A Keyword-Based Filtering Technique of Document-Centric XML using NFA Representation

Changwoo Byun, Kyounghan Lee, and Seog Park

Abstract**—** XML is becoming a de facto standard for online data exchange. Existing XML filtering techniques based on a publish/subscribe model are focused on the highly structured data marked up with XML tags. These techniques are efficient in filtering the documents of data-centric XML but are not effective in filtering the element contents of the document-centric XML. In this paper, we propose an extended XPath specification which includes a special matching character '%' used in the LIKE operation of SQL in order to solve the difficulty of writing some queries to adequately filter element contents using the previous XPath specification. We also present a novel technique for filtering a collection of document-centric XMLs, called Pfilter, which is able to exploit the extended XPath specification. We show several performance studies, efficiency and scalability using the multi-query processing time (MQPT).

Keywords**—**XML Data Stream, Document-centric XML, Filtering Technique, Value-based Predicates.

I. Introduction

THE eXtensible Markup Language (XML) can be used to mark up content in various ways [1]. Based on the content, XML documents are often broken down into two categories: *data-centric* and *document-centric* XML. Data-centric XML is a highly structured data marked up with XML tags. On the other hand, document-centric XML refers to loosely structured documents (often text) marked up with XML [2]. An example of a document-centric XML is *Really Simple Syndicate* (RSS) files. The top_stories.xml (available at http://rss.cnn.com/rss/edition.rss) RSS files that are disseminated from CNN.com have an average of 52 elements and a maximum file depth of 4. The structure of this XML is very simple because the average length of element contents is about 56 characters. However, it has long element contents between XML tags.

Many XML filtering techniques based on data-centric XML has been studied in the database research community. Although the previous filtering techniques may be applied to the dissemination of the document-centric XML (e.g., RSS), they are not insufficient. The main reason is that they do not support a special matching character for information retrieval of element contents in document-centric XML. In other words, XPath [3] or XQuery [4] exploit the *text()* function to process value-based predicates. Since the *text()* function simply supports the *string equality comparison operation* between an operand of a predicate and the element contents of XML, it is proper to write the value-based predicates used in data-centric XML containing short element contents. However, it is troublesome to make value-based predicates in document-centric XML.

In this paper we extend the idea of structure matching of the XML filtering system, and propose a novel XML filtering technique, called 'Pfilter', which is adequate for the document-centric XML. In order to copy with value-based predicates in document-centric XML, we make an addition, a special matching character '%' into the XPath specification, which is similar to the LIKE operator of the SQL statement. In addition, since the document-centric XML filtering is radically different from traditional XML query processing that depends heavily on such information for query processing and optimization, we propose a novel algorithm for processing the value-based predicates.

The Contributions of the Pfilter technique are as follows:

- First, the Pfilter is a filtering engine capable of processing a significant amount of value-based predicates.
- Second, the Pfilter proposes a special matching character '%' in the operand string of the traditional XPath *text()* function.
- Third, Pfilter separates structure matching and value-based predicate matching. So, value-based predicate matching technique may be applied to various and different structure matching techniques.

The rest of the paper is organized as follows: Section 2 briefly summarize the previous techniques in a XML filtering system. In Section 3, we propose a novel technique, which is to append a modified *Aho-Corasick dictionary matching tree* [5]. In Section 4, with respect to *maintenance cost*, *scalability*, and *efficiency*, we compare our results with a variant of Yfilter [6] with the original Aho-Corasick dictionary matching tree. Finally, Section 5 summarizes our work.

II. Related Works

A traditional RDBMS executes the *selection* operation prior to everything else in the query plan. Such modification of the query plan is an attempt to improve the efficiency of query

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No.R01-2006-000-10609-0).

processing by reducing the intermediary results and minimizing the input of the next operation. However, applying this heuristic to the XML filtering system in a manner similar to the Yfilter approach does not yield as good performance as expected. In addition to Yfilter, there have been numerous XML filtering techniques proposed. However, most studies focused on structural matching due to the XML characteristics that involve multiple nests of the upper elements [7]-[11]. They are very effective for applications in the data-centric XML, but are insufficient for filtering document-centric XML. This is not to say that there have been no XML filtering system research efforts to effectively process the value-based predicates.

