *w*, having two branches "/b" and "/c". However, all the three nodes labeled *w* in the XML document are not branch points. After the start tags *nitf*, *head*, *a*, *b*, *w* and *c* are read in sequentially, the end tags *c* and *w* follows. At this point the node *w* (id = "9") is detected. Because it is not a branch point in the document, the matching for the last branch point *w* in the query fails. Similarly, the other two nodes labeled w with id = "99" and id = "9" in the document also fail to match the query's branch point. Thus, the matching of the elements at higher levels of the query has never been performed in BFilter. When Yfiter processes this filtering, it decomposes the query into three simple

queries:

/nitf[@change.time=1]/head[@id=9]/a/c
/nitf[@change.time=1]/head[@id=9]/a/b/w/b
/nitf[@change.time=1]/head[@id=9]/a/b/w/c

```
Query:      /nitf[@change.time=1]/head[@id=9]/a[/c]/b/w[/b]/c
Document:   <nitf change.date="1" change.time="1" id="5" uno="11" version="1">
              <head id="9">
                <a id="9">
                  <b id="9">
                    <w id="9">
                      <c id="9"></c>
                    </w>
                    <w id="99">
                      <b id="19"></b>
                    </w>
                  </b>
                  <c id="9"></c>
                  <b id="9">
                    <w id="9">
                      <c id="9"></c>
                    </w>
                  </b>
                </a>
              </head>
            </nitf>
```

Fig. 10. The sample query and XML document.

Because each decomposed simple query can find a match in the document, YFilter spends time on matching each of them and finally discovers they are not matched in the same place in the document. Fig. 11 shows the comparison of the filtering time for the two filters. Because the filtering time for this test case is small, hence we perform the test case 20 times, 40 times, 60 times and 80 times. The results indicate that BFilter is more efficient than YFilter in the case of branch point mismatching.

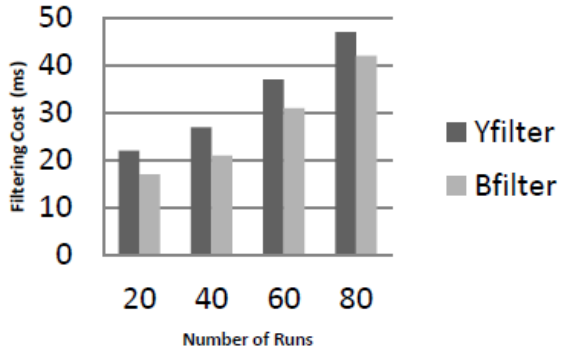


Fig. 11. Comparison of Yfilter and Bfilter for test case 1.

4.2. Test Case 2 — Test of Queries without a Branch point

This case is used to demonstrate a scenario in which the current implementation of BFilter is slower than YFilter while handling only simple queries. This is because the current implementation of BFilter reuses the

components in YFilter and reconstructs the classes on top of them to meet its needs. It has no performance gain to offset the overhead in the case of simple queries. A different implementation of BFilter that does not use the YFilter components is expected to avoid this issue.

Fig. 12 shows a comparison of the filtering time for various numbers of queries used in the matching. The criteria used to generate queries are shown in the figure: the probability of "*" and "//" are 20% and 0, respectively. The number of predicates (P) is 6, the number of branches is 0, and the query depth (L) is 6. The number of queries generated varies from 500 to 2000. None of the queries has branches.

Although the result in Fig. 12 is for a particular set of criteria, the variation of the parameters in the criteria does not change the result in the case of simple queries. Considering the fact that BFilter is more efficient than YFilter in dealing with complex query, we believe the current implementation has room for improvement.

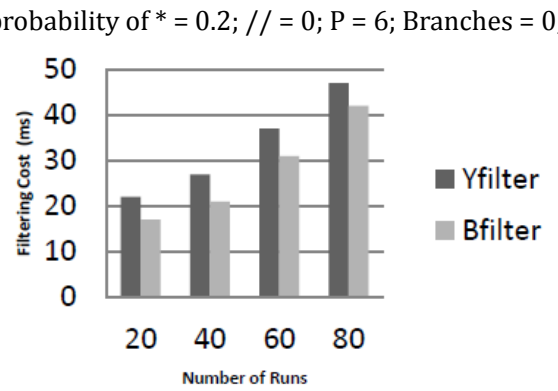


Fig. 12. Comparison of Yfilter and Bfilter for test case 2.

