Efficient Evaluation of Multiple Queries on Streamed XML Fragments

Huan Huo, Rui Zhou, Guoren Wang, Xiaoyun Hui,
Chuan Xiao, and Yongqian Yu

Institute of Computer System, Northeastern University, Shenyang, China
wanggr@mail.neu.edu.cn

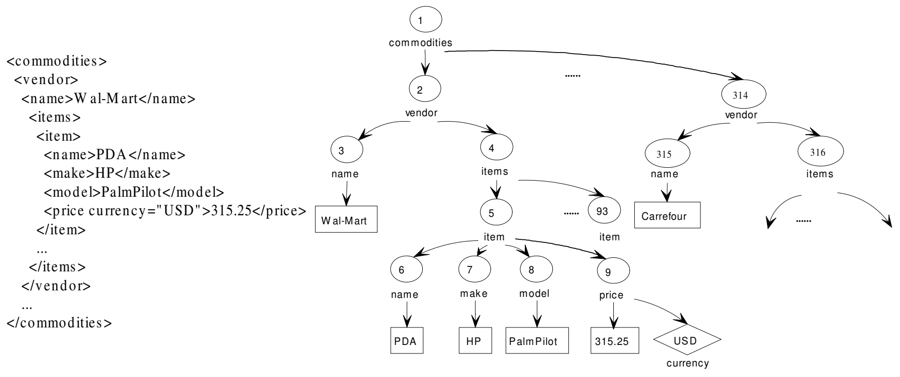
Abstract. With the prevalence of Web applications, expediting multiple queries over streaming XML has become a core challenge due to one-pass processing and limited resources. Recently proposed Hole-Filler model is low consuming for XML fragments transmission and evaluation, however existing work addressed the multiple query problem over XML tuple streams instead of XML fragment streams. By taking advantage of schema information for XML, this paper proposes a model of tid+ tree to construct multiple queries over XML fragments and to prune off duplicate and dependent operations. Based on tid+ tree, it then proposes a notion of FQ-Index as the core in M-XFPro to index both multiple queries and XML fragments for processing multiple XPath queries involving simple path and twig path patterns. We illustrate the effectiveness of the techniques developed with a detailed set of experiments.

1 Introduction

The recent emergence of XML [1]as a *de facto* standard for information representation and data exchange over the web has led to an increased interest in using more expressive subscription/filtering mechanisms that exploit both the structure and the content of XML documents. Evaluating XML queries, such as XPath [2] and XQuery [3], is thus widely studied both in traditional database management systems and in stream model for web applications. Figure 1 gives an XML document and its DOM tree, which acts as an example of our work.

Recently, many research works [4–10] focus on answering queries on streamed XML data, which has to be analyzed in real-time and by one pass. In the push-based model [4,5], XML streams are broadcasted to multiple clients, which must evaluate continuous, sophisticated queries (as opposed to simple, single path specifications) with limited memory capacity and processing power. In the pull-based model [6–10], such as publish-subscribe or event notification systems, XML streams are disseminated to subscribers, but a larger number of registered queries pose heavy workload on the server. Hence, expediting multiple queries on XML streams is the core technical challenge.

In order to reduce processing overhead, *Hole-Filler* model is proposed in [11]. In the model, a hole represents a placeholder into which another rooted subtree (a fragment), called a filler, could be positioned to complete the tree. In this



```
<commodities>
 <vendor>
  <name>Wal-Mart</name>
   <items>
    <item>
     <name>PDA</name>
     <make>HP</make>
     <model>PalmPilot</model>
     <price currency="USD">315.25</price>
    </item>
    ...
   </items>
  </vendor>
  ...
</commodities>
```

Fig. 1. An XML Document and its DOM Tree

way, infinite XML streams turn out to be a sequence of XML fragments, and queries on parts of XML data require less memory and processing time, without having to wait for the entire XML document to be received and materialized. Furthermore, changes to XML data may pose less overhead by sending only fragments corresponding to the changes, instead of sending the entire document.