The XML filtering system that uses XPush [12] and RDBMS [13] defines and shares the *atomic predicate*, which refers to the predicates that constitute the elements of conjunction in value-based predicates. Sharing atomic elements in this manner allows a short-cut evaluation of the predicate conjunctions, and enhances the processing efficiency of the value-based predicates. In order to effectively process the value-based predicates, XSQ [14] exploits the pushdown transducer to share the atomic predicates. This technique enables the sharing of numeric and string constants, and as far as we know, it is the most effective technique for processing value-based predicates. However, the number of states created by XSQ increases by $O(2^n)$ -fold (n is the number of occurrences of '*' in the test node of all queries) as compared to using NFA. There is the problem of consuming much memory.

However, it is difficult for the XML filtering systems mentioned above to apply in the XML filtering system with value-based predicates for which numerous queries should be registered. XSQ also has the disadvantage of not being able to use structure matching that shares the common prefix element.

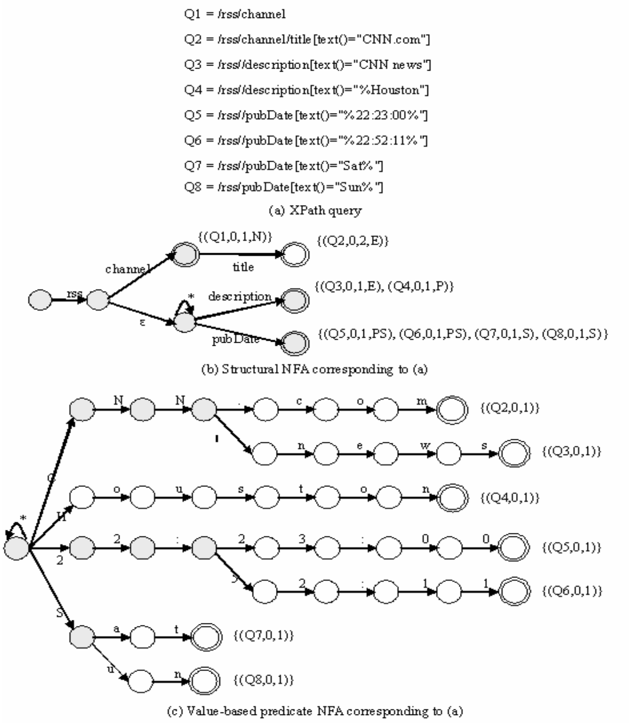
III. Value-Based Predicate Execution in Pfilter

A. Shared Value-Based Predicate Matching

1) Representation of the Value-based Predicate

In order to perform an efficient execution of value-based predicates, a novel technique should identify the common prefix characters of the operand in the predicate and share the processing among them. In comparison, the Yfilter sequentially processes the predicate operands for each predicate. The technique combines all value-based predicate operands into an NFA form, *i.e.*, a single *finite state machine* (FSM). This value-based predicate NFA has two characteristics: One is a single *accept state* which exists for each predicate operand and the other is the common prefix characters that appear only *once*.

Fig. 1 shows examples of a structural NFA representing eight queries and a value-based predicate NFA. Since the structural NFA closely resembles the structural NFA approach of Yfilter [6].



```
Q1 = /rss/channel
Q2 = /rss/channel/title[text()="CNN.com"]
Q3 = /rss//description[text()="CNN news"]
Q4 = /rss//description[text()="%Houston"]
Q5 = /rss//pubDate[text()="%22:23:00%"]
Q6 = /rss//pubDate[text()="%22:52:11%"]
Q7 = /rss//pubDate[text()="Sat%"]
Q8 = /rss/pubDate[text()="Sun%"]
```

(a) XPath query

(b) Structural NFA corresponding to (a)

(c) Value-based predicate NFA corresponding to (a)

Fig. 1 NFA-based representation of the XPath query

A point of note here is that unlike the Yfilter's structural NFA, each *accept state* (query ID, path ID, level, matching character info.) has additional information constructed in pairs. In matching character information, "N" refers to *no predicate*, "E" to *predicate exists but '%' not occurred*, "P" to *predicate exists and prefix '%' occurred*, and "PS" to *predicate exists and prefix-suffix '%' occurred*. The directed edge represents transition, and the characters appearing on the edge of the value-based predicate NFA refer to the input that induces transition. Finally, the shaded circle in the two NFA signifies that each location step of the path expression or the characters of the operand are shared in the structural NFA and the value-based predicate, respectively. The points of caution in such NFA form representation are that the two NFA share the common prefix, have multiple accept states, and that the special matching character '%' is not represented in the value-based predicate NFA.

The operand of each value-based predicate has a single accept state in the value-based predicate NFA. When the special matching character '%' is removed, a predicate with an identical operand can share the accept state.