4.3. Test Case 3 — Random Documents and Queries

This section presents experiments and results using different configurations for YFilter and BFilter. All the experiments are based on *NITF* documents and *NITF* queries generated randomly using the generator described [29], but the same documents and queries are used for both YFilter and BFilter. Table 1 and Table 2 demonstrate the filtering time, including specific values for the different parameters used for each experiment. The value for NP, the number of nested paths, is either 1 or 2 for the experiments to evaluate queries that are modest complex. The effect of NP is illustrated in Section 4.4.2.

Table 1. Filtering Time (ms) for L = 6, * = 0.2, // = 0, P = 3, NP = 1

	1000 queries		2000 queries	
XML docs size	**BFilter**	**YFilter**	**BFilter**	**YFilter**
5 KB	10	10	5	5
10 KB	15	10	15	15
33 KB	20	25	25	25
193 KB	45	45	55	55

Table 2. Filtering Time (ms) for L = 6, * = 0.2, // = 0, P = 3, NP = 2

	1000 queries		2000 queries	
XML docs size	**BFilter**	**YFilter**	**BFilter**	**YFilter**
5 KB	10	10	10	10
10 KB	10	10	15	15
33 KB	25	30	25	25
193 KB	45	50	50	65

Because the documents and queries are randomly generated, there is no close correlation for the 1000 queries and the 2000 queries generated. Hence, there is no close correlation for the results for 1000 and 2000 queries. As the results indicate, BFilter has an equal or lower filtering time in most cases observed from our experiments.

4.4. Measurement Results for Different Query Attributes

This section presents the effect of five different parameters on the performance of BFilter and YFilter. The selected values of each parameter are outlined below:

- Query depth (L): 3, 6.
- Number of nested paths (NP): 0, 1, 2, 3, 4, 5.
- Probability of wildcard (*): 0, 0.1, 0.2, 0.4.
- Probability of ancestor/descendant relation (//): 0, 0.1, 0.2, 0.4,
- Number of predicates (P): 0, 2, 5, 10.

A total of ten XML documents were generated and used in experiments. The default level of these documents is 6. A query cannot find a match if its depth is greater than 6, so the selected values for the query depth are 3 and 6. Since the number of nested paths cannot be greater than the maximum value of the query depth in a query, the number of nested paths is varied from 0 to 5. The probabilities of wildcard and ancestor/descendant relation range from 0 to 0.4. The number of predicates varies from 0 to 10. The ranges of parameters used in the experiments are apt for analyzing the relative performance of BFilter and YFilter. Because the total number of permutations of the five variables is very large, one parameter is varied at a time in each of the experiments. The number of queries generated by each group of values from the five parameters is set as 100.

4.4.1. The effect of the query depth

The purpose of this experiment is to determine the effect of query depth on the filtering time. In this experiment, the number of nested paths is varied from 0 to 5, and two parameter sets are used:

Set 1: the probabilities of "*" and "//" are 0, the number of predicates (P) is 2, and the query depth (L) is 6.

Set 2: the probabilities of "*" and "//" are 0, the number of predicates (P) is 2, and the query depth (L) is 3.

Fig. 13 displays two graphs. Fig. 13(a) uses the parameter set 1, and shows the result when the query depth is 6. Fig. 13(b) uses the parameter set 2, and shows the result when the query depth is 3.

In both Fig. 13(a) and 13(b) BFilter demonstrates a better performance in comparison to YFilter when the number of nested paths is larger than 0. Fig. 13(a) shows that the improvement in performance for BFilter increases significantly with an increase in the number of nested paths in the parameter set 1. For instance, when NP = 5, the results for BFilter and YFilter are about 8ms and 15ms, respectively, which has a large performance gap. When the number of nested paths increases, the probability of branch point matching decreases. Thus, the processing time for matching is reduced in BFilter, and hence BFilter's filtering cost decreases.

Fig. 13(b) shows that in the cases where query depth L is 3, the number of nested paths has no significant effect on the filtering cost after it reaches 3. This is because the query generator is a reused component from YFilter which can create at most one branch at each level of a query. YFilter does not deal with recursive nested paths (a branch containing another branch). If the query depth is 3, the query generator still only creates 3 nested paths for a query even though the number of nested paths (NP) is set to be greater than 3. So the query depth (L) will be set as 6 for the remaining experiments.

