An XML Publish/Subscribe Algorithm Implemented by Relational Operators*

Jiakui Zhao, Dongqing Yang, Jun Gao, and Tengjiao Wang

Database Lab, School of EECS, Peking University, Beijing 100871, China
{jkzhao,dqyang,gaojun,tjwang}@pku.edu.cn

Abstract. An XML publish/subscribe algorithm needs to store large numbers of XPath or XQuery subscriptions and match subscriptions with published XML documents. Since the number of the subscriptions may be very large, the performance and the scalability of the algorithm may be critical. The scalability of the method of constructing a large finite state automata or decision tree for all subscriptions is limited by amount of available physical memory. In this paper, we will introduce an XML publish/subscribe algorithm which is consisted of the publish algorithm, subscribe algorithm, and matching algorithm. The matching algorithm uses relational operators to match subscriptions with publications inside a relational database, so the scalability of the algorithm is no longer limited by amount of available physical memory. Experimental results show that the matching algorithm has very good performance and scalability.

1 Introduction

A publish/subscribe system receives messages from publishers and delivers to subscribers who require the messages. Earlier publish/subscribe systems are topic-based, in which subscribers subscribe topics and a published message is delivered to subscribers who subscribed the topic of the published message. Recent publish/subscribe systems are content-based, in which subscribers register "*rules*" and a published message is delivered to subscribers whose rules can be matched by the message. Content-based systems are more flexible and powerful than topic-based systems and are widely used in message oriented systems such as online auctions and financial information exchange. Almost all database and middleware vendors offer some publish/subscribe feathers in their software suits, in which users can use SQL-like language to express subscriptions and the messages may be something like relational tuples or dictionary data structures.

XML has become the standard data exchange format. There is an increasing demand for XML-based publish/subscribe systems which can support flexible document structures, and subscription rules should be expressed by powerful language such as XPath and XQuery; at the same time, the system must support high message throughput in case of millions of subscriptions. The main challenge

* Supported by NSF of China Grants 60473051 and 60503037, NSF of Beijing China Grant 4062018, and National High-Technical Project of China Grant 2006AA01Z230.

© Springer-Verlag Berlin Heidelberg 2007

of building such a system is how to match a published XML document with millions of XPath or XQuery subscriptions. Current matching strategies can be classified into two classes. The first class of strategies see subscriptions as filters, and subscribers are notified if a message passes through the filters. The filters are implemented by decision tree or finite state automata, and common computations between filters can be shared. This class of strategies maybe quite efficient if the decision tree or finite state automata can fit in memory; however, the scalability is limited by amount of available physical memory. The second class of strategies store subscriptions into a relational database, when a message is published, matching it with stored subscriptions by queries inside the database. The matching queries may be efficiently evaluated by using some appropriate indices, and the scalability is no longer limited by amount of available memory.

In this paper, we will introduce our publish/subscribe algorithm for implementing an XML-based publish/subscribe system by relational database, in which subscriptions are expressed by XPaths. To the best of our knowledge, our work is the first that implements the algorithm by non-recursive SQL language.

The rest of the paper is structured as follows. Section 2 describes previous works on publish/subscribe systems. Section 3 presents our publish/subscribe algorithm. Experiments are presented in Section 4, followed by our conclusions.

2 Previous Works

Some content-based publish/subscribe systems treat subscriptions as data; the matching between published messages and the subscriptions is implemented by join queries, and similar queries can share common computations. NiagaraCQ [1] uses signatures to group similar subscriptions, it uses a constant table extracted from the subscriptions along with a join to implement the matching algorithm. Commercial database vendors also use relational database engines to implement publish/subscribe systems [2], [3] whose scalability is not limited by available physical memory. However, most of the systems only handle tuple-like messages and rules are expressed by SQL. Some of these systems can handle XML messages, but the systems either use a wrapper to extract tuple-like data from XML messages [2] or use a very simple language to express the subscription rules [1].

