Information Systems 35 (2010) 170–185

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

A bi-labeling based XPath processing system

Yi Chen [a,*,1], Susan B. Davidson [b,2], Yifeng Zheng [c]

[a] Arizona State University, United States
[b] University of Pennsylvania, United States
[c] Amazon.com, United States

ARTICLE INFO

Article history:
Received 9 August 2008
Accepted 27 May 2009
Recommended by: D. Shasha

Keywords:
XML
XPath
Query processing

ABSTRACT

We present BLAS, a Bi-LAbeling based XPath processing System. BLAS uses two labeling schemes to speed up query processing: P-labeling for processing consecutive child (or parent) axis traversals, and D-labeling for processing descendant (or ancestor) axis traversals. XML data are stored in labeled form and indexed. Algorithms are presented for translating XPath queries to SQL expressions. BLAS reduces the number of joins in the SQL query translated from a given XPath query and reduces the number of disk accesses required to execute the SQL query compared with the traditional XPath processing using D-labeling alone. We also propose an approximate P-labeling scheme and the corresponding query translation algorithm to handle XML data trees that contain a large number of distinct tag names, and/or are very deep. This extension captures a spectrum of XPath-to-SQL query translation schemes, ranging from existing schemes that do not use P-labels to the one that uses exact P-labels. Experimental results demonstrate the efficiency of the BLAS system.

1. Introduction

As XML has become the de facto standard for representing data on the Web and XPath the basis of query languages for XML data, the ability to efficiently query XML data using XPath has received much attention.

One general approach, which has been proposed and studied in academia [8,13,15,20,21,24,26,34,41,44] as well as by major database vendors [7,35,39], is to design a framework for XML data processing that leverages relational database technology. In this framework, XML nodes are stored in table form with B+-tree indexes built on it, and XPath expressions are logically rewritten to SQL queries with joins implementing axis traversals. To reduce the number of expensive join operations in the resulting SQL queries, a number of labeling schemes have been proposed [2,4,22,34,36]. This paper expands the spectrum of existing labeling schemes to further speed up XPath query processing.

As an example, suppose a biologist is interested in proteins belonging to the "cytochrome c" family and wants to know who has worked on this family of proteins. Using the protein repository in XML format shown in Fig. 2, the XPath query shown in Fig. 1 could be used to retrieve the desired information.

One of the first proposals for using relational databases to store and process XML data was to treat an XML document as a graph and generate a tuple for each node, recording the identifier of the node as well as that of its parent node [24]. In this way, a child axis traversal can be achieved using a join.

To reduce the number of joins due to child axis traversal, techniques based on the schema of the data were proposed to *inline* each leaf node without siblings with the same tag name into the parent tuple

* Corresponding author.
E-mail addresses: yi@asu.edu (Y. Chen), susan@cis.upenn.edu (S.B. Davidson), yifeng@amazon.com (Y. Zheng).

[1] Supported by NSF CAREER award IIS-0845647 and NSF Grant IIS-0740129.
[2] Supported by NSF IDM 0415810 and NSF IIS 0513778.

0306-4379/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2009.05.005

```
/ProteinDatabase/ProteinEntry[.//protein/calssification/super-
family="cytochrome c"]//reference/refinfo/authors/author
```

Fig. 1. A sample XPath query.

```
<ProteinDatabase>
  <ProteinEntry>
    <protein>
      <name> cytochrome c [validated]</name>
      <classification>
        <superfamily>cytochrome c</superfamily>
      </classification> ...
    </protein>
    <reference>
      <refinfo>
        <authors>
          <author>Evans, M.J.</author> ...
        </authors>
        <year>2001</year>
        <title> The human somatic cytochrome c gene </title> ...
      </refinfo> ...
    </reference> ...
  </ProteinEntry> ...
</ProteinDatabase>
```

Fig. 2. Sample XML protein repository.

[8,13,15,21,41]. For example, suppose the schema for the XML data in Fig. 2 states that each `protein` node has a single child with tag name `name`. Then we can record information about the `name` node in the same tuple as its parent `protein` node, thus eliminating the need for a join when evaluating the child axis between them. However, in general, many joins are still needed to evaluate a single descendant axis traversal.

To reduce the number of joins and efficiently handle descendant axis traversals, a labeling scheme was proposed [2,4,20,22,34,44], which we will refer to as *D-labeling* (D stands for descendant axis). D-labeling encodes every XML node by a pair of numbers (an interval) such that the ancestor–descendant relationship between two nodes can be determined simply by comparing their intervals. The level of a node in the tree is also used to distinguish the parent–child relationship from the ancestor–descendant relationship. In this way, both a descendant and a child axis can be processed using one join. To improve the performance of joins based on D-labeling, several techniques have been proposed [3,11,17,29,30,43], which have been shown in DeHaan et al. [20] to be extremely efficient compared with other implementations of XML query processing.

However, for queries such as the one in Fig. 1 the number of joins needed using D-labeling is still too big. To evaluate this query using D-labeling, the list of nodes tagged with `ProteinDatabase` and with `ProteinEntry` are retrieved, respectively, and are joined according to their D-labels. The nodes tagged with `ProteinEntry` will also be joined with the nodes tagged with `protein`, and so on. Essentially, every XPath axis traversal in the query except for the one starting from the document root entails a join over D-labels. Thus in our example a total of eight joins are needed.