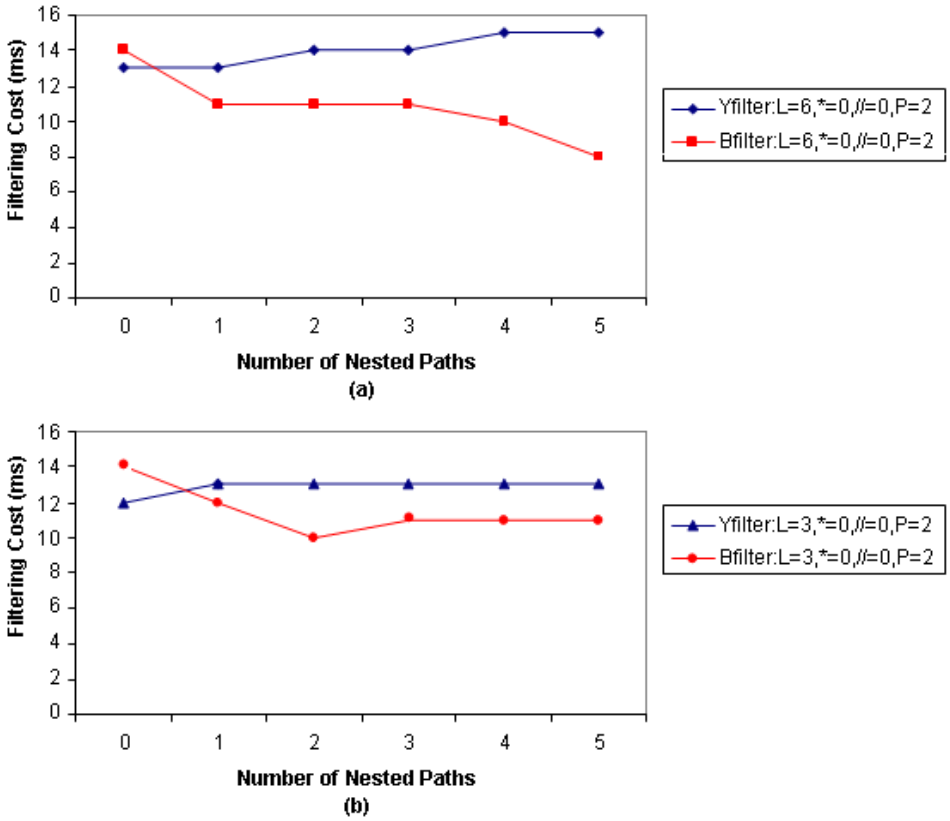


Fig. 13. The effect of the query depth (a) Set 1; (b) Set 2.

