Efficient preprocessing of XML queries using structured signatures

Yon Dohn Chung a, Jong Wook Kim b, Myoung Ho Kim b,*

a Department of Computer Multimedia Engineering, Dongguk University, 26, 3 ga, Pil-dong, Chung-gu, Seoul, 100-715, Republic of Korea
b Division of Computer Science, KAIST 373-1, Kusung-dong, Yusung-gu, Daejeon, 305-701, Republic of Korea

Received 8 May 2002; received in revised form 19 March 2003
Communicated by F.Y.L. Chin

Abstract

The paper proposes a preprocessing scheme for efficient processing of XML queries in XML-based information retrieval systems. For the preprocessing, we use a signature-based approach. In the conventional (flat document-based) information retrieval systems, user queries consist of keywords and boolean operators, and thus signatures are structured in a flat manner. However, in XML-based information retrieval systems, the user queries have the form of path queries. Therefore, the flat signature cannot be effective for XML documents. In the paper, we propose two structured signature methods for XML documents. Through experiments, we evaluate the performance of the proposed methods.

Keywords: Signature; Query processing; XML; Databases

1. Introduction

The XML (eXtensible Markup Language) has recently emerged as a standard for information representation and exchange on the Internet [1]. An XML document consists of data surrounded by user defined tags, and is represented by a tree structure, called the DOM tree [2]. Fig. 1 shows an example of an XML document and its DOM tree representation.

For information retrieval of XML documents, many query languages have been proposed: Lorel [3], XML-QL [4], XQuery [5], and so on. A common feature of these languages is that they contain path queries, which are represented by regular expressions [3,6]. For example, in Fig. 1, a path query represented by a regular expression “invoice/_/[address==’New Y ork’]” retrieves the records of sellers or buyers who live in ‘New York’. For readers’ convenience, we give a brief description on the terms of regular expressions in Table 1.

There are many approaches for storing XML documents: using relational databases or file-system based specialized data managers [6,7]. In the paper, we assume an XML-based information system which uses a file-system based data manager. While most previous papers considered the case of a single large XML document, this paper focuses on the case of many (small) XML documents, e.g., book records in a library.

Fig. 2 shows the concept of preprocessing of XML queries. There are many XML documents in the information retrieval system. We extract abstract

* Corresponding author.
E-mail addresses: ydchung@dgu.edu (Y.D. Chung), jwkim@dbserver.kaist.ac.kr (J.W. Kim), mhkim@dbserver.kaist.ac.kr (M.H. Kim).

0020-0190/03 - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0020-0190(03)00308-9
information (in the form of signatures [8]) from XML documents, and maintain the abstract information in main memory. When a user query (i.e., a path query) is submitted to the system, we preprocess the query with the abstract information. And, we deliver the identifiers of the documents which pass through the preprocessing module to the main query processing module. Note that the abstract information is small enough to be maintained in main memory, and the XML documents usually reside in disks. Therefore, by reducing the number of documents for (main) query processing through the preprocessing, we can effectively improve the overall performance of XML query processing.

The rest of the paper is organized as follows. In Section 2, we describe some background information on conventional signature methods. We also show the conventional signature structures are not effective for XML documents. Section 3 describes our proposed signature structures and algorithms. The performance of the proposed preprocessing method is experimentally evaluated in Section 4. And, we conclude the paper in Section 5.

2. Background and motivation

There have been many signature methods for retrieving documents matched with the user query string [8,9]. The signature is a hashed bit-stream of a string. By OR-ing (in case of the superimposed coding method) the signatures of strings of a document, we
can generate the signature of the document. For example, when using a hash function $H$ in Table 2, the signature of the document in Fig. 1 becomes “011101101” by OR-ing all its hashed bit-streams.

When a user query is submitted, we generate a signature of the query string, and compare it with the document’s signature. If the query signature is not covered by the document signature, we can say the document is not included in the query’s answer. In this way, we can filter the documents for the given user query with signatures.