2) Constructing a Combined NFA

The value-based predicate NFA shown in Fig. 1 (c) is the result of consecutively applying the method of constructing with respect to the eight queries that will be explained below. In fact, the structure of the value-based predicate NFA is simpler than the structural NFA, which considers '*' and '//'. The directional graphs, referred to as the *value-based predicate NFA fragments,* shown in Fig. 2 correspond to the eight queries of Fig. 1 (a). However, since query Q1 only represents path expression, only seven value-based predicate NFA fragments are displayed in Fig. 2.

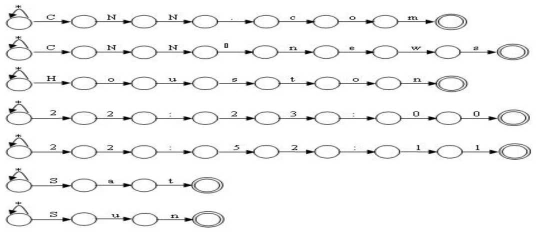


Fig. 2 Value-based predicate NFA fragments

Let us denote each value-based predicate NFA fragment as NFA_v. NFA_v can be combined into a single, value-based predicate NFA in a simple manner. As shown in Fig. 1 (c), there always exists an *initial state* that is shared among all NFA_v. The '*' that induces transition for all characters in the initial state of the value-based predicate NFA makes this model *nondeterministic*. The value-based predicate NFA, combined to insert a new $NFA_{v,}$ either (1) reaches the accept state of NFA_v or (2) repetitiously searches until there is no transition that corresponds to NFA_v. In the first case, the final state becomes the accept state (query ID, path ID, level), and the (query ID, path ID, level) pair is associated with the accept state. In the second case, a new *branch* is constructed in the final state of arrival of the combined value-based predicate NFA. This branch is structured with a non-matched transition of NFA_v. This value-based predicate NFA insertion/deletion algorithm is described in Fig. 3.

```
Input: inserting value-based predicate NFA fragments, $NFA_{fragments}$
Output: value-based predicate NFA, $NFA_{value}$
**for each** $NFA_v$ ∈ $NFA_{fragments}$ **do**
   **for each** state $s$ ∈ $NFA_v$ **do**
      **if** $s$ is accept state **then**
         associate $s$ with (query ID, path ID, level) pair w.r.t. $NFA_v$;
      **if** $s$ is not shared with $NFA_{value}$ **then**
         $NFA_{value}$ is branched off using remainder states of $NFA_v$;
         **break**;
      **else if** // $s$ is shared with $NFA_{value}$
         do nothing;
**return** $NFA_{value}$;
```

(a) Inserting algorithm

```
Input: deleting value-based predicate NFA fragments, $NFA_{fragments}$
Output: value-based predicate NFA, $NFA_{value}$
**for each** $NFA_v$ ∈ $NFA_{fragments}$ **do**
   **for each** state $s$ ∈ $NFA_v$ **do**
      **if** $s$ is not shared with $NFA_{value}$ **then**
         delete state $s$ form $NFA_{value}$;
         **if** $s$ is accept state **then**
            delete (query ID, path ID, level) pair w.r.t. $NFA_v$ from $NFA_{value}$;
      **else if** // $s$ is shared with $NFA_{value}$
         do nothing;
**return** $NFA_{value}$;
```

(b) Deletion algorithm

Fig. 3 Operands insertion/deletion algorithm for value-based predicate NFA

In the NFA model of Fig. 1 (c), if a state with a *self-loop* (*i.e.*, state ID 0) is transitioned into the next state, then transition takes place to the state with the self-loop and the current state when the next character is inputted. This signifies that the number of states to process during the next character input has increased from one to two.

The important thing in the construction of a value-based predicate NFA examined so far is the fact that the process is *incremental*. In turn, since a new operand can be easily added to the value-based predicate NFA, the advantage of this approach is that maintenance is simple for the NFA.

3) Implementing NFA Structure

In fact, automata can be implemented using various data structures. Therefore, in order to execute an effective value-based predicate, we implement the value-based predicate NFA with a hash table. The reason behind this approach is that automata based on a hash table can reduce the insertion/deletion time of the NFA state. In order to implement this approach, each state has the following data structure: (1) a variable that can store the state ID, (2) a small hash table that can store correct transition (*i.e.*, *transition hash table*), and (3) an accept state that has an *additional linked list* with (query ID, path ID, level) pairs as elements. A transition hash table for each state has a (input character, next state ID) pair. Here, the input character is the *key* of the hash table and is mapped with transition. Moreover, the next state ID is that which arrives when the current state ID is transitioned. There is no need for an additional data structure to represent the initial state with a self-loop transition in the hash table. Fig. 4 shows the result of the implementation of Fig. 1 (c) into hash tables. The number allocated to each hash table indicates the state ID, and the thick rectangle denotes the accept state. Each accept state has a (query ID, path ID, level) pair.