In this paper, we address the problem of reducing the number of joins and disk accesses required for XPath queries, as well as enabling more efficient joins by reducing the size of intermediate results participating in joins. Our approach, called BLAS (a Bi-LAbeling based System), uses two labeling schemes: P-labeling (where P stands for path) and D-labeling. *P-labeling* optimizes the processing of consecutive child axis traversals or consecutive parent axis traversals. *D-labeling* is used to optimize the processing of descendant axis traversals or ancestor axis traversals. An XPath query is decomposed according to the ancestor/descendant axes and predicates, resulting in a set of XPath subqueries. Each XPath subquery is transformed into an SQL subquery which can be efficiently evaluated using P-labeling. The SQL subqueries are then combined using D-labeling to obtain the final SQL query, which can be processed using either an off-the-shelf RDBMS or the optimized join techniques proposed in the literature [3,11,17,29,30,43].

Compared to the approaches based solely on D-labeling, the SQL query generated by BLAS for a given XPath query contains fewer selections and joins. For example, BLAS uses three joins to process the query in Fig. 1 while the D-labeling approach requires eight joins. Its execution also requires fewer disk accesses and produces smaller intermediate results. For example, the need to access nodes with tag name `protein`, `classification`, `reference`, `refinfo` or `authors` by the D-labeling approach is eliminated in BLAS.

For XML data trees that contain a large number of distinct tag names, and/or are very deep, P-labels may become very large and exceed the precision of the system. To address this challenge, we propose an *approximate* P-labeling scheme and the corresponding query translation algorithm. This extension captures a spectrum of XPath-to-SQL query translation schemes, ranging from existing schemes that do not use P-labels [3,11,17,20,29,30,34,43,44] to the one that uses exact P-labels.

The contributions and outline of this paper are:

- The BLAS system, based on P-labeling and D-labeling, is a generic framework for XML data storage and query processing (Section 2).
- P-labeling is a novel labeling scheme for processing consecutive child axis traversals or parent axis traversals (Section 3).
- Algorithms for rewriting an input XPath query to an efficient SQL query are presented (Section 4).

- We propose approximate P-labels and present a spectrum of XPath-to-SQL query translation alternatives (Section 5).
- Experimental results show the efficiency of our approach (Section 6).

After discussing related work in Section 7, we conclude in Section 8.

2. Background and system architecture

Query languages: XML trees can be traversed using XPath. The main construct of XPath is a path expression that selects a set of nodes relative to a context node. A path expression consists of a sequence of steps. Each step can have three parts: an axis that defines a node relationship with respect to the XML tree, a node test that can specify the names of the nodes to be selected, and (optionally) one or more predicates (or branches) denoted as [...], each of which can be an XPath expression. A wildcard "*" can be used to select nodes of any name. The node test in the final step of an XPath expression defines the *return node*. The number of axes in a query is called its *length*. We use "path expression" and "query" interchangeably.

Queries without predicates are called *path queries*. We also define *suffix path queries*, a subset of path queries, as follows.

Definition 2.1. A *suffix path expression* is a path expression P which optionally begins with a descendant axis step (`//`),[3] followed by zero or more child axis steps (`/`).

A *simple path expression*, which only contains child axis steps, is a special type of suffix path expression.

For example, `//protein/name` is a suffix path expression, whereas `/ProteinDatabase/ProteinEntry/protein/name` is a simple path expression.

Definition 2.2. The *source path* of a node x in an XML tree T, denoted as $SP(x)$, is the simple path expression P defined by the data path from the root to x in T.

Definition 2.3. Let $[\![P]\!]$ denote the set of XML nodes obtained by evaluating path expression P.[4] A path expression P is *contained* in a path expression Q, denoted as $P \subseteq Q$, if and only if for a context node in any XML tree T, $[\![P]\!] \subseteq [\![Q]\!]$.

Path expressions P and Q are *non-overlapping*, denoted as $P \cap Q = \emptyset$, if and only if for a context node in any XML tree T, $[\![P]\!] \cap [\![Q]\!] = \emptyset$.

Evaluating a suffix path query Q entails finding all the nodes x such that $SP(x) \subseteq Q$. Notice that a simple path expression P is contained in a suffix path expression Q if and only if P has a suffix Q excluding the leading "//". Therefore the evaluation of a suffix path query Q yields all the XML nodes whose source paths have a suffix Q.

System architecture: BLAS consists of three components—a data loader, a query translator, and a query engine—as shown in Fig. 3. As discussed earlier, two labeling schemes are exploited in BLAS for translating an XPath query to an efficient SQL query: P-labels, which are used to process consecutive child (parent) axis traversals, and D-labels, which are used to process descendant (ancestor) axis traversals. We build both P-labels and D-labels on XML data nodes, and compute P-labels for user queries.

The BLAS data loader takes as input an XML document, invokes an SAX parser, and builds the P-labels and D-labels for each node in the input document. The XML nodes are stored in a table with their labels and text values. Specifically, a tuple ⟨`plabel, dlabel, data`⟩ is generated for each node in the XML tree, where `data` stores the node value if there is any (otherwise, `data` is set to null). The relation is clustered by {`plabel, dlabel`}.

The BLAS query translator rewrites an input XPath query into an SQL query. It consists of three modules: query decomposition, SQL generation, and SQL composition. The query decomposition module splits the query into a set of suffix path queries, and records the ancestor–descendant relationship between the results of these suffix path queries. For each suffix path query, the SQL generation module computes the query's P-label and generates a corresponding subquery in SQL. Finally, the SQL subqueries are combined into a final query by the SQL composition module based on D-labeling and the ancestor–descendant relationship between the suffix path query results.