In conventional text-based information retrieval systems, the user query consists of strings (such as keywords) and boolean operators, and the text documents also consist of strings in a flat manner. However, in XML-based systems, the structures of the user query and documents are not flat, that is, the query is a path query and the document has a tree structure. For example, in conventional text-based systems, a user query finds documents that contain ‘invoice’ and ‘seller’. But, in XML-based systems, a user query (i.e., a path query) ‘invoice/seller’ finds documents which contain ‘invoice’ and ‘seller’ nodes connected in a parent-child relationship. Therefore, the conventional flat signature structures are not effective for XML documents.

### 3. The proposed method

In this paper we propose two kinds of signatures: the horizontal signature (HS) and the vertical signature (VS). After explaining the signature structures, we describe the preprocessing methods. Here, we assume an XML document is represented as a DOM tree, and there is a hash function $H$ for generating hashed streams of elements and attributes of XML documents. For convenience of description, we assume the DOM tree is a full $n$-ary tree. (In fact, the proposed method does not require this assumption.)

Firstly we define some notations for signature description. Let $h$ be the height of the DOM tree. Then, the root node is the 1st-level node and a leaf node is the $h$th-level one. In the DOM tree of a document $D$, $N_D(a, b)$ denotes the $b$th node in the $a$th-level. Thus, the right-most leaf node of the tree is represented as $N_D(h, n^{(h-1)})$. And, $S_D(a, b)$ denotes the signature of $N_D(a, b)$.

#### 3.1. Horizontal and vertical signatures (HS and VS)

The horizontal signature (HS) is the OR-ed signature of the nodes that reside in the same level of the DOM tree. And, the horizontal signature path (HSP) is the path of horizontal signatures arranged according to the level of the DOM tree.

**Definition 1.** The $i$th level horizontal signature of an XML document $D$, denoted by $HS_D(i)$, is defined as follows: (Here, ’|’ denotes the bit-wise OR operator.)

$$HS_D(i) = S_D(i, 1)\vert S_D(i, 2)\vert \cdots \vert S_D(i, n^{(i-1)})$$

And, the horizontal signature path of an XML document $D$, denoted by $HSP_D$, is “$HS_D(1)\vert HS_D(2)\vert \cdots \vert HS_D(h)$”.

The vertical signature (VS) is the OR-ed signature of the nodes that reside in a path (from the root to a leaf) in the DOM tree. And, the vertical signature set (VSS) is the set of vertical signatures of a DOM tree.

**Definition 2.** The vertical signature of the $k$th path, denoted by $VS_D(k)$ (i.e., from the root node to $N_D(h, k)$) of a document $D$ is defined as follows: (Here, $w_1$ and $w_2$ denote the position numbers of the nodes that reside in the $k$th path.)

$$VS_D(k) = S_D(1, w_1)\vert S_D(2, w_2)\vert \cdots \vert S_D(h, k)$$

And, the vertical signature set of a document $D$, denoted by $VSS_D$, is \{VS_D(k) | k = 1, \ldots, n^{(h-1)}\}.

Fig. 3 shows an example of HSP and VSS based on Fig. 1. When a user query is submitted, we convert the user query into the query signature based on the hash function $H$. And, we compare the query signature with the documents’ signatures (HSP and VSS). In this...
signature comparison (i.e., we call it ‘preprocessing’ in the paper), we can filter out many XML documents that do not match the user query. Now, we explain the preprocessing algorithms.

3.2. The algorithms

The path query (submitted by a user) is a string represented by a regular expression. Let us define \( PQ_i \) as the \( i \)th substring separated by ‘/’ in a path query \( PQ \). Let \( i \)' denote the number of substrings of \( PQ \). Then, \( PQ \) is represented by “\( PQ = PQ_1/PQ_2/\cdots/PQ_t \)”. For example, when given a \( PQ \) “invoice/_*/name”, \( PQ_1 \) is ‘invoice’, \( PQ_2 \) is ‘/_*’ and \( PQ_3 \) is ‘name’. We define two functions, get_first() and move_next(), for string management. get_first(\( PQ \)) returns the first substring \( PQ_1 \), and move_next(\( PQ \)) returns “\( PQ_2/\cdots/PQ_t \)”.