However, to the best of our knowledge, there is no work for evaluating multiple queries on streamed XML fragments so far. In XFrag [4] and XFPro [12], XML fragments can only be evaluated under simple, single queries. While other research work [6–10] consider problems on a stream of XML tuples, not XML fragments, and can not avoid "redundant" operations caused by fragments.

In this paper, we present an efficient framework and a set of techniques for processing multiple XPath queries over streamed XML fragments. As compared to the existing work on supporting XPath/XQuery over streamed XML fragments, we make the following contributions: (i)we propose techniques for enabling the transformation from multiple XPath expressions to optimized query plan. We model the query expressions using *tid+ tree* and apply a series of pruning policies, which enable further analysis and optimizations by eliminating the "redundant" path evaluations. (ii)based on tid+ tree, we present a novel index structure, termed FQ-Index, which supports the efficient processing of multiple queries (including simple path queries and twig path queries) for streamed XML fragments by indexing both the queries and the fragments. (iii)based on FQ-Index, we address the main algorithms of query evaluation in M-XFPro, which is able to both reduce the memory cost as well as avoid redundant matchings by recording only query related fragments. Note that, we assume the query ends cannot reconstruct the entire XML data before processing the queries.

The rest of this paper is organized as follows. Section 2 introduces *Hole-Filler* model as the base for our XML fragments. Section 3 gives a detailed statement of our multiple query processing framework. Section 4 shows experimental results from our implementation and reflects the processing efficiency of our framework. Our conclusions are contained in Section 5.

2 Model for Streamed Fragmented XML Data

In our approach, we adopt the hole-filler model [11] to correlate XML fragments with each other. We assume that XML stream begins with finite XML documents and runs on as and when new elements are added into the documents or updates occur upon the existing elements.

Given an XML document tree $T_d = (V_d, E_d, \Sigma_d, root_d, Did)$, a filler $T_f = (V_f, E_f, \Sigma_f, root_f, fid, tsid)$ is a subtree of XML document associating a fid and a $tsid$, where V_f, E_f, Σ_f is the subset of node set V_d, edge set E_d and element type set Σ_d respectively, and $root_f$ $(\in V_f)$is the root element of the subtree; a hole H is an empty node $v(\in V_d)$ assigned with a unique hid and a $tsid$, into which a filler with the same fid value could be positioned to complete the tree. Note that the filler can in turn have holes in it, which will be filled by other fillers. We can reconstruct the original XML document by substituting holes with the corresponding fillers at the destination as it was in the source. In this paper, we assume that XML documents have been fragmented already. Fragmenting algorithm is stated in [13] and omitted here. Figure 2 gives two fragments of the document in Figure 1.

```
Fragment 1:
<commodities filler id="0" tsid="1">
  <vendor>
    <name>Wal-Mart</name>
      <items>
        <stream: hole id="10" tsid="5" />
        <stream: hole id="20" tsid="5"  />
        ...
  </vendor>
  ....
</commodities>
```

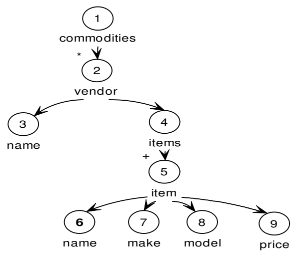
```
Fragment 2:
<stream: filler id="10" tsid="5">
  <item>
    <name>PDA</name>
    <make>HP</make>
    <model>PalmPilot</model>
    <price currency="USD">315.25</price>
  </item>
</stream: filler>
```

Fig. 2. XML Document Fragments

In order to summarize the structure of XML fragments, *tag structure* [11] is exploited to provide structural information (including fragmentation information) for XML and capture all the valid paths. A tag structure $TS = (V_t, E_t, root_t, \Sigma_t, TYPE_t)$ is itself structurally a valid XML fragment with the highest priority, where V_t is a set of tag nodes in XML document, E_t is a set of edges, Σ_t is a set of $tsid$s identifying the tag nodes in XML document, and $TYPE_t$ is a set of tag node type. Tag structure can be generated according to XML Schema or DTD, and also can be obtained when fragmenting an XML document without DTD. The DTD and the corresponding tag structure of the XML document (given in Figure 1) are depicted in Figure 3.