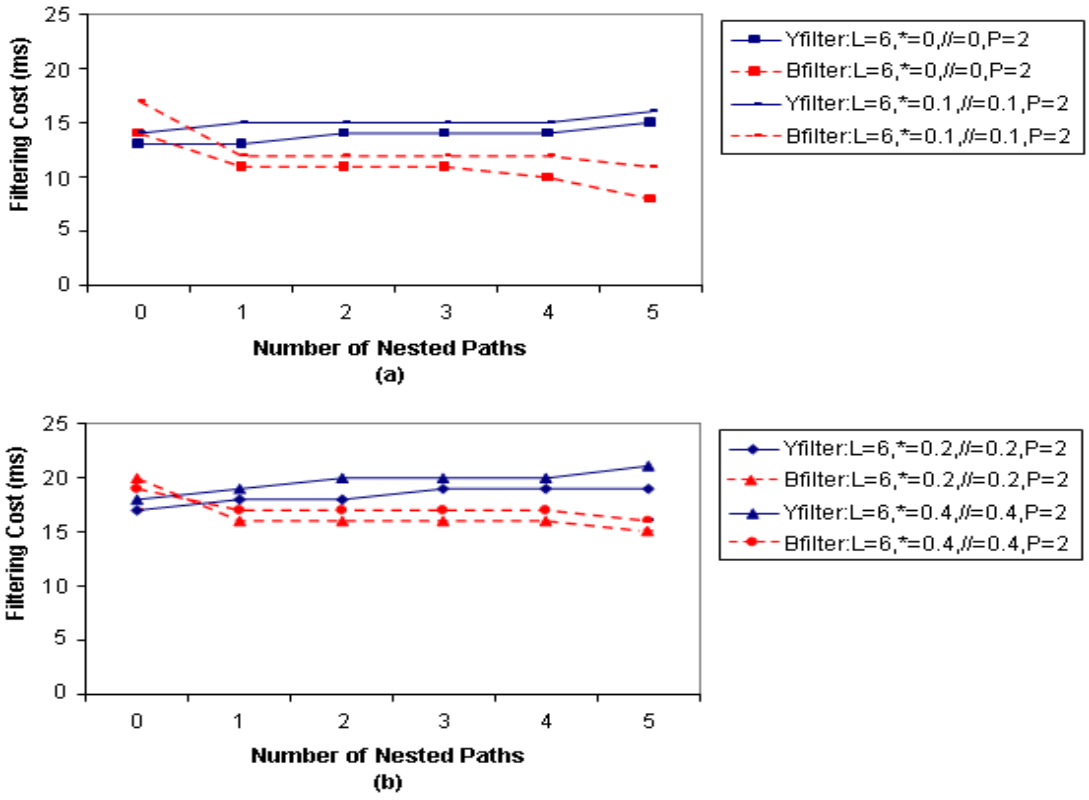


Fig. 14. The effect of the number of nested path [27].

4.4.2. The effect of the number of nested paths

In this experiment, the number of nested paths is varied from 0 to 5. There are four groups of comparisons that use different sets of parameters as shown in Fig. 14.

Fig. 14 contains two graphs. Each graph shows two groups of comparisons. Fig. 14(a) presents the results for group 1 and 2. Fig. 14(b) presents the results for group 3 and 4. In every group of comparison, BFilter performs better than YFilter except when the number of nested paths equals zero; in other words, when the queries are simple. This is because the current implementation of BFilter reuses the components in YFilter and reconstructs the classes on top of the YFilter components. It has no performance gain to offset the overhead in the case of a simple query.

In YFilter, when the number of nested paths increases the number of simple queries decomposed from complex queries also increases. This is because YFilter decomposes a complex query by creating a simple query for each nested path. Therefore, YFilter's filtering cost tends to increase because it needs to do more work during post- processing to verify the separated query as a whole. In the case of BFilter, because the number of nested paths increases, the probability of branch point matching decreases, as explained in Section 3.1. Thus, the processing for matching is reduced in BFilter, and hence BFilter's filtering cost decreases [27]. The gap of filtering time for YFilter and BFilter generally increases as the number of nested paths increases. The performance gain for BFiliter can be large in percentage when NP is 4 or 5 for complex queries.

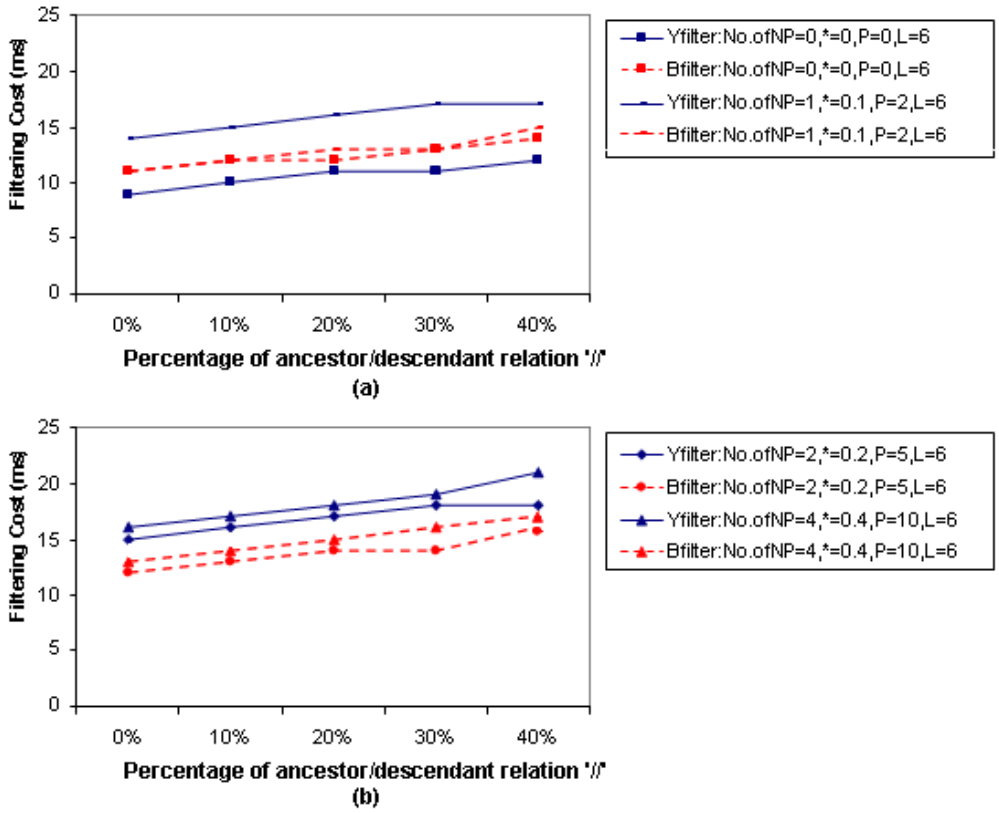


Fig. 15. The effect of the probability of "//" [27].