We use the holistic twig join implementation as the query engine [11]. Selections over `plabel` and `data` attributes in the translated SQL queries are evaluated using B^+ tree indexes. The joins over `dlabel` attributes are evaluated in a holistic fashion using the authors' implementation of the algorithm in [11], which achieves an optimal I/O and CPU complexity for queries containing only ancestor and descendant axes.[5]

We present the data and query labeling schemes in Section 3, and the query translator in Section 4.

3. The labeling schemes

In this section, we present the D-labeling and P-labeling schemes. For simplicity, we focus the discussion on a single document. The algorithm can be easily extended to multiple documents by introducing document id information into the labeling scheme.

3.1. D-labeling

D-labeling is used to determine the ancestor–descendant relationship between two XML nodes [2,4,20,22,34,44].

[3] For conciseness, we use "//" to denote the descendant axis. In XPath [18], "//" is short for "/descendant-or-self::node()/child::".

[4] To be precise, the node set of a query result preserves the document order.

[5] Alternatively, we can also use an off-the-shelf relational database as query engine.

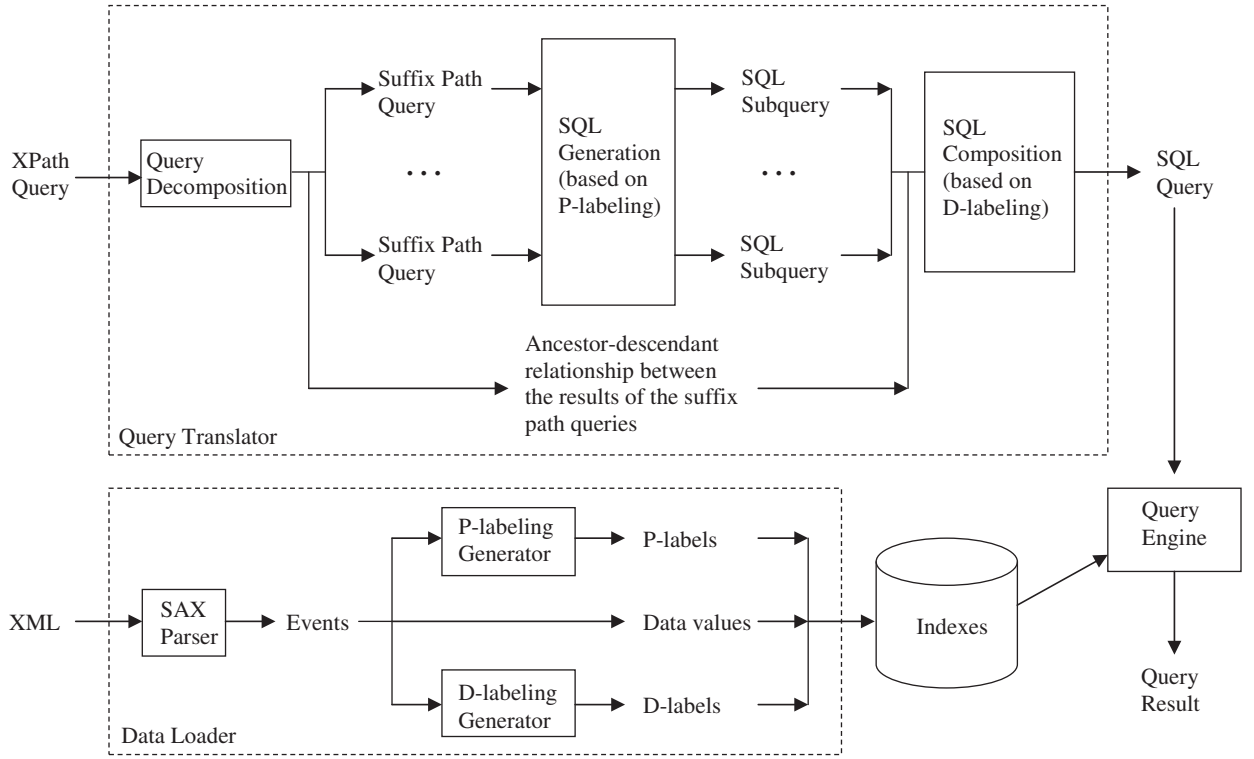

Fig. 3. The architecture of BLAS.

Definition 3.1. The *D-label* of an XML node x is a triplet $\langle d_1, d_2, d_3\rangle$, where $x.c$ denotes component c of node x's label. For any two nodes x and y, $x \neq y$, we have:

(i) Validation: $x.d_1 \leq x.d_2$.
(ii) Descendant: y is a descendant of x if and only if $x.d_1 < y.d_1$ and $x.d_2 > y.d_2$.
(iii) Child: y is a child of x if and only if y is a descendant of x and $x.d_3 + 1 = y.d_3$.
(iv) Nonoverlap: x and y have no ancestor–descendant relationship if and only if $x.d_2 < y.d_1$ or $x.d_1 > y.d_2$.