3 M-XFPro Query Handling

Based on the Hole-Filler model, we have proposed M-XFPro, a system aimed at providing efficient evaluation for multiple queries over streamed XML fragments. In this section, we first introduce *tid+ tree* for rewriting the queries for



```
<stream: structure>
  <tag name="commodities" id="1" Filler="true">
    <tag name="vendor" id="2"  Filler="true">
      <tag name="name" id="3" />
      <tag name="items" id="4">
        <tag name="item" id="5"  Filler="true">
          <tag name="name" id="6" />
          <tag name="make" id="7" />
          <tag name="model" id="8" />
          <tag name="price" id="9" />
        </tag>
      </tag>
    </tag>
  </tag>
</stream: structure>
```

Fig. 3. Tag Structure of Hole-Filler Model

XML fragments, and describe the pruning policies to eliminate "redundant" path evaluations. Then we present our novel FQ-Index for processing streamed XML fragments based on optimized *tid+ tree*. We present the main matching algorithms for query handling with FQ-Index at last.

3.1 Tid+ Tree Construction

In our earlier framework [12], we propose *tid tree* to represent the structural patterns in an XPath query. Each navigation step in an XPath is mapped to a tree node labelled with a tag code, which encodes the tsid and "TYPE" together. For "$Filler = true$", we set the end of the tag code with "1", otherwise we set it with "0". As for tsid, we separate it from the "TYPE" code by a dot. By checking the end of the code, we can easily tell subroot nodes (i.e. the root of a filler) from subelement nodes (i.e. the node that locates in a filler but is not the root of the subtree).

We expand the concept of *tid tree* into *tid+ tree* to represent multiple query expressions and enable further analysis and optimizations on query operations.

Given a collection of XPath expressions $P = \{p_1, p_2, \cdots, p_n\}$, we map multiple queries into a single tree, noted as *tid+ tree*, by defining $root_t$ as a special root node, which allows for conjunctive conditions at the root level. Parent-child relationship is represented by a single arrow, while ancestor-descendant relationship is represented by a double arrow. And the output of each query q_i is depicted by a single arrow and marked with the ID of q_i. In order to distinguish between the nodes that represent a tag code and the nodes that represent an atomic predicate, we represent nodes of tag code with circles and values of predicate with rectangles. The operators (such as $<$, $>$, $\geq$, $\leq$, $=$) and boolean connectors are represented with diamonds. Note that the common prefixes of all the queries are shared.

Figure 4 shows an example of such a tid+ tree, representing three queries on the XML document described in Section 1, where *Query 1* and *Query 2* share the common prefix "$/commondities/vendor$" (i.e./1.1/1.2). Since "*name*" in *Query*

2 corresponds to two tsids in the tag structure, we enumerate all the possible tsids in the tid+ tree such that *Query 2* has two output arrows.

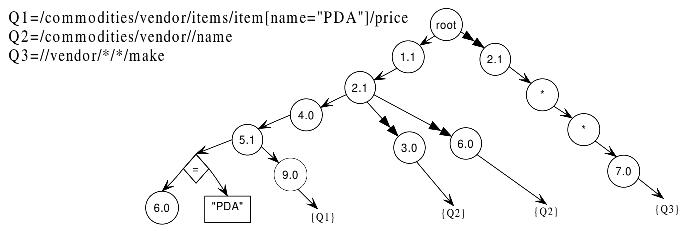

Q1=/commodities/vendor/items/item[name="PDA"]/price
Q2=/commodities/vendor//name
Q3=//vendor/*/*/make
root
1.1
2.1
2.1
*
4.0
*
5.1
3.0
6.0
7.0
=
9.0
6.0
"PDA"
{Q1}
{Q2}
{Q2}
{Q3}

Fig. 4. Tid+ Tree

Since tid+ tree is the base for FQ-Index to install multiple XPath expressions into the indexing structure, the optimization of tid+ tree impacts both the space and performance of the index. We now introduce two kinds of optimizations on tid+ tree to eliminate the redundant operations as early as possible.