Other content-based publish/subscribe systems treat subscriptions as filters over messages. In [4], a in-memory decision tree is built for all subscriptions. The system walks down the decision tree for each message and subscriptions at the leaf node are notified. Several XML-based publish/subscribe systems employee a finite state automata for all XPath subscriptions in the main memory. A sequence of SAX parsed events of an XML message are streamed through the finite state automata to match the XPath subscriptions. [5] first introduced how to match an XML message with many XPath subscriptions using an in-memory finite state automata, a later paper [6] introduced how to share states in finite state automata construction, and [7], [8] introduced how to build an index on sub-strings of XPath expressions and share common computations between common sub-strings. [9], [10] introduced how to use in-memory finite state automata

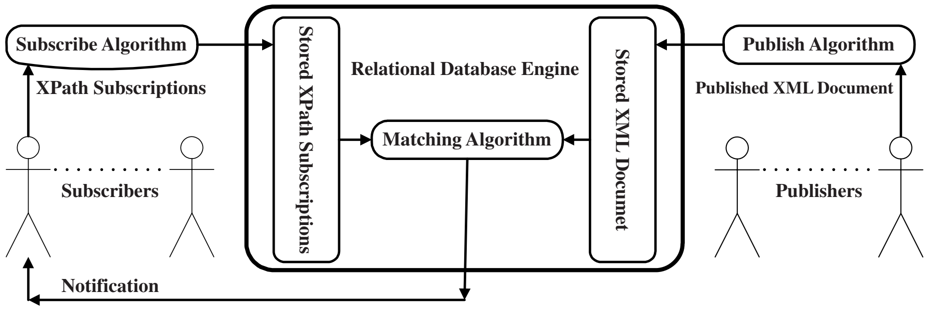


Fig. 1. Publish/subscribe framework

to evaluate XPath subscriptions over XML streams. [11] introduced a dynamic state construction approach, in which each state of a finite sate automata is constructed in memory when it is actually used; this approach is less attractive for long running systems. The main disadvantage of the in-memory approaches is the number of the subscriptions is limited by available physical memory, and insertions and deletions of subscriptions are difficult in the long-running systems.

[12] is the first that implements a long-running XML publish/subscribe system using a relational database, in which the subscriptions are expressed by an XPath subset called "*branching path expression*", and the number of the subscriptions is no longer limited by amount of available physical memory. Each XPath is rewritten as a conjunction of predicates, and each predicate contains one branch in the graph presentation of the XPath. For each published XML document, the branch path of each predicate is matched firstly, then a recursive SQL query is used to match all the predicates of each XPath. If all the predicates and branch points of an XPath are matched successfully, the owners of the XPath are notified. Our method is different from that proposed in [12] for we match XPaths with the XML document level-by-level using non-recursive SQL.

3 Publish/Subscribe Algorithm

Fig. 1 shows our publish/subscribe framework. Subscribers subscribe XML-style messages by XPaths; subscribe algorithm translates the XPath subscriptions into relational tuples and stores the tuples into a relational database. Publishers publish XML documents; publish algorithm translates an XML document into relational tuples and stores the tuples into a relational database. Matching algorithm matches stored XPath subscriptions with a stored XML document inside the relational database by relational operators to find out matched subscriptions and deliver the XML document to subscribers whose subscriptions are matched.

3.1 Publish Algorithm

We use the range-based static labeling scheme [13] to label the nodes of an XML tree. The scheme initializes a counter to one and carries out a depth-first

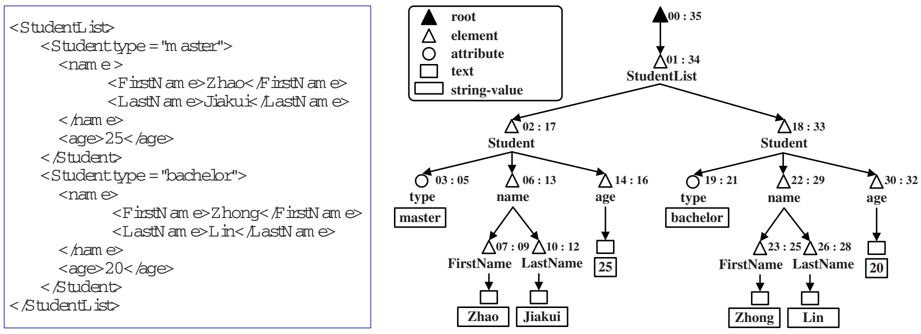
Table 1. The XML document shown in Fig. 2 stored in relational table *XMLFrame*

start_point	*end_point*	*depth*	*node*	*start_point*	*end_point*	*depth*	*node*
00	35	00	*-root-*	01	34	01	StudentList
02	17	02	Student	18	33	02	Student
03	05	03	@type	19	21	03	@type
06	13	03	name	22	29	03	name
14	16	03	age	30	32	03	age
07	09	04	FirstName	23	25	04	FirstName
10	12	04	LastName	26	28	04	LastName