In this paper, we adopt the implementation of D-labeling suggested in [20,44]. Let d_1 and d_2 for a node x be the position of the start tag and end tag of x in the XML document, respectively, and d_3 be the *level* of x in the XML tree. The *level* of x is defined as the length of the path from the root to x. In what follows, a D-label is represented as $\langle start, end, level\rangle$. For example, in Fig. 2, considering each start tag, end tag and text value as a unit, the first node that is tagged with `classification` begins at position 7, ends at position 11, and has a level of 4.

To process queries with descendant axis traversals, such as $//t_1//t_2$, we test the ancestor–descendant relationship between nodes in list l_1 and those in list l_2 using D-labeling, where l_1 and l_2 are the lists of nodes reachable by $//t_1$ and by $//t_2$, respectively. Interpreting l_1 and l_2 as relations, this test is a join with the "descendant" property as the join predicate. We therefore call this a *D-join*.

Example 3.1. Consider the query `//ProteinDatabase //refinfo` and let `pDB` and `refinfo` be the relations which store nodes tagged by `ProteinDatabase` and `refinfo`, respectively. The D-join would be expressed in SQL as follows:

```
select pDB.start, pDB.end, refinfo.start, refinfo.end
from pDB, refinfo
where pDB.start < refinfo.start and pDB.end > refinfo.end
```

Note that using D-labeling, a child or descendant axis traversal can be processed as a join. In the next section we discuss how to reduce the number of D-joins for queries with multiple child axis steps.

3.2. P-labeling

The P-labeling scheme is designed to efficiently process consecutive child axis steps (a suffix path query).

P-labeling properties: The intuition is that each XML node x is annotated with a label according to its source path $SP(x)$ and each suffix path query Q is also assigned a P-label, such that the containment relationship between $SP(x)$ and Q can be determined by examining their labels. Since the evaluation of a suffix path query Q entails finding all the XML nodes x such that $SP(x)$ is contained in Q, it can be processed efficiently using P-labeling.

Definition 3.2. The *P-label* for a *suffix path expression* Q is an interval $[p, q)$, denoted as $I_Q = \langle p, q\rangle$, such that for any

Path expression	P-label
//name	$< 4 \times 10^{10}, 5 \times 10^{10} >$
//protein/name	$< 4.03 \times 10^{10}, 4.04 \times 10^{10} >$
//ProteinEntry/protein/name	$< 4.0302 \times 10^{10}, 4.0303 \times 10^{10} >$
//ProteinDatabase/ProteinEntry/protein/name	$< 4.030201 \times 10^{10}, 4.030202 \times 10^{10} >$
/ProteinDatabase/ProteinEntry/protein/name	$< 4.030201 \times 10^{10}, 4.03020101 \times 10^{10} >$

Fig. 4. Illustration of interval partition.

two suffix path expressions Q, Q':

(i) Validation: $Q.p < Q.q$.
(ii) Containment: $Q \subseteq Q'$ if and only if interval I_Q is contained in $I_{Q'}$, i.e. $Q'.p \leq Q.p$ and $Q'.q \geq Q.q$.
(iii) Nonintersection: $Q \cap Q' = \emptyset$ if and only if I_Q and $I_{Q'}$ do not overlap, i.e. $Q.p > Q'.q$ or $Q.q < Q'.p$.

We also assign each XML node a P-label according to its source path's P-label. Notice that here the concept of P-label is overloaded for suffix paths and XML nodes.

Definition 3.3. For an XML node x, let the P-label of its source path $SP(x)$ be $\langle p, q \rangle$. Then the *P-label* for node x, denoted as *x.plabel*, is p.

Using the P-label definition for suffix paths (Definition 3.2) and the P-label definition for XML nodes (Definition 3.3), a suffix path query Q can be evaluated by selecting the set of XML nodes whose P-labels are contained in the P-label of Q.

Proposition 1. *Let Q be a suffix path query. Then $[\![Q]\!] = \{x | Q.p \leq x.plabel < Q.q\}$. Furthermore, if Q is a simple path, then $[\![Q]\!] = \{x | x.plabel = Q.p\}$.*

If we consider the P-label of a node to be an attribute in a relation, this test is essentially a select operation using "containment" on P-labels as the predicate. If we build a B^+ tree on P-labels, this can be evaluated very efficiently. The advantage of P-labeling is that we do not need to evaluate every child axis in a suffix path P; qualified nodes can be found by checking their P-labels. In contrast, using D-labeling, a total of $(l-1)$ D-joins are needed, where l is the number of axis steps in P.

P-label construction: For illustration purpose, let the *domain* of the numbers in a P-label be the integers in $[0, m)$. Let the *length* of the P-label of a suffix path expression be the number of integers contained in the P-label interval. Suppose that there are n distinct tags $(t_1, \ldots, t_n)$ in the XML data, and that there is an ordering of tags; the particular ordering used is not important and can be arbitrary. We assign "/" a *ratio* r_0, and each tag t_i a *ratio* r_i, where $1 \leq i \leq n$, such that $\sum_{i=0}^{n} r_i = 1$. Using the tag ratios and the order, the construction of P-labels for suffix path expressions can be illustrated as follows:

(i) Path // is assigned an interval (P-label) of $\langle 0, m \rangle$.
(ii) Partition the interval $\langle 0, m \rangle$ in tag order proportional to /'s ratio r_0 and t_i's ratio r_i $(1 \leq i \leq n)$. Assuming that the order of tags is $t_1, t_2, \ldots, t_n$, this means that we allocate an interval $\langle p_0, p_1 \rangle$ to / and $\langle p_i, p_{i+1} \rangle$ to each t_i, such that $(p_{i+1} - p_i)/m = r_i$ and $p_0 = 0$. Intuitively, we allocate $\langle p_0, p_1 \rangle$ to path "/", and $\langle p_i, p_{i+1} \rangle$ to suffix paths "$//t_i$".
(iii) For the interval of a path $//t_i$, we further partition it into subintervals by tags in order according to their ratios. Each path $//t_j/t_i$ (or $/t_i$), $1 \leq j \leq n$, is now assigned a subinterval, and the proportion of the length of the interval of $//t_j/t_i$ (or $/t_i$) over the length of the interval of $//t_i$ is the ratio r_j (or r_0). Intuitively, since $[\![//t_j/t_i]\!] \subseteq [\![//t_i]\!]$ and $[\![/t_i]\!] \subseteq [\![//t_i]\!]$, we partition the interval for $//t_i$ into subintervals according to the ratio of all tags t_j and the ratio for /.
(iv) Continue to partition over each subinterval to obtain P-labels for successively longer suffix paths as found in the XML data or queries.

For simplicity, in the rest of the paper we assign the same ratio for each tag, that is, $r_i = 1/(n+1)$, $1 \leq i \leq n$.[6] As an example, the partitioning procedure for $m = 10\,000$ and tags $t_1, t_2, t_3, \ldots, t_9$ is illustrated in Fig. 4. The P-label assigned to path $/t_1/t_2$ is $\langle 2100, 2110 \rangle$.

As we can see, this technique partitions the interval of a suffix path Q to subintervals and assigns these subintervals to the suffix paths that are contained in Q. Suffix paths that are non-overlapping are assigned disjoint intervals. Therefore this implementation of P-labeling is valid with respect to Definition 3.2. So far we assume that the length of a suffix path h satisfies $(n+1)^h \leq m$, where $[0, m)$ is the domain of P-labels and n is the number of distinct tag names in the XML data tree. Section 5 presents how to construct P-labels when suffix paths have lengths larger than h.

As shown in Proposition 1, to evaluate a suffix path query Q we check whether the P-label of a node is contained in Q's P-label.

[6] In our implementation, we choose $r_i = 1/k$ where $2^{k-1} \leq n+1 \leq 2^k$ so that the P-labeling computation is done by bit-operations instead of multiplications and divisions to achieve better performance. The extra encoding space is left for future new tags. Furthermore, in the case where future changes to the document may result in a larger number of tags, we can choose an even smaller r_i to reserve enough space for these tags.

Example 3.2. To construct P-labels for the sample XML data in Fig. 2, let us assume that $m = 10^{12}$ and that there are 99 tags. Each tag is assigned a ratio 0.01. Suppose the order is /, ProteinDatabase, ProteinEntry, protein, name, ... Fig. 5 shows how to construct a P-label for the suffix path P = /ProteinDatabase/ProteinEntry/protein/name. According to Definition 3.3, every node reachable by P is assigned the P-label 4.030201×10^{10}.

Now suppose we wish to evaluate the query //protein/name. First we compute its P-label, $\langle 4.03 \times 10^{10}, 4.04 \times 10^{10} \rangle$, as shown in Fig. 5. We then find all nodes x such that $4.03 \times 10^{10} \leq x.plabel < 4.04 \times 10^{10}$. Assuming that the XML nodes are stored in a relation nodes with attribute plabel, the suffix path query can be evaluated by the following SQL statement:

```
select * from nodes
where nodes.plabel ≥ 4.03 × 10^10
  and nodes.plabel < 4.04 × 10^10
```

As illustrated above, nodes with source path /Protein- Database/ProteinEntry/protein/name have a plabel 4.030201×10^{10}, and are therefore part of the answer to this query.

Note that the P-labeling construction technique discussed so far assumes that all the tags are known. However, it is clearly desirable to construct P-labels as needed when new tags are met without prior knowledge of all the tags. Algorithm 1 constructs the P-labels for an XML tree during a single traversal of the tree given only an upper bound of the number of distinct tags. For each XML node, its P-label is computed according to the P-label of its parent, its tag interval and ratio. We use Algorithm 1 to compute P-labels for both XML nodes and suffix path queries. For a suffix path Q, we return the P-label interval obtained after processing the last node in Q. If Q starts with a child axis '/', one more step needs to be performed at the end to refine the interval according to the ratio of '/'.

Algorithm 1. P-Label(Tree: T).

```
1:  Stack s
2:  for i = 1; i ≤ n; i++ do
3:    ⟨p_i, q_i⟩ = P-Label(//t_i)
      {Let i be the position of t_i in tag order.}
4:  end for
5:  push(s, ⟨0, m⟩)
6:  Depth-first search(T){
7:  if current tag is ⟨t_i⟩ then
8:    ⟨p, q⟩ = top(s)
9:    p = p_i + p * r_i
10:   q = p_i + q * r_i
11:   push(s, ⟨p, q⟩)
12:   label this node with p
13: end if
14: if current tag is ⟨/t_i⟩ then
15:   pop(s)
16: end if
17: }
```

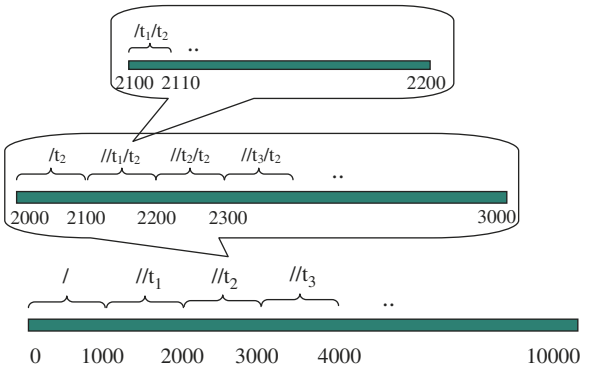

Fig. 5. P-labels for some suffix path expressions.