Fig. 4 describes the preprocessing algorithm that uses VSS.\(^1\) In the algorithm, we first convert the path query (\( PQ \)) into the query signature (Q-Sig), where the query signature is defined as “\( H(PQ_1)/(PQ_2)/\cdots/H(PQ_t) \)” using the given hash function \( H \). And we compare each VS in VSS with Q-Sig.

The preprocessing using HSP is done as follows:

1. convert the user path query into a query signature path, and
2. compare the query signature path with the HSP of XML documents.

(Note that we use query signature paths in the HSP-algorithm while we use query signatures in the VSS-algorithm.) The query signature path is a sequence of hashed streams of the path query, where the regular expression symbols in the path query remains in the query signature path. For example, path query “invoice/_*/name” is converted into a query signature path “01000100/001101001/000101100”. Since ‘/’ denotes ‘any node’, we hash it into ‘111...1’. The comparison of a query signature path and HSP is basically the non-deterministic finite automata (NFA). Fig. 5 shows an NFA of the above query signature path “01000100/111111111*/000001000” and an HSP “01000100/001101001/000101100”.

After filtering out the not-qualified documents through the two preprocessing methods, we do the main query processing with the remaining documents. In the main query processing, we can use various indexing and query processing methods developed in the past [6,10].

\(^1\) We assume the path query does not contain the ‘or’ operator (\( \lor \)). When a query with ‘or’ operators is submitted, we separate the query into multiple queries without ‘or’ operator.
function VSS-algorithm(VSS, PQ)
/* INPUT: VSS (the set of VS for a document), PQ (the string of a path query) */
/* OUTPUT: TRUE (matched) or FALSE (not matched) */
{
remove ‘?’, ‘+’, and ‘∗’ characters from PQ;
Q-Sig = ‘000…0’;
while (PQ != NULL) {
    String = get_first(PQ);
    if (String starts with ‘_’) { /* do nothing */}
    else {Q-Sig = Q-Sig | H(String); /* bit-wise OR operation */}
    PQ = move_next(PQ);
}
foreach VS in VSS {
    if ((VS & Q-Sig) == Q-Sig){ /* bit-wise AND operation */
        return TRUE;
        /* no more check */
    }
}
return FALSE;
}

Fig. 4. The preprocessing algorithm using VSS.

Fig. 5. Using NFA for preprocessing with HSP.

4. Performance experiments

4.1. Experiments

The experiments were conducted on a PC with Pentium III CPU 600 MHz and 256 MB main memory running Linux (mandrake 8.1). We implemented the proposed methods and the conventional method using g++, and used Xerces toolkit (version 1.5.2) [11] for parsing XML documents. We created XML documents using the DocBook [12] DTD and NITF (News Industry Text Format) [13] DTD with IBM’s XML generator tool [14]. The number of documents used for experiments is 2000 (1000 for DocBook DTD and 1000 for NITF DTD). And, we set the value of maximum depth (max_depth) as 10 and that of maximum number of children (max_repeat) as 5. Note that all documents are generated with a uniform distribution of element names as provided by the IBM’s XML generator. The average size of an XML document is about 24 KB with its depth of 9 and the average number of nodes of a document is 75.

The experimented methods were:

1. VSS only,
2. HSP only,
3. VSS + HSP, and
4. the conventional signature method (C-Sig).

We measured the filtering ratio and the memory usage of each preprocessing method. We used two signature sizes: 32-bit signature and 64-bit signature.

4.2. Results

In Tables 3 and 4, we show the filtering ratios of preprocessing methods. The filtering ratio is defined as “the number of documents that are filtered out / the total number of documents × 100%”. Thus, the filtering ratio ‘100’ means that all documents are filtered out.