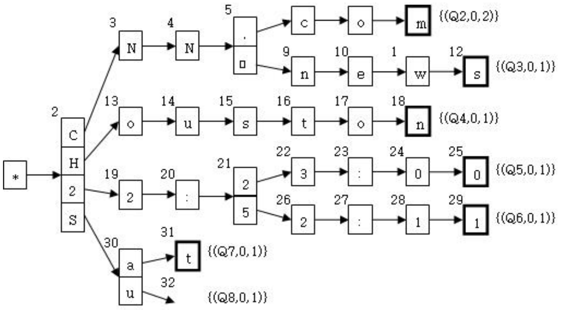


Fig. 4 Hash-based implementation of value-based predicate NFA

Pfilter and ACfilter can be defined based on the representation of the value-based predicate, the automata construction method, and its implementation as explained above.

Definition 1 (Pfilter) Pfilter is defined as the document-centric XML filtering technique implementing the predicate set as a hash-based single NFA form using the insertion/deletion algorithm of Fig. 3 after removing the value-based predicate processing part of Yfilter for effective filtering of the value-based predicate. Pfilter converts the Aho-Corasick dictionary matching tree [5] from DFA to NFA.

Definition 2 (ACfilter) ACfilter is defined as the document-centric XML filtering technique that replaces the value-based predicate processing part with the Aho-Corasick dictionary matching tree and implements it into the linked list-base for rational comparison between Pfilter and Yfilter, which does not permit the special matching character '%'.

4) Execution of the value-based predicate NFA

Let us examine the execution of the value-based predicate NFA machine implemented using a hash table. The value-based predicate NFA considers a single character as an event and executes the predicate with the *event-driven* method. When the XML document to filter arrives at the Pfilter, the content of the element can be obtained with the *Characters()* method of the SAX parser [15]. The element is read sequentially from the first character to the last while generating an event for each character. The generated event is sent to the handler and generates transition in the value-based predicate NFA. For reference, the ACfilter can apply the NFA execution algorithm almost without any modification. Therefore, the execution algorithm of the ACfilter will not be discussed anymore.

a) Start Element Content Handler

When the element content starts, the algorithm prepares to begin transition in the initial state of the value-based predicate NFA. If the number of self-loop transitions is 0, the initial state is stored in the objective state linked list.

b) Character Handler

This handler is activated every time a character is read from the element content. Value-based predicate NFA execution performs transition when there is matching for each state currently activated. For a more detailed explanation, let us assume that the number of self-loop transitions thus far is *SL*, and the number of remaining characters not inputted to the operand is *RC*. Then the following three procedures are carried out under each activated state.

(1) Search the current state (*i.e.*, hash table) using the character key input to the value-based predicate NFA. If a next state ID that corresponds to the character key exists, we insert the state ID into the objective state linked list and reduce the *RC*.

(2) If a next state ID does not exist in the hash table, the current state ID is removed from the objective state linked list. However, if the current state ID is the initial state, it is not removed from the objective linked list, and *SL* is incremented.

(3) After transition, if the state is the accept state, we refer to *SL* and *RC*, and insert the (query ID, path ID, level) pair into an adequate place in one of the four following hash sets.

- If $SL=0 \wedge RC=0$, insert the (query ID, path ID, level) pair into the *equal hash set* (EHS).
- If $SL>0 \wedge RC=0$, insert the (query ID, path ID, level) pair into the *pure prefix hash set* (PPHS).
- If $SL=0 \wedge RC>0$, insert the (query ID, path ID, level) pair into the *pure suffix hash set* (PSHS).
- If $SL>0 \wedge RC>0$, insert the (query ID, path ID, level) pair into the *pure prefix-suffix hash set* (PPSHS).

c) End Element Content Handler

When each character of the element content is inputted into the value-based predicate NFA, the four hash sets will have the (query ID, path ID, level) pair which considers the special matching character '%' for the current element content. The (query ID, path ID, level, matching character info) set can be recognized from the structural NFA, and the value-based predicate processing can be completed according to the following procedures.

(1) For operands without the prefix, suffix, and prefix-suffix '%' (*i.e.*, the matching character info is "N"), we check if there is a (query ID, path ID, level) pair in the EHS. If there is, the pair satisfies the value-based predicate.