4. Query translation

In this section, we discuss how to translate an XPath query into an efficient SQL query using our labeling scheme.

For illustration purpose, we start with the discussion on a subset of XPath queries that contain child axes ('/'), descendant axes ('//') and predicates. This subset of XPath expressions is commonly referred to as *tree queries*, since they can be represented as trees. For example, the query Q in Fig. 6 is represented as the query tree in Fig. 7. We create a node for each node test in the query, and annotate the node with the tag name or wildcard. The return node title is darkened. An unannotated line between two nodes represents a child axis, and a line annotated with // represents a descendant axis. The root has an incoming edge to indicate whether the query starts with an axis / or //. If a node has more than one child then it is called a *branching point*, such as the nodes tagged with ProteinEntry and refinfo. If the return node is not a leaf then it is also called a branching point.

Processing general XPath queries will be discussed in Section 4.3.

4.1. Split algorithm

The Split algorithm splits the input query tree into one or more suffix path queries. This is done in two steps: *descendent axis elimination* and *branch elimination*. These two steps can be performed in any order or can be interleaved.

Descendant axis elimination (Algorithm 2) performs a depth-first traversal on the input query tree, and splits any descendent axis form $p//q$ into p and $//q$. Branch elimination (Algorithm 3) performs a depth-first traversal on the query tree and splits any branch form $p[q_1 \text{ and } q_2 \text{ and} \ldots \text{ and } q_l]/r$ into $p, //q_1, //q_2, \ldots, //q_l, //r$. The function answer (line 9) is an abstract function. If Q contains branching points, branch elimination is invoked; if Q contains descendant axes, descendant elimination is invoked; or if Q is a suffix path query, it is translated to a selection and evaluated using P-labeling. A *D-join* is then

Q = /ProteinDatabase/ProteinEntry[./protein//superfamily="cytochrome c"]/reference
/refinfo[.//author = "Evans, M.J." and ./year = "2001"]/title

Fig. 6. An XPath query Q.

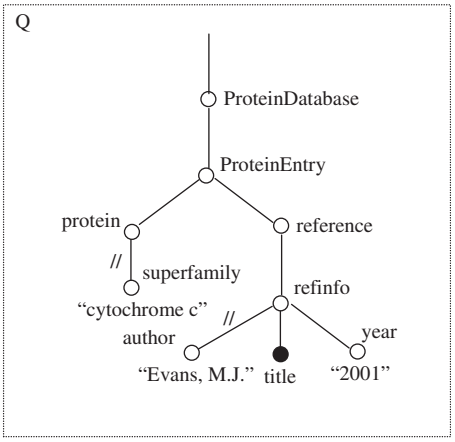

Fig. 7. Query tree of *Q*.

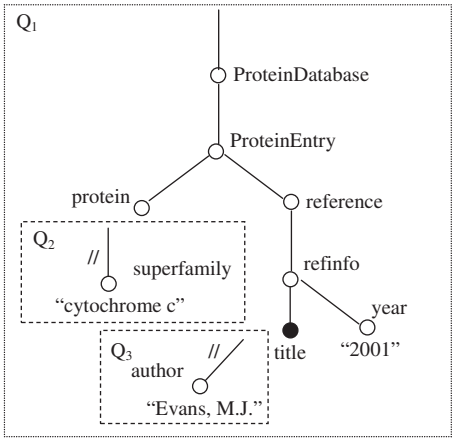

Fig. 8. Descendant axis elimination for *Q*.

used to join intermediate results by their D-labels as discussed in Section 3.1. The required level difference between intermediate results is specified in the where clause of the generated SQL statement.

Algorithm 2. D-elimination(query tree *Q*).

```
List intermediate-result
Depth-first search(Q){
if current node reached by a // edge then
  Q′ = the subtree rooted at the current // edge
  Cut Q′ from Q;
  intermediate-result.add(D-elimination(Q′), //)
end if
}
result = answer(Q)
for all r in intermediate-result do
  result = D-join(result,r)
end for
return result
```

Algorithm 3. B-elimination(query tree *Q*).

```
List intermediate-result
Depth-first search(Q){
if current node has more than one child then
  for all child of Q: Q′ do
    cut Q′ from Q
    Q′ = //Q′
    intermediate-result.add(B-elimination(Q′), the axis between
  current node and Q′)
  end for
end if
}
result = answer(Q)
for all r in intermediate-result do
  result = D-join(result,r)
end for
return result
```