Table 2. The XML document shown in Fig. 2 stored in relational table *XMLValue*

start_point	*node*	*value*	*start_point*	*node*	*value*
03	@type	master	19	@type	bachelor
14	age	25	30	age	20
07	FirstName	Zhao	23	FirstName	Zhong
10	LastName	Jiakui	26	LastName	Lin

traversal of the XML tree. If a node is seen for the first time, it is assigned the value of the counter as its "*start-point*", and the node is assigned the counter value as its "*end-point*" when it is encountered again. The counter is incremented by one each time its value is assigned to a node. Compared with dynamic labeling schemes such as the float number labeling scheme [14], the prefix labeling scheme [15], [16], and the prime number labeling scheme [17] which need not to re-label after insertions and deletions of nodes, the range-based labeling scheme has smaller storage requirements, so it is possible to use a fixed length representation to store the labels, and we can take advantage of the standard SQL92 data types and operations to determine the ancestor-descendant relationships



```
<StudentList>
    <Student type="master">
      <name>
            <FirstName>Zhao</FirstName>
            <LastName>Jiakui</LastName>
      </name>
      <age>25</age>
    </Student>
    <Student type="bachelor">
      <name>
             <FirstName>Zhong</FirstName>
             <LastName>Lin</LastName>
      </name>
      <age>20</age>
    </Student>
</StudentList>
```

Fig. 2. An XML document and the XML tree marked by "*start-point* : *end-point*"

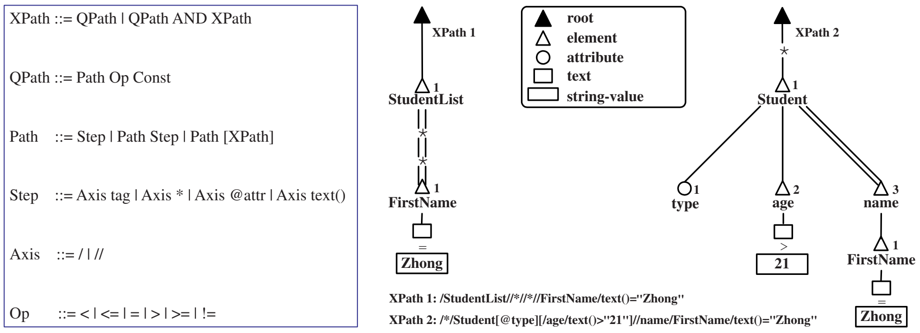
Table 3. The XPaths shown in Fig. 3 stored in relational table *XPathFrame*[*level*]

id	*high_num*	*low_num*	*high_node*	*low_node*	*min_diff*	*max_diff*	*final*	*branch_point*
level=0:								
01	01	01	*-root-*	StudentList	01	01	*false*	*false*
02	01	01	*-root-*	student	02	02	*false*	*false*
level=1:								
01	01	01	StudentList	FirstName	03	∞	*false*	*false*
02	01	01	student	@type	01	01	*true*	*true*
02	01	02	student	age	01	01	*false*	*true*
02	01	03	student	name	01	∞	*false*	*true*
level=2:								
02	03	01	name	FirstName	01	01	*false*	*false*

Table 4. The XPaths shown in Fig. 3 stored in relational table *XPathPred*[*level*]

level	*id*	*node_num*	*node*	*value*	*operation*
2	01	01	FirstName	Zhong	00 (=)
2	02	02	age	21	01 (>)
3	02	01	FirstName	Zhong	00 (=)

between nodes. For example, α is the ancestor of β iff *start-point*$_\alpha$ <*start-point*$_\beta$ and *end-point*$_\alpha$ >*end-point*$_\beta$. In addition to *start-point* and *end-point*, we record the "*depth*" of each node. α is the parent of β iff α is the ancestor of β and *depth*$_\beta$ =*depth*$_\alpha$ + 1. Each XML document is parsed by a SAX parser, and the parsed structure information and value information are stored in relational table *XMLFrame* and *XMLValue* respectively. Fig. 2 shows an XML document and the corresponding XML tree. Table 1 and Table 2 demonstrate how the XML document shown in Fig. 2 is stored in *XMLFrame* and *XMLValue* respectively.



```
XPath ::= QPath | QPath AND XPath

QPath ::= Path Op Const

Path   ::= Step | Path Step | Path [XPath]

Step   ::= Axis tag | Axis * | Axis @attr | Axis text()

Axis   ::= / | //

Op     ::= < | <= | = | > | >= | !=
```