(2) For operands with the prefix '%' (*i.e.*, the matching character info is "P"), we check if there is a (query ID, path ID, level) pair in the EHS and PPHS. If there is, the pair satisfies the value-based predicate.

(3) For operands with suffix '%' (*i.e.*, the matching character info is "S"), we check if there is a (query ID, path ID, level) pair in the EHS and PSHS. If there is, the pair satisfies the value-based predicate.

(4) For operands with prefix-suffix '%' (*i.e.*, the matching character info is "PS"), we check if there is a (query ID, path ID, level) pair in all hash sets. If there is, the pair satisfies the value-based predicate.

After checking value-based predicate matching according to the above procedures, if the remaining structure matching is satisfied (e.g., nested path matching), the query matches the current XML document.

Fig. 5 shows an example of a value-based predicate NFA execution. Each rectangle in Fig. 5 (b) stores the (active state ID, the number of self-loop transitions) pair for the character input. The figure confirms that the execution of the value-based predicate NFA does not use a failure function necessary for executing the Aho-Corasick dictionary matching tree.

IV. Experiment and Evaluation

A. Experiment Setup

Pfilter and ACfilter are implemented using Java. In the case of Yfilter, there is an open source code[1]. All experiments for this paper were conducted using Java virtual machine 1.5 running on PentiumIV 3.2GHz processor, 1GB of memory, and Windows Server 2003. In order to avoid the influence of the garbage collector in the Java virtual machine, a new process was used for each experiment, and the results were evaluated.

1) Workload Generation

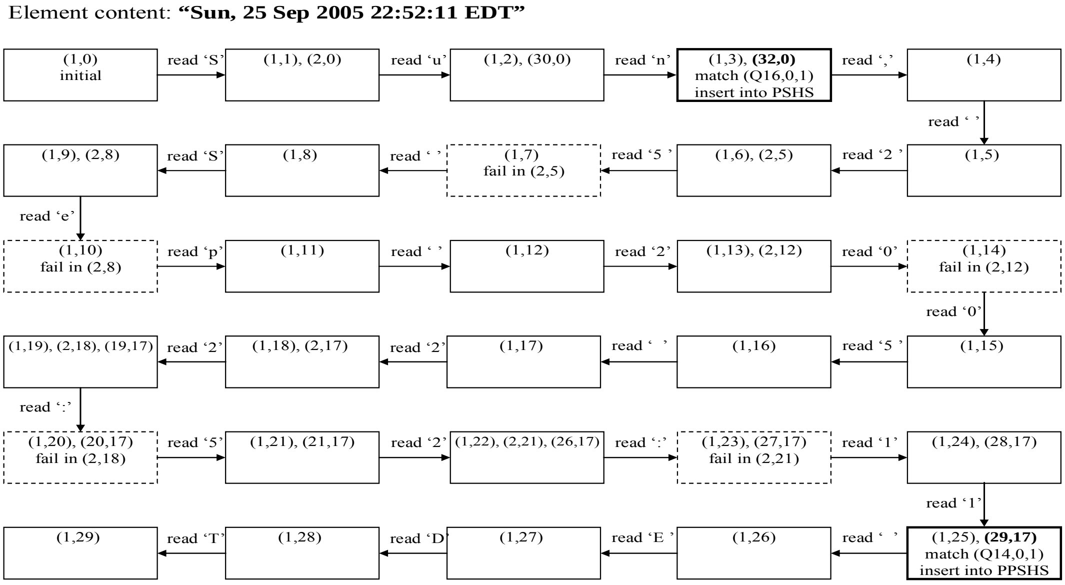
If the Yfilter query generator was used without modification, the value-based predicate path expression according to the

[1] http://yfilter.cs.berkeley.edu

parameters will be generated. However, there is the problem of the operand not being generated correctly according to the parameters, and thus the Yfilter query generator was modified. One of the notable points in the modification of the Yfilter query generator is that only 1,000 of the 3,000 words become operands to the value-based predicates.