Duplication Pruning Given an XPath p, we define a simple subexpression s of p if s is equal to the path of the tag nodes along a path $< v_1, v_2, \cdots v_n >$ in the tid tree of p, such that each v_i is the parent node of $v_{i+1} (1 \leq i < n)$ and the label of each v_i (except perhaps for v_1) is prefixed only by "/".

Definition 1. *Given a collection of XPath expressions $P = \{p_1, p_2, \cdots, p_n\}$, subexpression s is a common subexpression if more than one tid tree of p_i contains s. If a common subexpression is also a simple subexpression, we define it as a simple common subexpression. A common subexpression s is defined as a* maximal common subexpression *if no other longer common subexpression in the tid+ tree of P contains s.*

Common subexpressions degrades the performance significantly, especially when the workload has many similar queries. Since the common prefixes of all the queries are shared in tid+ tree, we consider optimizing tid+ tree by grouping all the common subexpressions in the structure navigation.

In order to extract the common subexpressions, we have to find out the structural relationship shared among the queries. By taking advantage of tag structure, we can replace "//" in tid+ tree with the corresponding structure consisting of "/" and expand "*" in tid+ tree to specify query execution. As for twig pattern query, we add the subroot nodes involved in the branch expression into the tid+ tree if the testing node and the branch expression belong to different fragments. In this way, common subexpressions turn out to be simple common subexpressions, and all the possible duplicated expressions can be pruned off.

Figure 5(a) presents the tid+ tree in Figure 4 after eliminating "//" and "*" based on tag structure, where the dashed regions enclose the subexpression (i.e. /1.1/2.1/4.0/5.1) shared by *Query 2* and *Query 3* while the solid regions enclose the subexpression (i.e. /1.1/2.1/4.0) shared by *Query 1*, *Query 2* and *Query 3*. Since tid node 5.1 in *Query 1* has a predicate, which is not included in the other two queries, we treat the tid node 5.1 in *Query 1* as a different node and exclude it in the common subexpression. Note that Figure 5(a) captures all the maximal common subexpressions among the queries. The optimized tid+ tree after pruning off the duplicated subexpressions is presented in Figure 5(b).

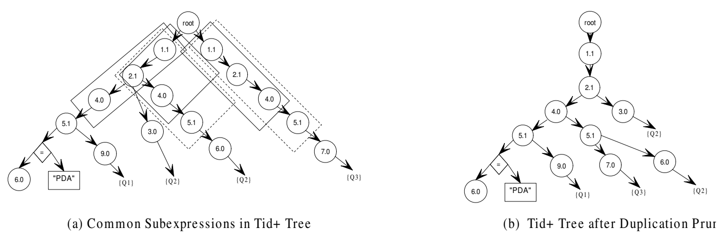

root
1.1
2.1
4.0
5.1
3.0
6.0
7.0
9.0
"PDA"
{Q1}
{Q2}
{Q3}
(a) Common Subexpressions in Tid+ Tree
(b) Tid+ Tree after Duplication Prur

Fig. 5. Duplication Pruning on Tid+ Tree

Dependence Pruning Before we describe the dependence pruning policy for tid+ tree, we first introduce some definitions of operation dependence.

Definition 2. *Given any pair of nodes in a tid+ tree* $< n_1, n_2 >$*, if the query result of* n_2 *is valid only if the query result of* n_1 *is valid,* n_2 *is defined as dependent on* n_1*. We use a directed edge* $e = (n_1, n_2)$ *to imply the dependence between* n_1 *and* n_2*.*

Definition 3. *Given any pair of nodes in a tid+ tree* $< n_1, n_2 >$*, we say that* n_2 *is subsumption dependent on* n_1 *if: (i)* n_2 *is dependent on* n_1*, and (ii) the query result of* n_2 *is a subset of the query result of* n_1*.*

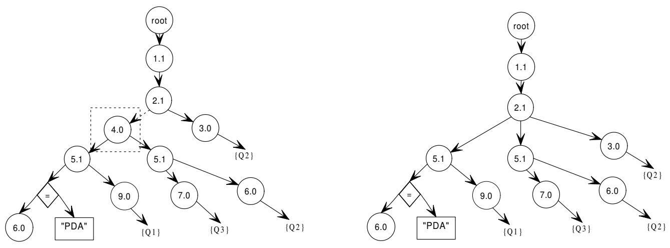
In streaming XML fragment model, operation dependence usually occurs when the query results to preceding query node and successive query node are in the same fragment(here we are not considering predicates), since the fragments with the same tsid share the same structure so that any fragment matching the preceding node also matches the successive one. In most cases, the dependence operation can be eliminated by removing the successive query nodes.