4.4.3. The effect of the probability of "//"

In this experiment, the probability of "//" that appears in the queries is varied from 0 to 40%. There are four groups of comparisons that use different sets of parameters as shown in Fig. 15.

BFilter uses NFA to implement query indexing as YFilter does. The increase of the number of "//" in queries causes an exponential increase in the number of active states in the NFA. Both of YFilter and BFilter

suffer from this overhead. However, the method that BFilter uses to perform matching from the underlying NFA is different from YFilter.

There are two graphs in Fig. 15. Each graph shows two groups of comparisons. Fig. 15(a) presents the results for group 1 and 2. Fig. 15(b) presents the results for group 3 and 4. The results show that BFilter is faster than YFilter in all cases when the number of nested paths, e.g., NP, is not zero. When NP = 0, the results for YFilter are better than that of BFilter, as shown in Fig. 15(a).

As mentioned above, the current BFilter implementation reuses the query index tree of YFilter. It suffers the same overhead as explained in YFilter due to the large number of active states produced by '//'. This can be seen from the figure: the filtering cost of both YFilter and BFilter increases when the probability of ancestor/descendant relationship '//' increases [27]. However, BFilter gains in the matching process but not in the query indexing. When the technique of backward matching branch point is used in BFilter, although the number of active states is large in *Runtime Stack* in the NFA, these active states may not actually trigger the matching operation. If a branch point of a query cannot find match, all of the active states rooted at this branch point will simply be popped out from the stack.

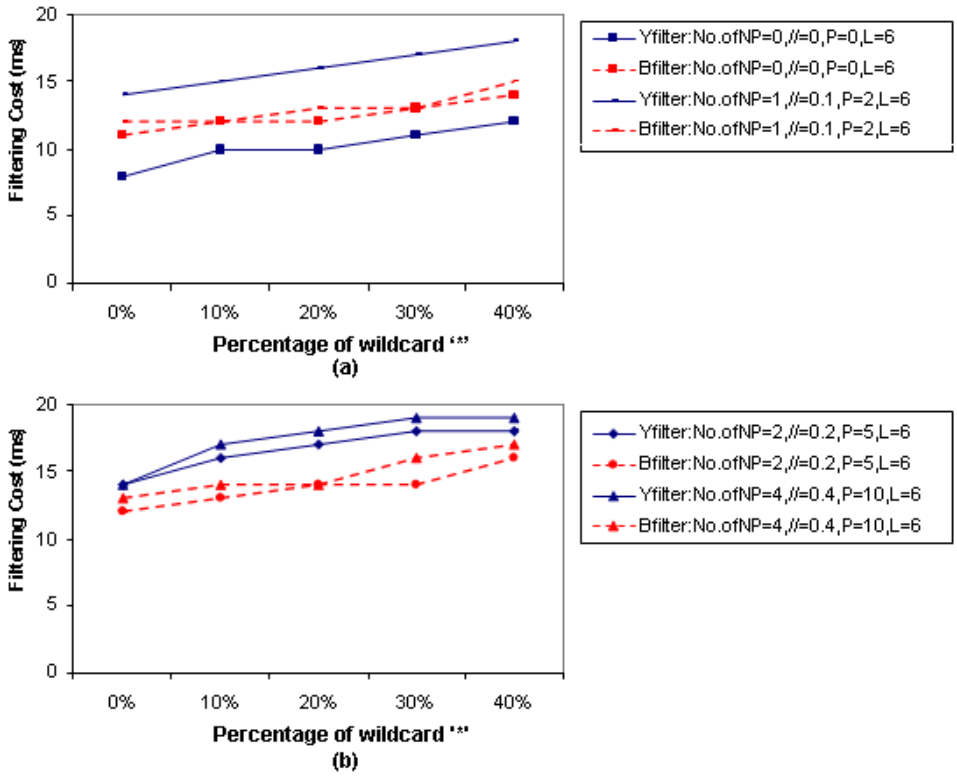


Fig. 16. The effect of the probability of "*".