Example 4.1. To translate query *Q* in Fig. 7, suppose we first eliminate descendant axes as shown in Fig. 8, generating subqueries Q_1, Q_2 and Q_3. Since Q_1 contains branching points, it is further decomposed into queries Q_4, Q_5, Q_7, Q_8 and Q_9, as shown in Fig. 9. Notice that each subquery is evaluated from the root of the XML tree, and that therefore its leading axis should be "//".[7] Each resulting subquery is a suffix path query and is evaluated as a selection on node P-labels. Suppose the evaluation of Q_4 `//ProteinDatabase/ProteinEntry` and Q_7 `//reference/refinfo` results in a list of nodes `pEntry` and `refinfo`, respectively. The lists `pEntry` and `refinfo` are D-joined as follows:

```
select pEntry.start, pEntry.end, refinfo.start, refinfo.end
from pEntry, refinfo
where pEntry.start ⟨ refinfo.start and pEntry.end ⟩
  refinfo.start and pEntry.level = refinfo.level − 2
```

Notice that we have `pEntry.level = refinfo.level - 2` in the where clause, which is different from the general D-join where no level predicate is specified. This is because the paths in Q_4 and Q_7 are connected directly in the original query (Fig. 7); therefore we choose the `pEntry` nodes returned by Q_4 that are the grandparent of one of the `refinfo` nodes returned by Q_7 rather than a general ancestor. The information about level difference can be obtained from the branch elimination procedure. The D-labels of both `pEntry` and `refinfo` are also recorded since they may be involved in D-joins with other intermediate results.

[7] Using "//" as the leading axes of subqueries potentially leads to larger intermediate results. We address this inefficiency in Section 4.2.

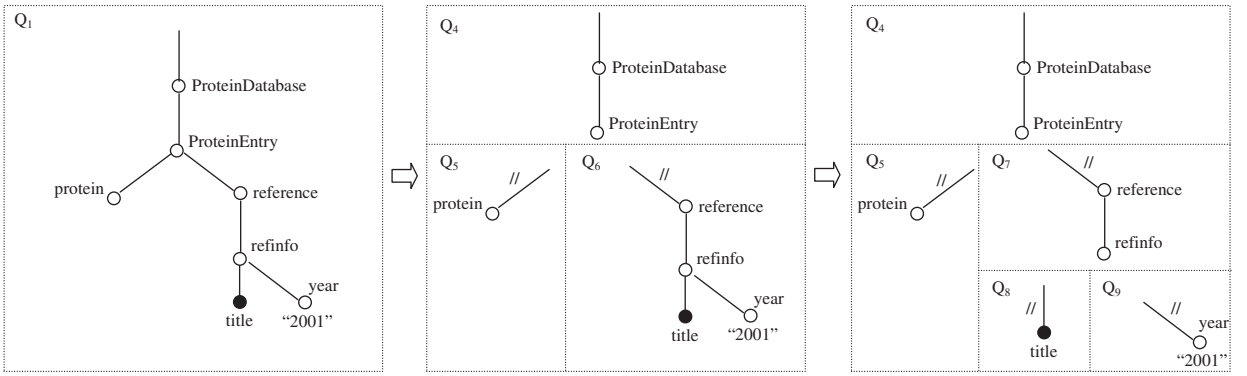

Fig. 9. Branch elimination for Q_1.

4.2. Push-up algorithm

Recall that the branch elimination step in Section 4.1 decomposes a branch form $p[q_1$ *and* q_2 *and* ... *and* $q_l]/r$ into $p, //q_1, //q_2, \ldots, //q_l, //r$. We observe that the roots of q_i $(1 \le i \le l)$ and r are children of the leaf of p. Therefore we can *push up* the path expressions q_i and r toward the root, and decompose the branch into p, $p/q_1, p/q_2, \ldots, p/q_l$, and p/r (Algorithm 4). Since p/q_i and p/r are more specific than $//q_i$ and $//r$, the number of disk accesses and the size of the intermediate results can be reduced without affecting the final query result. The key difference between Algorithms 3 and 4 is that we use a variable *SP* to record the complete path from the root of the input query tree Q to the root of a subtree Q'. Then we concatenate *SP* with Q' and evaluate SP/Q'. We call this step *push-up branch elimination*, and the whole query processing algorithm the *Push-up* algorithm.

Algorithm 4. PushUp B-elimination(query tree Q).

```
1:    return PushUp B-eliminate-sub(Q,'/');
function PushUp B-eliminate-sub(query tree Q, path expression P)
1:    List intermediate-result
2:    Boolean Path = true
3:    Depth-first search (Q){
4:    if current node x has more than one child then
5:      Path = false
6:      for all child of x: Q' do
7:        cut Q' from Q
8:      end for
9:      x.SP = P/Q
10:     for all child of x: Q' do
11:       intermediate-result.add(PushUp B-eliminate-sub(Q',SP), axis
      between Q' with x)
12:     end for
13:   end if
14:   }
15:   if Path then
16:     SP = P/Q
17:   end if
18:   result = answer(SP)
19:   for all r in intermediate-result do
20:     result = D-join(result,r)
21:   end for
22:   return result
```

In contrast to the Split algorithm, the ordering of the descendant axis elimination and push-up branch elimination in the Push-up algorithm matters in terms of performance. If we apply descendant axis elimination first, as shown in the example in Fig. 10, each SP/Q' is a suffix path expression and can be evaluated using P-labeling. This is because the input to push-up branch elimination is a subquery tree without any descendant axis steps in the middle of the query, as a result of the descendant axis elimination algorithm. Therefore *SP* is suffix path, Q' is a simple path, and their concatenation is a suffix path. However, if we apply push-up branch elimination first, the same descendant axis may be pushed up by all subquery trees below it. Although all descendant edges will eventually be cut, the descendant elimination will be invoked over the same path fragment repeatedly. We therefore apply descendant-elimination before push-up branch elimination.