When the query node involve predicates, if the result set of predicate p_2 is a subset of that of predicate p_1, we refer to p_2 as subsumption dependent on p_1. Subsumption-free queries are intuitively queries that do not contain "redundancies". Some queries can be rewritten to be subsumption-free, by eliminating redundant portions.

Much of our analysis focuses on pruning off operation dependencies on tid nodes caused by fragmentation to eliminate "redundant" structural evaluations.

Since tag structure guarantees that the fragments with the same tsid share the same structure, we keep all the subroot nodes and delete the subelement nodes which have no predicates and are not the leaf nodes in tid+ tree. According to tag code, subroot nodes ended with “1” are kept in the tid+ tree while subelement nodes ended with “0” and without predicate nodes in their children are removed. Thus the original tid+ tree becomes an optimized tid+ tree.

Figure 6(a) shows the operation dependence in the optimized tid+ tree in Figure 5(b), where tid node 4 depends on tid node 2 and is referred to as a dependent node. We use dashed arrows to represent operation dependencies, and dashed rectangles for dependent nodes. Figure 6(b) shows the optimized tid+ tree after pruning off the operation dependencies.



(a) The Dependence Node in Tid+ Tree (b) The Tid+ Tree after Dependence Pruning

Fig. 6. Dependence Pruning on Tid+ Tree

3.2 FQ-Index Scheme

Our FQ-Index is a hybrid index structure, which indexes both the queries and fragments on the basis of optimized tid+ tree. An FQ-Index consists of two key components: (1) a query index (denoted by QI), constructed by tid+ tree to facilitate the detection of query matchings in the input XML fragments; and (2) a filler table (denoted by FT), which stores the information about each XML fragment. Both of the components share a hash table for subroot nodes in tid+ tree. We now describe each of these two components in detail.

Query Index Query index is generated from optimized tid+ tree before processing to keep track of the query steps that are supposed to match next. Let $P = \{p_1, p_2, \ldots, p_n\}$ denote the set of XPath expressions, and $T = \{t_1, t_2, \ldots, t_n\}$ denote the subroot nodes in optimized tid+ tree. Query index QI of P for each t_i is a 4-tuple list. Each item in the query list for t_i is a 4-tuple (*query id set, predecessor, successor, predicate*), denoted as *q-tuple*, where:

– *Query id set* represents the queries in set P that share the same predicate, predecessor and successor.

– *Predecessor* refers to the tag code of the fragment in tid+ tree corresponding to the parent node of q_i. (*Predecessor* = NULL if q_i is a root node.)
– *Successor* refers to the tag code of the fragment in tid+ tree corresponding to the child node of q_i. (*Successor* = NULL if q_i is the end of the query.)
– *Predicate* is the branch expression of twig path queries in tid+ tree.

Predecessor and successor in each item keep track of the query steps, while predicate keeps the reference of branch expressions. With the help of query id set, we can avoid duplicate evaluations shared by multiple queries. Since subroot nodes indicate the tsids of the fragments involved in the queries, we can directly access the relative query steps by the corresponding entry of the hash table when a fragment arrives. Figure 7 presents the query index converted from the optimized tid+ tree(*"all"* represents all of the queries in set P) in Figure 6(b).