4.4.4. The effect of the probability of "*"

In this experiment, the probability of "*" that appears in the queries is varied from 0 to 40%. There are four groups of comparisons that use different sets of parameters as shown in Fig. 16.

The effect of the probability of "*" in queries on the NFA is similar to that of "//". An increase in the number of "*"also causes the number of active states to increase because a "*" matches any tag from the document. Each tag creates an active state on the Runtime Stack. However, it does not lead to an exponential increase in the number of active states in the NFA.

There are two graphs in Fig. 16. Each graph shows two groups of comparisons. Fig. 16(a) presents the results for group 1 and 2. Fig. 16(b) presents the results for group 3 and 4. Similar to results presented in

Fig. 15, the results show that BFilter is faster than YFilter in all cases except when the number of nested paths is zero.

The results also demonstrate that when the percentage of wildcard '*' increases, the filtering cost of both YFilter and BFilter increases. This is because the wildcard matches any incoming tag from a XML document so that both YFilter and BFilter need to do more work due to the increase in number of active states. The extra work slows down the matching process.

5. Conclusions and Future Research

Many XML filtering and matching techniques have been proposed. As the volume, variety, and velocity of data increase considerably, the efficiency becomes a crucially important. We proposed a novel XML filtering and matching algorithm, BFilter, which performs XML message filtering and matching by leveraging branch points in both XML documents and user requests or queries. BFilter matches branch points backwards in a bottom-up fashion to defer further matching processes until branch points in both the XML document and a user query match. Unlike the well-known YFilter, BFilter matches requests with branches without having to decompose them. It treats a complex query as a whole, and no post-processing is needed for a complex query. In comparison to other XML filter techniques, BFilter not only performs backward matching (which is more efficient than a top-down approach), but also detects and utilizes branch point matching as a precondition for further steps. In other words, the matching of query branches is delayed until the branch point they attach to is matched. In this way, BFilter has a high probability of detecting mismatches earlier in the matching process. Thus, as illustrated in theoretical analysis, XML message filtering in BFilter can be performed more efficiently in comparison to the other algorithms. We have also demonstrated by the experimental results that BFilter has better performance than that of YFilter. Further, in general, the performance gain for BFilter increases, as the queries become more complex.

The measurement results from the current implementation show that BFilter performs better than YFilter while handling complex queries. Because the current BFilter implementation used in this research is built on the top of the YFilter implementation, BFilter shares some of the characteristics of the YFilter implementation that was used in this research. The gain of BFilter is not from the specific implementation used in this paper, but from the filtering algorithm based on the novel concept of backward branch point matching introduced in this research. In case all the queries generated are simple, BFilter is slower than YFilter, which is due to the overheads from wrapping of the existing components of YFilter that are reused in the current implementation of BFilter. One future direction is to conduct reengineering of YFilter design and implementation, so that only the necessary components for BFilter are used for higher efficiency.

While performing matching, BFilter filters a document part by part. Thus, BFilter can specify the matched parts of a document for a particular query. This adds an option for upstream filters to deliver the appropriate parts of a document to downstream filters in a pub/sub system. The potential benefit is lower processing time for shorter messages for subsequent brokers or subscribers. The approach warrants further research.

Acknowledgment

The authors wish to acknowledge Alcatel-Lucent and Ontario Centres of Excellence for their financial support throughout the project. We also thank Yang Cao for performing some experiments for the paper.

References

[1] Abdul, N. M., Suresh, B. G., & Kumar, P. S. (2009). SFilter: A simple and scalable filter for XML streams. *Proceedings of the International Conf. on Management of Data*.

[2] Banerjee, S., Bhattacharjee, B., & Kommareddy, C. (2002). Scalable application layer multicast. *Proceedings of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications* (pp. 205–217).

[3] Bhargava, A., Kothapalli, K., Riley, C., Scheideler, C. , & Thober, M. (2004). Pagoda: A dynamic overlay network for routing, data management, and multicasting. *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures* (pp. 170–179).

[4] Cao, F., & Singh, J. P. (2005). MEDYM: Match-early and dynamic multicast for content-based publish-subscribe service networks. *Proceedings of the International Conference on Middleware* (pp. 292–313).