<rss><pubDate>Sun, 25 Sep 2005 22:52:11 EDT</pubDate></rss>

(a) An XML fragment



(b) Value-based predicate matching part

EHS	PPHS	PSHS	PPSHS
no pair!	no pair!	(Q16,0,1)	(Q14,0,1)

(c) Four type of hash set after (b)

Because of (Q13,0,1,**PS**), search EHS, PPHS, PSHS, and PPSHS → no match!
Because of (Q14,0,1,**PS**), search EHS, PPHS, PSHS, and PPSHS → **match!**
Because of (Q15,0,1,**S**), search EHS and PSHS → no match!
Because of (Q16,0,1,**S**), search EHS and PSHS → **match!**

(d) Query matching using four hash sets

Fig. 6 An example of value-based predicate NFA execution

In order to perform a more accurate generation of the value-based predicate, it must be generated by reflecting the skewness of the characters that occurred in each operand. Such requirements are also applied to XML element contents synthesized for matching the queries and XML. Our experiment used the Zipf distribution [16]. The number of paths for queries and the number of predicates for each path were fixed in our experiment to 1, which also fixed the probability of '*' and '//' appearing in each location step to 0.2. Although the number of paths for queries was fixed at 1, Pfilter and ACfilter are supporting the execution of nested path queries. Table 1 lists the workload parameters for synthesizing XML and value-based predicates.

To briefly explain *P*, *Prefix*, *Suffix,* and *PS* among the parameters in Table I, *P* denotes the probability of '%' appearing in the value-based predicate operand regardless of the prefix '%', suffix '%', or prefix-suffix '%'. *Prefix* (*Suffix*, *PS*) is the probability of a value-based predicate operand with '%' appearing to be a prefix '%' (suffix '%', prefix-suffix '%'), respectively. Therefore, *Prefix+Suffix+PS=1* is always satisfied.

The experimental result provided hereafter is the average time it takes to process 200 XMLs. An XML document is read one by one from the disk, and the execution result of the XML with respect to a value-based predicate becomes the value-based predicate ID that matches the corresponding XML.

TABLE I
WORKLOAD PARAMETERS FOR CONSTRUCTING QUERIES AND DOCUMENT

Parameter	Range	Description
Q	1,000 ~ 500,000	Number of distinct queries
ZC	0 ~ 2	Skewness of characters in operand
LO	2 ~ 64	Average length of operand in all value-based predicates
LC	2 ~ 1,000	Average length of element contents in all XML documents
P	0 ~ 1	Probability of '%' occurring at operands of all value-based predicates
Prefix	0 ~ 1	Probability that the operand with '%' is prefix '%'
Suffix	0 ~ 1	Probability that the operand with '%' is suffix '%'
PS	0 ~ 1	Probability that the operand with '%' is prefix-suffix '%'

2) Metric

The metric for our experiments is the *multi-query processing time* (MQPT), which is defined as follows.

Definition 3 (MQPT) Let us denote the time of a document-centric XML input into the filtering system as t_{start}, the time of SAX parsing the completion of input XML as $t_{parsing}$, and the time of transition completion of structure NFA and value-based predicate NFA using SAX parsing event and finding the final matching query as t_{end}. Then MQPT can be defined as t_{end}-$t_{parsing}$.

In order to correctly understand MQPT, one must note that unlike the definition of the filtering time t_{end}-t_{start}, MQPT does not include the time it takes for parsing the XML.

B. Experiment 1: Maintenance Cost of Pfilter and ACfilter

This experiment examines the insertion cost of the operand in the value-based predicate automata of Pfilter and ACfilter. Due to spatial restrictions, a discussion about the deletion cost, which renders similar results as the insertion cost, has been omitted.

There is no need to generate the synthesized XML to search for matching in order to assess the maintenance costs of Pfilter and ACfilter. Therefore, the related parameter LC's change is not considered for this experiment. Moreover, the special matching character '%' has no role in inserting/deleting operands into/from value-based predicate automata, respectively. Since it is only related in terms of which hash set the (query ID, path ID, level) pair should be inserted into when executing a value-based predicate, P=0 is fixed. Moreover, an increase of the parameter LO in a value-based predicate is fixed at 8 since it generally increases the insertion/deletion cost for the two systems. Since the number of paths is fixed at 1 for all synthesized value-based predicates, the parameter Q is identical to the number of operands. Therefore, the results of Experiment 1 signify the time required for inserting/deleting 4,000 queries or 4,000 operands.

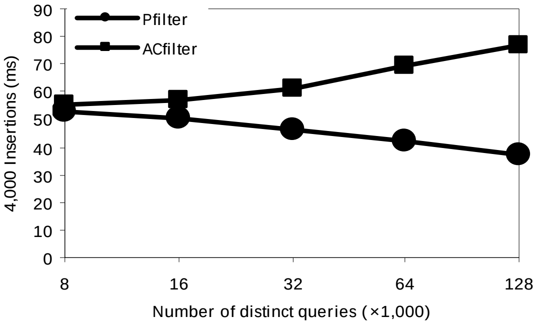


Fig. 7 Cost of inserting 4,000 queries (varying Q, ZC=0)

Fig. 7 shows the time required for inserting 4,000 value-based predicates. As the number of the queries registered in Pfilter increases, the probability of the character key being shared also increases, and the number of insertions of the (character key, next state ID) pair into the hash table decreases. However, ACfilter needs to search the linked list to prevent redundancy in the linked list and to check whether or not there is a string to insert into the linked list. Therefore, an increase in the number of queries increases the number of operands, the length of each *state linked list* in the value-based predicate DFA, the search time, and even the insertion time.