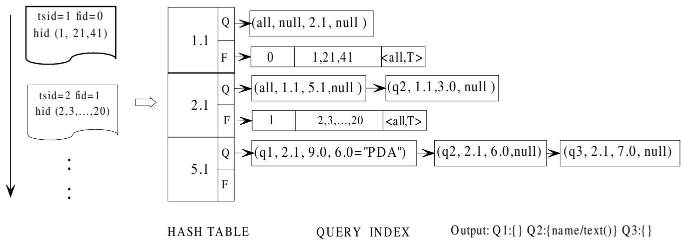


Fig. 7. FQ-Index of Tid+ Tree

Filler Table As fragments in the original document may arrive in any order and query expressions may contain predicates at any level in the XML tree, it is necessary to keep track of the parent-child links between the various fragments. We maintain the fragments' information in filler table at each entry of the hash table when processing arrived fragments. Since the structural information corresponds to a small part of the actual data in the XML fragment, the rest of which is not relevant in producing the result, we discard the fragments corresponding to intermediate steps to save space cost.

The filler table FT contains one row for each fragment. Each row in FT is denoted as a *f-tuple* $(fillerid, \{holeid\}, \{< q_i, tag >\})$, in which tag can be set to $true$, $false$, undecided ($\perp$), or a result fragment corresponding to q_i in set P. While the former three values are possible in intermediate steps that do not produce a result, the latter is possible in the terminal steps in the tid+ tree branch. Figure 7 shows the construction of filler table.

With the hash table, the filler table and the query index cooperate together as FQ-Index. Taking advantage of the query index, we can quickly inquire the parent fragment by matching the same *holeid* in the *predecessor's* FT. In this way, filler table enhances the performance by only maintaining the information of fragments that will contribute to the results. Thus FQ-Index efficiently supports the online evaluation of multiple queries over streamed XML fragments, including both simple path queries and twig pattern queries.

3.3 Query Handling

In this section, we address the main algorithms of query evaluation in M-XFPro. The basic idea of the matching algorithms is as follows. We use the query index QI to detect the occurrence of matching tsids as the input fragments stream in, since before we record the structrual information of a fragment, it needs to verify if the preceding operation has excluded its parent fragment due to either predicate failure or due to exclusion of its ancestor.

For example, *Query 1: /commodities/vendor[name=“Wal-Mart”]//item* [*make* = “HP”] is a twig pattern query with two atomic predicates, while *Query 2: /commodities/vendor[name=“Price-Mart”]//item[make=“IBM”]* is a similar query just with different predicates. When the “commodities” fragment with tsid “1”, filler id “0” and hole ids “1, 21, 41” arrives, the FT to the entry 1 is updated as $(0, \{1, 2, 41\}, \{< all, T >\})$. Note that, the “commodities” filler can be discarded as it is no more needed to produce the result and the hole filler association is already captured. This results in memory conservation on the fly. When the “vendor” fragment with tsid “2”, fillerid “1”, holeid “$2, 3, \cdots, 20$” and “name=Wal-Mart” arrives, the FT to the entry 2 is updated as $(1, \{2, 3, \cdots, 20\}, \{< q2, T >, < q3, F >\})$. When the “item” fragment with fillerid “2” arrives, only after determine that the filler matches the predicate of *Query 1* [$make$ = “HP''], the fragment can be regarded as the query result of *Query 1*. Taking advantage of QI, it won’t be mixed up with the result of *Query 2* [$make$ = “IBM''] since the *Predecessor* has excluded its parent fragments.

Algorithm 1 startElement()

```
1: if ( isFragmentStart()==true ) then
2:    $fid$=getFid(); $tsid$=getTsid();
3:    if ( hashFindEntry($tsid$)!=null) then
4:       fillQueryFT($tsid$,createFTuple($fid$));
         // generate an f-tuple and fill it into the corresponding queries’ lists in $FT$;
5:    end if
6: end if
7: if ( isHoleTag()==true ) then
8:    $hid$=getHid(); $tsid$=getTsid();
9:    addQueryFT($tsid$,$hid$);
      // find the entry by $tsid$ and fill $hid$ into the corresponding f-tuple;
10: else if ( isElementTag()==true ) then
11:    $tsid$=getTsid();
12:    if ( isQueryRelatedTag()==true ) then
13:       $relevantTag$==true;
14:    end if
15: end if
```

We implement the callback functions startElement() and endElement() of SAX interface when parsing each XML fragment. In algorithm 1, if an element is a subroot node, the information of the corresponding fragment in which it falls will be captured and loaded into FT. Similar operation is performed when encountering the element representing a hole. The variable *relevantTag* will be set to *true* if the element is query related. In algorithm 2, parent fragment and predicate fragment of the filler containing the element are inquired, and *tag* value of the corresponding f-tuple is set to *true* in case both kinds of the above fragments are valid. Child fragments need to be trigged as well, for some early arrived fragments may be set to "⊥" and waiting for their parent fragments.