---

In the XPath syntax, it is “set//title/*/procedure”.

---
Table 3
The filtering ratios of preprocessing methods (%)–(32-bit signature)

<table>
<thead>
<tr>
<th>No</th>
<th>Query</th>
<th>HSP</th>
<th>VSS</th>
<th>HSP + VSS</th>
<th>C-Sig</th>
<th>Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/set/book/title</td>
<td>27</td>
<td>15</td>
<td>31</td>
<td>4</td>
<td>54</td>
</tr>
<tr>
<td>2</td>
<td>/set/book/._*/title</td>
<td>6</td>
<td>15</td>
<td>15</td>
<td>4</td>
<td>71</td>
</tr>
<tr>
<td>3</td>
<td>/set/book/._title</td>
<td>43</td>
<td>15</td>
<td>46</td>
<td>4</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>/set/book/_/title</td>
<td>39</td>
<td>15</td>
<td>44</td>
<td>4</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>/set/book/body</td>
<td>99</td>
<td>69</td>
<td>100</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>/set/book/_*/body</td>
<td>99</td>
<td>69</td>
<td>100</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>/set/book/_/body</td>
<td>99</td>
<td>69</td>
<td>100</td>
<td>58</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>/set/*title/procedure</td>
<td>62</td>
<td>79</td>
<td>82</td>
<td>34</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>/set/.*title/_/procedure</td>
<td>71</td>
<td>79</td>
<td>82</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>/set/*title/_/procedure</td>
<td>70</td>
<td>79</td>
<td>82</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>/set/<em>title/</em>/procedure</td>
<td>63</td>
<td>79</td>
<td>80</td>
<td>34</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>_*/title/book</td>
<td>63</td>
<td>15</td>
<td>64</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><em>*/title/</em>*/book</td>
<td>43</td>
<td>15</td>
<td>45</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

(DocBook DTD documents)

Table 4
The filtering ratios of preprocessing methods (%)–(64-bit signature)

<table>
<thead>
<tr>
<th>No</th>
<th>Query</th>
<th>HSP</th>
<th>VSS</th>
<th>HSP + VSS</th>
<th>C-Sig</th>
<th>Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/set/book/title</td>
<td>28</td>
<td>24</td>
<td>38</td>
<td>7</td>
<td>54</td>
</tr>
<tr>
<td>2</td>
<td>/set/book/._*/title</td>
<td>11</td>
<td>24</td>
<td>24</td>
<td>7</td>
<td>71</td>
</tr>
<tr>
<td>3</td>
<td>/set/book/._title</td>
<td>53</td>
<td>24</td>
<td>53</td>
<td>7</td>
<td>41</td>
</tr>
<tr>
<td>4</td>
<td>/set/book/_/title</td>
<td>64</td>
<td>24</td>
<td>65</td>
<td>7</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>/set/book/body</td>
<td>99</td>
<td>87</td>
<td>100</td>
<td>59</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>/set/book/_*/body</td>
<td>99</td>
<td>87</td>
<td>100</td>
<td>59</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>/set/book/_/body</td>
<td>99</td>
<td>87</td>
<td>100</td>
<td>59</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>/set/*title/procedure</td>
<td>77</td>
<td>84</td>
<td>88</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>/set/.*title/_/procedure</td>
<td>79</td>
<td>84</td>
<td>90</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>/set/*title/_/procedure</td>
<td>85</td>
<td>84</td>
<td>93</td>
<td>60</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>/set/<em>title/</em>/procedure</td>
<td>61</td>
<td>84</td>
<td>84</td>
<td>60</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>_*/title/book</td>
<td>81</td>
<td>24</td>
<td>82</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td><em>*/title/</em>*/book</td>
<td>65</td>
<td>24</td>
<td>67</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(DocBook DTD documents)

Table 4
The filtering ratios of preprocessing methods (%)–(64-bit signature)