XPath 1: /StudentList//*//*//FirstName/text()="Zhong"
XPath 2: /*/Student[@type][/age/text()>"21"]//name/FirstName/text()="Zhong"

Fig. 3. The XPath subscription grammar and XPath trees of two subscription examples

3.2 Subscribe Algorithm

Fig. 3 shows our XPath subscription grammar and two XPath subscriptions with the corresponding XPath trees. The grammar is a subset of XPath called "*branching path expression*". The grammar does not include the boolean OR expression, an XPath with boolean OR expression should be rewritten to disjunctive normal form and disjunctions are submitted as separate subscriptions.

The depths of XML trees are usually very low; a statistical analysis over 200,000 XML documents had been performed [18], and discovered that 99% of the documents have less than 8 levels of nesting. For "//" may be used in XPath expressions, the depths of XPath trees should be less than the depths of XML trees; so, we match XPath subscriptions with each published XML document level-by-level; XPath subscriptions are parsed level-by-level, and the edges and the predicates in the ith level of the XPath tree are stored in relational table $XPathFrame[i]$ and $XPathPred[i]$ respectively. Each XPath subscription has an "*id*" which can uniquely identify the XPath. Each edge connects two nodes in the XPath tree; we use "*high-node*" and "*low-node*" to record the node that is close to the root and the node that is close to the leaf respectively. As shown in Fig. 3, the elements and the attributes that have the same depth in an XPath tree each has a unique serial number; we use "*high-num*" and "*low-num*" to record the serial number of the *high-node* and the serial number of the *low-node* respectively. The edges in the XPath tree that are connected by "$\star$" are merged into a single edge; for example, the path from "*StudentList*" to "*FirstName*" in the XPath tree of XPath 1 in Fig. 3 has been merged into a single edge, and the depth difference between "*StudentList*" and "*FirstName*" in the XML tree must be not less than 3. We use "*max-diff*" and "*min-diff*" to record the maximal depth difference and the minimal depth difference between the *high-node* and the *low-node* in the XML tree respectively; in this way, "//", "/", and merged edge can be recorded in a uniform manner. In addition, we use "*branch-point*" and "*final*" to record whether the *high-node* of an edge is a branching point and whether the *low-node* of an edge is a leaf node respectively. For predicates, we use "*node*", "*node-num*", "*value*", and "*operation*" to record the tag of the node, the serial number of the node, the value, and the operation respectively; operations are recorded by integer values from 0 to 5, which mean "$=$", "$>$", "$\geq$", "$<$", "$\leq$", and "$\neq$" respectively. Table 3 and Table 4 demonstrate how the two XPaths in Fig. 3 are stored in the *XPathFrame* tables and the *XPathPred* tables respectively. Deleting an XPath subscription on line is as easy as deleting all the tuples whose *id* equals to the *id* of the to be deleted XPath subscription.

3.3 Matching Algorithm

Fig. 4 and Fig. 5 show our algorithm for matching XPath subscriptions with an XML document inside the relational database by relational operations to find out matched subscriptions. The matching algorithm tries to reconstruct each XPath tree from the XML tree in a bottom-up manner; if an XPath tree can be reconstructed from the XML tree, the corresponding subscription is

Algorithm Matching(max_level ξ)

```
01. INSERT   INTO MatchedXPath(id)
02. SELECT   DISTINCT id
03. FROM     XPathFrame[0];
04. for $i = \xi$ downto 0 do
05. begin
06.     FrameMatching($i$);
07.     PredicateMatching($i$);
08.     NeighborPruning($i$);
09.     BranchingPointPruning($i$);
10.     MatchedXPathCutting($i$);
11. end
```

Algorithm FrameMatching(level i)

```
INSERT   INTO  XMLMatch[i] (id, high_num, low_num, final, branch_point,
         high_start_point, low_start_point)
SELECT   id, high_num, low_num, final, branch_point,
         FH.start_point, FL.start_point
FROM     XMLFrame AS FH, XPathFrame[i], XMLFrame AS FL
WHERE    id IN (SELECT id FROM MatchedXPath)
AND      FH.node = high_node
AND      FL.node = low_node
AND      FH.start_point < FL.start_point
AND      FH.end_point > FL.end_point
AND      FL.depth − FH.depth BETWEEN min_diff AND max_diff;
```