Algorithm 2 endElement()

```
1: if ( isFragmentEnd()==true ) then
2:    // $ft$ is the corresponding f-tuple of the current fragment
3:    if ( findParentFTuple($ft$)!=null ) then
4:       $ft$.parentValue=parentFTuple($ft$).parentValue;
5:    end if
6:    if ( findTwigPredicate($ft$)!=null ) then
7:       $ft$.conditionValue=conditionFTuple($ft$).conditionValue;
8:    end if
9:    if ( findChildFTupleList($ft$)!=null ) then
10:      for each child f-tuple $ftc$ of $ft$ do
11:         $ftc$.parentValue=$ft$.parentValue && $ft$.conditionValue;
12:      end for
13:   end if
14: end if
```

4 Performance Evaluation

In this section, we present the results of performance evaluation of various algorithms over queries with different types, depths and document sizes on the same platform. We consider the following algorithms: (1) M-XFPro, (2)Du-XFPro, i.e. M-XFPro based on tid+ tree without dependence pruning, (3)De-XFPro, i.e. M-XFPro based on tid+ tree without duplication pruning. All the experiments are run on a PC with 2.6GHz CPU, 512M memory. Data sets are generated by the xmlgen program [14]. We have written an XML fragmenter that fragments an XML document into filler fragments to produce an XML stream, based on the tag structure defining the fragmentation layout. And we implemented a query generator that takes the DTD as input and creates sets of XPath queries of different types and depths.

In figure 8(a) three kinds of processing strategies over various query numbers are tested and compared. The numbers of queries in each set are 1,2,10 respectively. From the result, we can conclude that dependence pruning and duplication pruning in M-XFPro play an important role in efficiently evaluating multiple queries. In the following experiments, we fix the query number and test other properties of the queries. Figure 8(b) shows the performance on different types of queries: (1)simple path queries only involving “/”, denoted as Q_1 (2)simple path queries involving “*” or “//”, denoted as Q_2 (3)twig pattern queries with value predicates, denoted as Q_3. We can see that for any query type, M-XFPro outperforms its counterparts, and query types do not bring in exceptions, i.e. query performance doesn’t vary much on different query types. For simplicity, but without losing generality, we only test twig queries in the next two set of experiments. Figure 8(c) shows the impacts of various query depths. Considering the depth of the XML documents generated by xmlgen, we design three query sets of depth 3, 5 and 7 respectively. As is shown in the figure, when the depth increases, the processing time of De-XFPro and Du-XFPro increases due to the increased path steps. While with duplication and dependence pruning, M-XFPro greatly reduces path steps, furthermore time cost of deep queries is much less than short queries, since fragment processing is much faster. Figure 8(d) shows the influence of different document size: 5M, 10M and 15M.