C. Experiment 2: Efficiency and Scalability of Pfilter and ACfilter

In order to focus on the execution of the value-based predicate operand, the number of paths to the parameter Q=50,000> If the special matching character ' %' exists, matching between the value-based predicate and XML can occur when LO≤ LC, so the default values were set as LO=8 and LC=100, which were adequately modified when LO and LC became independent variables.

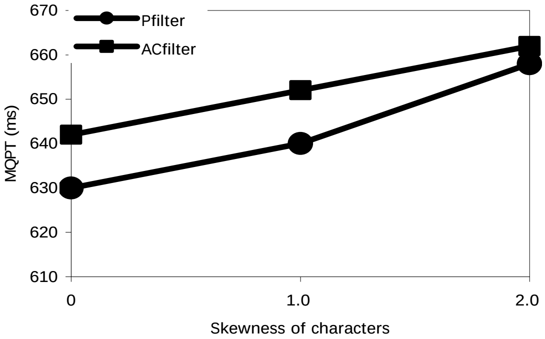


Fig. 8 Varying skewness of characters (LO=8, LC=100, P=0.7, Prefix=Suffix=PS=1/3)

Fig. 8 shows that the MQPT of the two filtering systems as skewness of the characters is varied in the operand. As ZC increases, the difference between the MQPTs of Pfilter and ACfilter is reduced. However, the MQPTs of the two filtering techniques display an increasing tendency. It is due to the fact that the number of the matching value-based predicates also increases, as well as the number of insertions of the (query ID, path ID, level) pairs into the four hash sets.

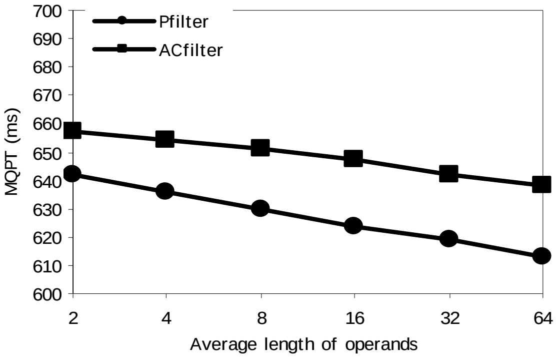


Fig. 9 Varying average length of operands (ZC=0, LC=100, P=0.7, Prefix=Suffix=PS=1/3)

Fig. 9 shows the MQPTs of the two filtering systems as the average operand length increases. If LO is short, then the number of matching between value-based predicates and XML increases. In turn, the number of insertions of the (query ID, path ID, level) pairs in the four hash sets increases, which heightens the MQPT.

Fig. 10 shows the MQPTs of the two techniques as the average element length of synthesized XML increases. An increase in LC escalates the input to the value-based predicate automata, which increases the number of transition tests as well as the MQPT. Since Pfilter using the hash table has constant access time to each state's hash table, its MQPT improves as compared to ACfilter using a linked list. Moreover, it was verified in this experiment that LC is the parameter with the most significant impact on the MQPT of the Pfilter technique.

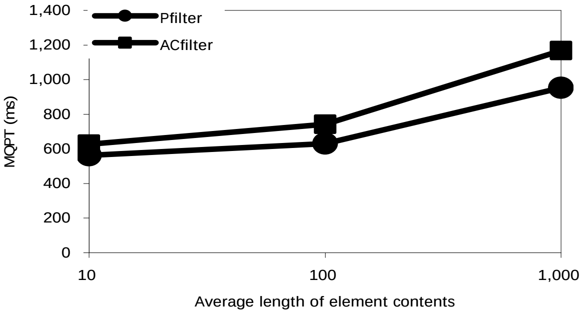


Fig. 10 Varying average length of element contents (ZC=0, LO=8, P=0.7, Prefix=Suffix=PS=1/3)

V. Conclusion

We proposed Pfilter, which is adequate for the document-centric XML. Since Pfilter shares common prefix characters in order to effectively process the value-based predicate, this approach can have many advantages in the real world in terms of value-based predicate matching. In the case of value-based predicates used in most article searches, there are words that are searched frequently at specific times. As the number of these words registered in the filtering system increases, the probability of the common prefix character sharing becomes higher.