Algorithm PredicateMatching(level i)

```
INSERT   INTO  XMLMatch[i] (id, high_num, low_num, final, branch_point,
         high_start_point, low_start_point)
SELECT   id, node_num, null, true, false, start_point, null
FROM     XMLValue AS V, XPathPred[i] AS P
WHERE    id IN (SELECT id FROM MatchedXPath)
AND      P.node = V.node
AND      (CASE
           WHEN  operation = 0 AND V.value = P.value    THEN 1
           WHEN  operation = 1 AND V.value > P.value    THEN 1
           WHEN  operation = 2 AND V.value >= P.value   THEN 1
           WHEN  operation = 3 AND V.value < P.value    THEN 1
           WHEN  operation = 4 AND V.value <= P.value   THEN 1
           WHEN  operation = 5 AND V.value <> P.value   THEN 1
           ELSE  0
         END) = 1;
```

Fig. 4. The matching algorithm (1)

Algorithm NeighborPruning(level i)

```
DELETE
FROM      XMLMatch[i] AS MH
WHERE     MH.final = false
AND       NOT EXISTS (
            SELECT     *
            FROM       XMLMatch[i + 1] AS ML
            WHERE      MH.id = ML.id
            AND        MH.low_num = ML.high_num
            AND        MH.low_start_point = ML.high_start_point);
```

Algorithm BranchingPointPruning(level i)

```
DELETE
FROM      XMLMatch[i] AS M1
WHERE     M1.branch_point = true
AND       EXISTS ((
              SELECT   XF.low_num
              FROM     XPathFrame[i] AS XF
              WHERE    M1.id = XF.id
              AND      M1.high_num = XF.high_num)
            EXCEPT     (
              SELECT   M2.low_num
              FROM     XMLMatch[i] AS M2
              WHERE    M1.id = M2.id
              AND      M1.high_num = M2.high_num
              AND      M1.high_start_point = M2.high_start_point));
```

Algorithm MatchedXPathCutting(level i)

```
DELETE
FROM      MatchedXPath AS MX
WHERE     EXISTS ((
              SELECT   id
              FROM     XPathFrame[i] AS XF
              WHERE    MX.id = XF.id)
            UNION(
              SELECT   id
              FROM     XPathPred[i] AS XP
              WHERE    MX.id = XP.id))
AND       NOT EXISTS (
            SELECT     id
            FROM       XMLMatch[i] AS XM
            WHERE      MX.id = XM.id);
```

Fig. 5. The matching algorithm (2)

matched. Firstly, *id*s of all XPath subscriptions are inserted into the *MatchedXPath* table; then, all XPath subscriptions are reconstructed level-by-level in the bottom-up manner, and the reconstruction process for each level is consisted of five successive steps called frame matching, predicate matching, neighbor pruning, branching point pruning, and matched XPath cutting respectively. After the bottom-up level-by-level matching, *id*s in the *MatchedXPath* table are *id*s of the matched XPath subscriptions, and the published XML document should be delivered to all the subscribers whose subscriptions can be matched ultimately.

Step 1: Frame Matching. The frame matching step reconstructs the ith level edges of each XPath subscription from the published XML document and inserts the reconstructed edges into the *XMLMatch*[i] table. Only the ith level edges of the XPaths whose *id* is in current *MatchedXPath* table are considered to be reconstructed. The *XMLFrame* table occurs two times in the FROM clause, one for matching the *high-node* of the edge, the other for matching the *low-node* of the edge. As presented in Section 3.1, the ancestor-descendent relationship between two nodes α and β is determined by $\textit{start-point}_\alpha < \textit{start-point}_\beta$ and $\textit{end-point}_\alpha > \textit{end-point}_\beta$, and as presented in Section 3.2, the depth difference between the descendent and the ancestor is kept between *min-diff* and *max-diff*.

Step 2: Predicate Matching. The predicate matching step reconstructs the ith level predicates of each XPath subscription from the published XML document and inserts the reconstructed predicates into the *XMLMatch*[i] table. Only the ith level predicates of the XPaths whose *id* is in the *MatchedXPath* table are considered to be reconstructed. As presented in Section 3.2, integer values from 0 to 5 mean "$=$", "$>$", "$\geq$", "$<$", "$\leq$", and "$\neq$" respectively, and a CASE-WHEN clause was used to math the XML node values with the XPath predicate values.