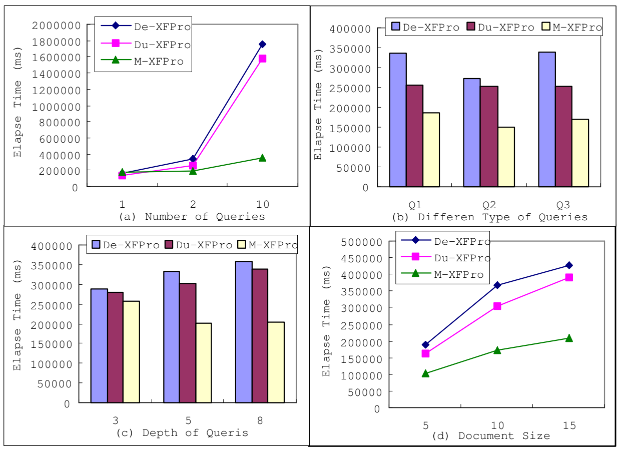


Fig. 8. Experimental Results

5 Conclusions

In this paper, we have proposed a framework and a set of techniques for processing multiple XPath queries over streamed XML fragments. We first model the multiple queries into tid+ tree, which helps to transform queries on element nodes to queries on XML fragments and serves as the base for analyzing “redundant” operations caused by common subexpression and operation dependence. Based on optimized tid+ tree after duplication pruning and dependence pruning, FQ-Index is proposed to index both the queries and fragments by sharing a hash table for tid nodes, which supports not only simple path queries, but also twig pattern queries. Our experimental results over multiple XPath expressions with different properties have clearly demonstrated the benefits of our approach.

Acknowledgments This research was partially supported by the National Natural Science Foundation of China (Grant No. 60273079 and 60573089) and Specialized Research Fund for the Doctoral Program of Higher Education (SRFDP).

References

1. W3C Recommendation: Extensible Markup Language (XML) 1.0 (Second Edition). (2000) http://www.w3.org/TR/REC-xml.
2. W3C Working Draft: XML Path Languages (XPath), ver 2.0. (2001) Tech. Report WD-xpath20-20011220, W3C, 2001, http://www.w3.org/TR/WD-xpath20-20011220.
3. W3C working draft: XQuery 1.0: An XML Query Language. (2001) Technical Report WD-xquery-20010607, World Wide Web Consortium.
4. Bose, S., Fegaras, L.: XFrag: A query processing framework for fragmented XML data. In: Eighth International Workshop on the Web and Databases (WebDB 2005), Baltimore, Maryland (June 16–17,2005)
5. Bose, S., Fegaras, L., Levine, D., Chaluvadi, V.: A query algebra for fragmented XML stream data. In: Proceedings of the 9th International Conference on Data Base Programming Languages, Potsdan, Germany (September 6–8, 2003)
6. Altmel, M., Franklin, M.: Efficient filtering of XML documents for selective dissemination of information. In Abbadi, A.E., Brodie, M.L., Chakravarthy, S., Dayal, U., Kamel, N., Schlageter, G., Whang, K.Y., eds.: Proceedings of the 26th International Conference on Very Large Data Bases, Cario, Egypt, Morgan Kaufmann (2000) 53–63
7. Diao, Y., Fischer, P., Franklin, M., To, R.: YFilter: efficient and scalable filtering of XML documents. [15]
8. Chan, C.Y., Felber, P., Garofalakis, M.N., Rastogi, R.: Efficient fltering of XML documents with XPath expressions. [15]
9. Gupta, A.K., Suciu, D.: Stream processing of XPath queries with predicates. In: SIGMOD Conference, San Diego, CA, ACM (2003) 419–430
10. Lee, M.L., Chua, B.C., Hsu, W., Tan, K.L.: Efficient evaluation of multiple queries on streaming XML data. In: Eleventh International Conference on Information and Knowledge Management, McLean, Virginia, USA (November 4–9, 2002)
11. Fegaras, L., Levine, D., Bose, S., Chaluvadi, V.: Query processing of streamed XML data. In: Eleventh International Conference on Information and Knowledge Management (CIKM 2002), McLean, Virginia, USA (November 4–9, 2002)
12. Huo, H., Wang, G., Hui, X., Zhou, R., Ning, B., Xiao, C.: Efficient query processing for streamed XML fragments. In: The 11th International Conference on Database Systems for Advanced Applications, Singapore (April 12–15,2006)
13. Huo, H., Hui, X., Wang, G.: Document fragmentation for XML streams based on hole-filler model. In: 2005 China National Computer Conference, Wu Han, China (October 13–15,2005)
14. Diaz, A.L., Lovell, D.: XML Generator. (1999) http://www.alphaworks.ibm.com/tech/xmlgenerator.
15. Proceedings of the the 2002 International Conference on Data Engineering. In: ICDE Conference, San Jose, California, USA (2002)