<table>
<thead>
<tr>
<th>No</th>
<th>Query</th>
<th>HSP</th>
<th>VSS</th>
<th>HSP + VSS</th>
<th>C-Sig</th>
<th>Selectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>_*/li/media</td>
<td>55</td>
<td>60</td>
<td>62</td>
<td>47</td>
<td>25</td>
</tr>
<tr>
<td>15</td>
<td>_*/li/.*media</td>
<td>53</td>
<td>60</td>
<td>60</td>
<td>47</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td><em>*/li/</em>/media</td>
<td>82</td>
<td>60</td>
<td>82</td>
<td>47</td>
<td>18</td>
</tr>
</tbody>
</table>

(NITF DTD documents)
Table 5

The memory usage

<table>
<thead>
<tr>
<th></th>
<th>DocBook DTD</th>
<th>HSP</th>
<th>VSS</th>
<th>HSP + VSS</th>
<th>C-Sig</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit</td>
<td>3.5 KB</td>
<td>8.5 KB</td>
<td>12 KB</td>
<td>400 Bytes</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td>64-bit</td>
<td>7 KB</td>
<td>17 KB</td>
<td>24 KB</td>
<td>800 Bytes</td>
</tr>
<tr>
<td>32-bit</td>
<td>3.5 KB</td>
<td>11.5 KB</td>
<td>15 KB</td>
<td>400 Bytes</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td>64-bit</td>
<td>7 KB</td>
<td>23 KB</td>
<td>30 KB</td>
<td>800 Bytes</td>
</tr>
</tbody>
</table>

filtered out during the preprocessing phase, and thus no document remains for the main query processing phase. The column selectivity shows the ratio (%) of documents that are actually qualified by the query in the main query processor, that is, the ‘100-selectivity’ indicates the upper-bound of filtering ratio.

We used various query types: a class of queries that contain the nodes near to the root (Q1~Q4), a class of queries that contain some nodes which are not defined in the DTD (Q5~Q7), a class of queries with deep nodes (Q8~Q11), a class of queries whose node arrangement is different from the DTD (Q12~Q13), and so on.

Since the conventional signature and the vertical signature are made by OR-ing hashed streams from the root to a leaf, they show similar performance patterns. Also, they do not distinguish different orders of nodes that comprise the path query. For example, in C-Sig and VSS methods, the queries "/set/_*/title/procedure", "/set/_*/title/f/procedure" and "/set/_*/title/_/procedure" mutually have no difference. On the contrary, in the HSP method, the filtering ratios of the queries are different.

For all cases, the proposed methods perform better than the conventional method (C-Sig). For the queries with deep nodes (Q8~Q11), the VSS method is better than the HSP method. And, for the other queries, the HSP method performs better than the VSS method. In addition, we can see that the combination of two methods overcomes the drawbacks of each method.

Table 5 shows the amount of required memory space for the preprocessing methods, where 100 documents are used for signature construction. Necessarily, the proposed signature structures need more memory space than the conventional signature, and the memory usage is directly proportional to the number of documents. However, when considering the continuous decrease of memory cost, this overhead can be negligible.

5. Conclusion

In the paper, we proposed a signature-based preprocessing method for efficient processing of XML queries. We showed the conventional signature structures are not effective for path queries and XML documents, and proposed two signature structures (horizontal signature: H and vertical signature: VSS) which effectively represent the abstract information of XML documents. With the signature information we can efficiently preprocess (i.e., filter) the XML documents, and actually improve the query performance. Through experiments, we evaluated the performance of our preprocessing methods.

In the future, we will investigate the way to extend inverted list methods [15] for preprocessing of XML queries. The inverted list (or file) methods are widely used for text-based information retrieval, and in many cases they perform better than signature methods because they incur no false drop. However, the inverted list methods in their current forms cannot handle structural information in path queries and XML documents. Therefore, the current inverted list architectures need to be modified such that it can effectively represent structured relationships in XML queries and documents.

Acknowledgements

This work was supported by the Dongguk University Research Fund.
References