Step 3: Neighbor Pruning. The neighbor pruning step deletes the ith level reconstructed edges from the *XMLMatch*[i] table whose *low-node* is not a leaf node (*final=false*) and not exists an (i+1)th level reconstructed edge whose *high-node* is the same as the ith level reconstructed edge's *low-node* and not exists an (i+1)th level reconstructed predicate whose *node* is the same as the ith level reconstructed edge's *low-node*. Since the edges can not contribute to the reconstruction of the XPaths, they are deleted from the *XMLMatch*[i] table.

Step 4: Branching Point Pruning. The branching point pruning step deletes the ith level reconstructed edges from the *XMLMatch*[i] table whose *high-node* is a branching point and not all the branches of the branching point can be reconstructed from the published XML document. If the set difference between the set of branches in the XPath tree and the set of matched branches is not empty, we can ensure that not all the branches of the branching point can be reconstructed, and the branches should be deleted from the *XMLMatch*[i] table.

Step 5: Matched XPath Cutting. After neighbor pruning and branching point pruning, the matched XPath cutting step deletes *id*s of the XPath subscriptions from the *MatchedXPath* table which can not be reconstructed from the published XML document. If an XPath has at least one *i*th level edge or predicate, but not exists a reconstructed *i*th level edge or predicate, it is definite that the XPath can not be reconstructed from the published XML document, and the *id* of the XPath should be deleted. Without the cutting step, our algorithm still functions correctly, for we may delete *id*s of the XPath subscriptions which can not be reconstructed only in the top level. But, with the cutting step, *id*s of the XPaths which can not be reconstructed from the published XML document are deleted as early as possible; so, upper level frame matching and predicate matching need not to match the edges and the predicates of the subscriptions that had been deleted, and our algorithm can achieve better time performance.

4 Experiments

With our implementation, a wide variety of experiments can be conducted under different conditions. Due to space limitation, we only use two sets of experimental results to report the performance and the scalability of our publish/subscribe algorithm. The experiments run on a 1.4 GHz Pentium IV CPU with 2G of memory and a 160GB of SCSI hard disk driver running RedHat Enterprise Linux Advanced Server 4 and Oracle 9i Enterprise Edition; the database cache size was set to 32MB, and the database block size was set to 8KB. Only three SQL92 data types are used in the relational tables, which are *integer*, *boolean*, and *varchar* respectively. We set primary key for each relational table, and a B^+ tree unique index on the primary key will be created implicitly by the database system. {*id*}, {*start-point*, *end-point*}, {*start-point*}, {*id*, *high-num*, *low-num*}, {*id*, *high-num*}, and {*id*, *high-num*, *low-num*, *high-start-point*, *low-start-point*} were set as primary key of the *MatchedXPath* table, the *XMLFrame* table, the *XMLValue* table, *XPathFrame* tables, *XPathPred* tables, and *XMLMatch* tables respectively. In addition, two B^+ tree indices on the *XMLFrame* table are created explicitly for improving time performance of the frame matching step, one is on the *depth* attribute, the other is on the *node* attribute. The matching algorithm was implemented by a stored procedure in the Oracle database server, and the procedure has only one parameter which is the maximal level of the subscriptions.

In our experiments, XML messages and XPath subscriptions generated from Document Type Definition for the data and metadata at the Astronomical Data Center at NASA/GSFC [19] are used. Each XML message contains information about the metadata for a dataset, such as title, keywords, references, authors etc, and all of the associated tables, descriptions, and history. The XML messages are generated from the DTD by IBM's XML generator [20]; four XML messages are generated, whose size is 3KB, 15KB, 40KB and 100 KB respectively, and the maximal level of nesting is 5, 10, 11, and 12 respectively. The XPath subscriptions are generated from the DTD by XPath generator; each node in the XPath tree