[5] Chand, R., & Felber, P. (2003). A scalable protocol for content-based routing in overlay networks. *Proceedings of the IEEE International Symposium on Network Computing and Applications* (pp. 123-130).

[6] Chand, R., & Felber, P. (2004). XNET: A reliable content-based publish/subscribe system reliable distributed systems. *Proceedings of the International Symposium on Reliable Distributed System* (pp. 264–273).

[7] Chu, Y., Rao, S., & Zhang, H. (2002). A case for end systems multicast. *IEEE Journal on Selected Areas in Communication: Networking Support for Multicast*, *20(8)*, 1456–1471.

[8] Diao, Y., Rizvi, S., & Franklin, M. (2004). Towards an Internet-scale XML dissemination service. *Proceedings of the International Conference on very Large Databases* (pp. 612–623).

[9] Florescu, D., Hillery, C., Kossmann, D., & Lucas, P. (2003). The BEA/XQRL streaming XQuery processor. *Proceedings of the of International Conference on VLDB* (pp. 997–1008).

[10] Green, T. J., Gupta, A. Miklau, G. Onizuka, M., & Suciu, D. (2004). Processing XML streams with deterministic automata and stream indexes. *ACM Trans. on Database Systems*, *29(4)*, 752–788.

[11] Ives, Z., Halevy, A., & Weld, D. (2002). An XML query engine for network-bound data. *The VLDB Journal*, *11(4)*, 380–402.

[12] Kwon, J., Rao, P., Moon, B., & Lee, S. (2005). FiST: Scalable XML document filtering by sequencing twig patterns. *Proceedings of the International Conference on Very Large Databases* (pp. 217–228).

[13] Kwon, J., Rao, P., Moon, B., & Lee, S. (2009). Fast XML document filtering by sequencing twig patterns. *ACM Transactions on Internet Technology*, *9(4)*, 1–51.

[14] Opyrchal, L., Astley, M., Auerbach, J., Banavar, G., Strom, R., & Sturman, D. (2000). Exploiting IP multicast in content-based publish-subscribe systems. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms* (pp. 185–207).

[15] Saxena, P., & Kamal, R. (2013). System architecture and effect of depth of query on XML document filtering using Pfilter. *Proceedings of the International Conference on Contemporary Computing* (pp. 192–195).

[16] Yoneki, E., & Bacon, J. (2005). Distributed multicast grouping for publish/subscribe over mobile ad hoc networks. *Proceedings of the IEEE International Conference on Wireless Communications and Networking*, *vol. 4*, (pp. 2293–2299).

[17] Zheng, Q.-H., Jiang, S., Zhang, F., Peng, T., & Chen, C. (2008). TapMulti: A scalable and low-delay application-layer multicast protocol on tapestry overlay network. *Information Technology Journal*, *7(5)*, 728–736.

[18] Langerman, S., Lodha, S., & Shah, R. (2001). Algorithms for efficient filtering in content-based multicast. *Proceedings of the 9th Annual European Symposium on Algorithms, Lecture Notes in Computer Science*, *2161*, 428–439.

[19] Lee, S.-J., Gerla, M., & C.-C., Chiang. (1999). On-demand multicast routing protocol. *Proceedings of the*

IEEE International Conference on Wireless Communications and Networks (pp. 1298-1302).

[20] Paris, G., Arrufat, M., Lopez, P. G., & Sanchez-Artigas, M. (2008). An application layer multicast for collaborative scenarios: The OMCAST protocol. *Proceedings of the International Conference on Networking*, 99–104.

[21] Segall, B., & Arnold, D. (1997). Elvin has left the building: A publish/subscribe notification service with quenching. *Proceedings of the Australian UNIX and Open Systems User Group Conference* (pp. 243-255).

[22] Yan, T. W., & Garcia-Molina, H. (1999). The SIFT information dissemination system. *ACM Transactions on Database Systems*, *24(4)*, 529–565.

[23] Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13(7)*, 422–426.

[24] Altinel, M., & Franklin, M. (2000). Efficient filtering of XML documents for selective dissemination of information. *Proceedings of the International Conference on Very Large Data Base* (pp. 53-64).

[25] Burstein, P., & Rizvi, S. (2009). Peer-to-peer result dissemination in high-volume data filtering. Retrieved August 20, 2009, from http://www.cs.berkeley.edu/~kubitron/courses/cs294-4-F03/projects/rizvi_burst.pdf

[26] Fenner, W., Rabinovich, M., Ramakrishnan, K., Srivastava, D., & Zhang, Y. (2005). XTreeNet: Scalable overlay networks for XML content dissemination and querying (synopsis). *Proceedings of the International Workshop on Web Content Caching and Distribution*, 41–46.