Pfilter has been implemented by separating structure matching and value-based predicate matching, and the structure matching technique can be selectively deployed. This signifies that since most structure matching techniques have different characteristics, they can be replaced with adequate structure matching techniques according to the circumstances.

The Aho-Corasick dictionary matching tree algorithm is to solve the problem of the static dictionary matching, but not to solve the problem of the dynamic dictionary matching which should consider changing the operand set. Future studies will be focused on dynamic environments, in which subscribers are likely to join and leave, and that the data interests of existing subscribers may also evolve over time. We will implement a system which reacts quickly to query changes without adversely affecting the processing of incoming XML documents.

References

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 Second Edition W3C Recommendation. Technical Report REC-xml-200010006, World Wide Web Consortium.

[2] J. Kamps, M. Marx, M. de Rijke, and B. Sigurbjörnsson, "Best-match Query form Document-centric XML," In Proc. Int. Workshop on the Web and Databases, pp. 55-60, 2004.

[3] J. Clark, and S. DeRose. XML Path Language (XPath) Version 1.0 W3C Recommendation. Technical Report REC-xpath-19991116, World Wide Web Consortium.

[4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language W3C Working Draft. Technical Report WD-xquery-20050404, World Wide Web Consortium.

[5] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," Communications of the ACM, Vol. 18, Issue 6, pp. 333-340, 1975.

[6] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path Sharing and Predicate Evaluation for High-Performance XML Filtering," ACM Trans. Database Systems, Vol. 28, Issue 4, pp. 467-516, 2003.

[7] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata and Stream Indexes," ACM Trans. Databases Systems, Vol. 29, Issue 4, pp. 752-788, 2004.

[8] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, "Navigation- vs. Index-based XML Multi-query Processing," In Proc. IEEE Int. Conf. Data Engineering, pp. 139-150, 2003.

[9] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," In Proc. IEEE Int. Conf. Data Engineering, pp. 235, 2002.

[10] V. Josifovski, M. Fontoura, and A. Barta, "Querying XML Streams," Int. J. Very Large Data Bases, Vol. 14, Issue 2, pp. 197-210, 2005.

[11] J. Kwon, P. Rao, B. Moon, and S. Lee, "FiST: Scalable XML Document Filtering by Sequencing Twig Patterns," In Pro. Int. Conf. Very Large Data Bases, pp. 294-315, 2005.

[12] A. K. Gupta and D. Suciu, "Stream Processing of XPath Queries with Predicates," In Proc. ACM SIGMOD Int. Conf. Management of Data, pp. 419-430, 2003.

[13] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki, "Implementing A Scalable XML Publish/Subscribe System Using Relational Database Systems," In Proc. ACM SIGMOD Int. Conf. Management of Data, pp. 479-490, 2004.

[14] F. Peng, and S. S. Chawathe, "XSQ: A Streaming XPath Engine," ACM Trans. Databases Systems, Vol. 30, Issue 2, pp. 577-623, 2005.

[15] D. Megginson. SAX: A Free API for Event-based XML Parsing. Available: http://www.saxproject.org, 2005.

[16] C. D. Manning and H. Schütze. Foundations of Statistical Natural Language Processing. The MIT Press, 1999.

Changwoo Byun received the B.S. and M.S. degrees in the Department of Computer Science from Sogang University, Seoul, Korea, in 1999 and 2001, respectively. Since 2001, he has been studying for a Ph.D. at the Department of Computer Science of Sogang University. His areas of research include role-based access control model, access control for distributed systems, access control for XML data, Dynamic access control, and XML Data Stream Processing.

Kyounghan Lee received the B.S. and M.S. degrees in the Department of Computer Science from Sogang University, Seoul, Korea, in 2001 and 2006, respectively. Since 2006, he has been working at Samsung Electronics. His areas of research include XML Data Stream Processing, XML Indexing, and Embedded software development.

Seog Park is a Professor of Computer Science at Sogang University. He received the B.S degree in Computer Science from Seoul National University in 1978, the M.S. and the Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1980 and 1983, respectively.

Since 1983, he has been working in the Department of Computer Science of the College of Engineering, Sogang University. His major research areas are database security, real-time systems, data warehouse, digital library, multimedia database systems, role-based access control and Web database. Dr. Park is a member of the IEEE Computer Society, ACM and the Korea Information Science Society. Also, he has been a member of Database Systems for Advanced Application (DASFAA) steering committee since 1999.