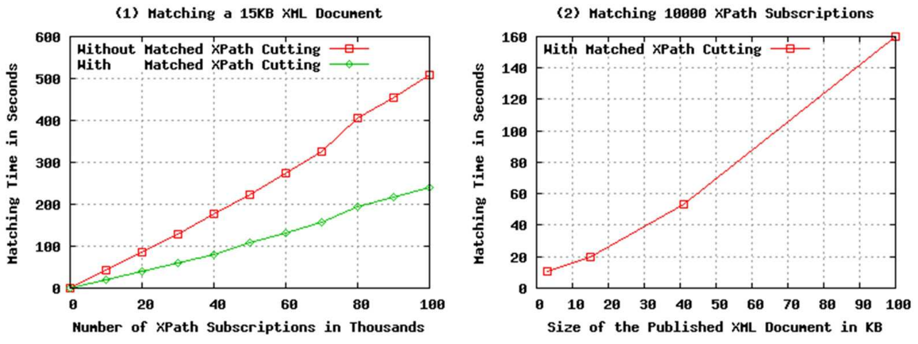


Fig. 6. Time performance of the matching algorithm. (1) The matching time in seconds for matching a 15KB XML document with XPath subscriptions whose number increases from 0 to 100000. (2) The matching time in seconds for matching 10000 XPath subscriptions with an XML document whose size increases from 3KB to 100KB.

has a 30% probability of being a branching point, has a 10% probability of being a "$\star$"; each edge in the XPath tree has a 20% probability of being a "//"; the average and the maximal depth of the XPath trees are 5 and 10 respectively. In addition, due to the code of the complicated publish/subscribe system proposed in [12] belongs to IBM, and the paper only gives a brief introduction of the matching algorithm, we did not conduct comparison with the algorithm in [12].

In the first set of experiments, we match the 15KB XML message whose XML tree contains 328 inner nodes and 227 values with XPath subscriptions whose number increases from 0 to 100000. Fig 6.1 shows the time performance of the matching algorithm with and without matched XPath cutting. We can see that the matching algorithm with matched XPath cutting outperforms the algorithm without matched XPath cutting in a ratio of about 50%. In the second set of experiments, we match each of the generated 4 XML messages with 10000 XPath subscriptions. Fig 6.2 shows the time performance of the matching algorithm with matched XPath cutting. We can see that the growth rate of the matching time increases with the increase of the XML message size, the reason is that the percentage of the matched subscriptions increases at the same time.

5 Conclusions

In the paper, we introduced an XML publish/subscribe algorithm which is consisted of publish algorithm, subscribe algorithm, and matching algorithm. The matching algorithm uses relational operators to match XPath subscriptions with XML documents inside relational databases, so the scalability of the algorithm is no longer limited by amount of available physical memory. Experimental results show that the matching algorithm has good performance and scalability, so the algorithm should be a good choice for implementing publish/subscribe systems.

References

1. Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. Proceedings of SIGMOD 2000, 379-390.
2. Praveen Seshadri. Building Notification Services with Microsoft SQLServer. Proceedings of SIGMOD 2003, 635-636.
3. Hansjörg Zeller. NonStop SQL/MX Publish/Subscribe: Continuous Data Streams in Transaction Processing. Proceedings of SIGMOD 2003, 636.
4. Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching Events in a Content-Based Subscription System. Proceedings of PODC 1999, 53-61.
5. Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. Proceedings of VLDB 2000, 53-64.
6. Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and Scalable Filtering of XML Documents. Proceedings of ICDE 2002, 341.
7. Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. Proceedings of ICDE 2002, 235-244.
8. Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath Expressions. VLDB Journal. 2002, 11(4), 354-379.
9. Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata. Proceedings of ICDT 2003, 173-189.
10. Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. Proceedings of SIGMOD 2003, 431-442.
11. Ashish Kumar Gupta and Dan Suciu. Stream Processing of XPath Queries with Predicates. Proceedings of SIGMOD 2003, 419-430.
12. Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing A Scalable XML Publish/Subscribe System Using Relational Database Systems. Proceedings of SIGMOD 2004, 479-490.
13. Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. ACM Transactions on Internet Technology. 2001, 1(1), 110-141.
14. Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. QRS: A Robust Numbering Scheme for XML Documents. Proceedings of ICDE 2003, 705-707.
15. Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. Proceedings of SIGMOD 2002, 204-215.
16. Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML Trees. Proceedings of PODS 2002, 271-281.
17. Xiaodong Wu, Mong-Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. Proceedings of ICDE 2004, 66-78.
18. Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. Web XML: A First Study. Proceedings of WWW 2003, 500-510.
19. http://tarantella.gsfc.nasa.gov/xml/dataset_dtd.txt
20. http://www.alphaworks.ibm.com/tech/xmlgenerator