[27] Dai, L., Lung C.-H., & Majumdar, S. (2010). BFilter — A XML Message filtering and matching approach in publish/subscribe systems. *Proceedings of the IEEE GLOBECOM* (pp. 1–6).

[28] Zhao, H., Xia, W., & Zhao, J. (2012). The research on XML filtering model using lazy DFA. *Journal of Software*, *7(8)*, 1759–1766.

[29] Diao, Y., Altinel, M., Franklin, M., Zhang, H., & Fischer, P. M. (2003). Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Database Systems*, *28(4)*, 467–516.

[30] Martins, R., & Pereira, J. (2011). WFilter: Effcient XML filtering for large scale publish/subscribe systems. *Proceedings of the Symp. on INForum* (pp. 1–14).

[31] Selcuk, C. K., Hsiung, W.-P., Chen, S., Tatemura, J., & Agrawal, D. (2006). AFilter: Adaptable XML filtering with prefix-caching and suffix-clustering. *Proceedings of the International Conference on Very Large Data Bases* (pp. 559–570).

[32] Boone, P. (2007). A hybrid XML filtering engine for publish/subscribe content-based routing. *Technical Report, the School of Computer Science*, Carleton University.

[33] Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., & Candan, K. S. (2008). Scalable filtering of multiple generalized-tree-pattern queries over XML streams. *IEEE Transactions on Knowledge and Data Engineering*, *20(12)*, 1627–1640.

[34] Chen, S., Li, H.-G., Tatemura, J., Hsiung, W.-P., Agrawal, D., & Candan, K. S. (2006). Twig2Stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. *Proceedings of the International Conference on Very Large Data Bases* (pp. 283-294).

[35] Chen, Z., Jagadish, H. V., Lakshmanan, L. V. S., & Paparizos, S. (2003). From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. *Proceedings of the International Conference on Very Large Databases* (pp. 237–248).

[36] Sun, W., Qin, Y., Yu, P., Zhang, Z., & He, Z. (2008). HFilter: Hybrid finite automaton based stream filtering for deep and recursive XML data. *Database and Expert Systems Applications, Lecture Notes on Computer Science*, 566–580.

[37] Luo, B., Lee, D., Lee, W.-C., & Liu, P. (2004). QFilter: Fine-grained run-time XML access control via Nfa-based query rewriting. *Proceedings of the International Conference on Information and Knowledge*

Management (pp. 543–552).
[38] Sax project organization. (2001). Retrieved May 18, 2008, from http://www.saxproject.org
[39] Barbosa, D., Mendelzon, A., Keenleyside, J., & Lyons, K. (2002). ToXgene: A template-based data generator for XML. *Proceedings of the International Workshop on the Web and Databases* (pp. 49-54).

Liang Dai is a M.Sc. student at the School of Computer Science at Carleton University. He received the B.Sc. degree in computer science at Carleton University (2006). His research interests are in XML message transformation and publication subscription system.

Chung-Horng Lung received the B.S. degree in computer science and engineering from Chung-Yuan Christian University, Taiwan and the M.S. and Ph.D. degrees in computer science and Engineering from Arizona State University. He was with Nortel Networks from 1995 to 2001. In September 2001, he joined the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he is now a full professor. His research interests include are communication networks, software engineering, and distributed systems.

Sikharesh Majumdar is a professor and the director of the Real Time and Distributed Systems Research Centre at the Department of Systems and Computer Engineering at Carleton University in Ottawa, Canada. He represents Carleton University in the board managing the Huawei-TELUS Centre of Innovation for Enterprise Cloud Services that focuses on collaborative research among Huawei, TELUS and Carleton. Dr. Majumdar is a member of the faculty team actively involved with Carleton University's Canada-India Centre for Excellence. He holds a Ph.D. degree in computational science from University of Saskatchewan, Saskatoon, Canada. His research interests are in the areas of cloud and grid computing, operating systems, middleware and performance evaluation. Dr. Majumdar actively collaborates with the industrial sector and has performed his sabbatical research at Nortel and Cistech. He is the area editor for the Simulation Modelling Practice and Theory journal published by Elsevier. Dr. Majumdar is a member of ACM and IEEE and was a Distinguished Visitor for the IEEE Computer Society from 1998 to 2001.