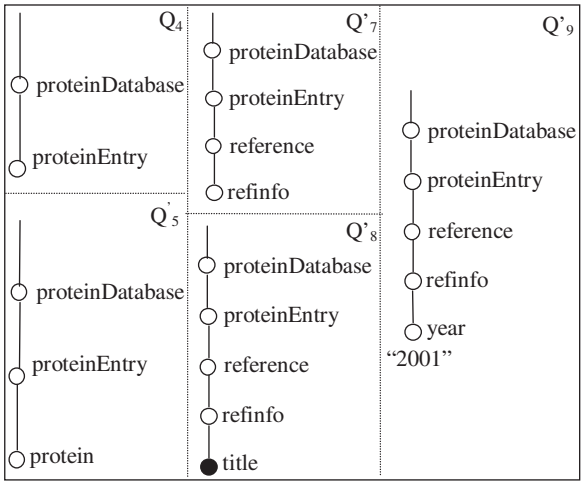

Fig. 10. Push-up branch elimination for Q_1.

4.3. Processing general XPath queries

Now we discuss how to process an XPath query with parent, child, descendant and ancestor axes. The query tree representation introduced in Section 4 can be generalized by allowing the annotation for edges to be child ('/'), parent, descendant ('//'), or ancestor. Note that

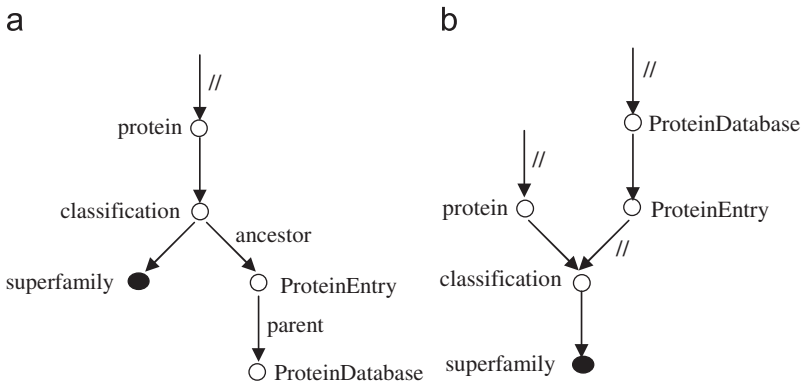

Fig. 11. Query representation. (a) A query tree. (b) The corresponding query DAG.

all the query trees are directed, and we show edge directions explicitly in this section. For example, an XPath query `/protein/classification[./ancestor::Protein Entry/parent::ProteinDatabase]/superfamily` can be represented as the query tree of Fig. 11(a). Equivalently, such a query tree can be converted to a directed acyclic graph (DAG) with the same set of nodes, but with edge annotations which are child or descendant [6]. The query DAG G is obtained from a query tree T as follows. First, the edges in a query tree T annotated with / or // become edges in the G. For each edge in T annotated with parent, create an edge connecting the same pair of nodes in the reverse direction and annotated with /. Similarly, edges annotated with ancestor are reversed and annotated with //. Finally, for any node in G that has no incoming edges, add an edge annotated with // as its incoming edge. As we see, there can be more than one root in a query DAG. For example, the sample query above can be represented as the DAG shown in Fig. 11(b).

To decompose a query DAG to suffix path queries, first we apply Algorithm 2 to eliminate descendant axes. Then we invoke Algorithm 5 to remove branches. Notice that there are several salient differences between Algorithms 5 and 4.

Algorithm 5. PushUp B-elimination(query DAG Q).

```
1:  List nodes = Topological-sort(Q)
2:  for i = 0; i<length(nodes); i++ do
3:    node x = nodes[i]
4:    if x has no parent node then
5:      x.SP = concatenate(x.axis, x.tag)
6:      x.JP = ε
7:    end if
8:    if x is a branching point or a leaf then
9:      for all parent of x:x' do
10:       generate a selection for suffix path: concatenation(x'.SP,
      x.axis, x.tag)
11:     end for
12:     x.query = join all the selections generated
13:   end if
14:   for all child of x:x' do
15:     if length(x'.SP) < length(x.sp) +1 then
16:       x'.SP = concatenation(x.SP, x'.axis, x'.tag)
17:       if x is a branching point then
18:         x'.JP = x
19:       else
20:         x'.JP = x.JP
21:       end if
22:     end if
23:   end for
24: end for
25: return the join of all the branching or leaf nodes x.query with
    x.JP.query if exists
```

First, since a node can have more than one parent, there can be several incoming suffix paths to which we can push up the node. For example, for the node with tag `classification`, there are two possible incoming suffix paths: `//protein/classification` and `//classification`. We should choose the most specific suffix path to decrease the number of disk accesses and the size of intermediate results. If we have statistics about the selectivity of each suffix path, we choose the most selective one. Otherwise, we choose the longest one based on the heuristic that a longer path is more selective.

In Algorithm 5, to obtain the longest suffix path of each node, we first topologically sort the nodes in Q and generate a list of nodes such that each ancestor of a node appears before it in the list. For each node x in the list, we then compute its longest incoming suffix path $x.SP$ in one scan of the node list. If x does not have a parent, we initialize $x.SP$ to be its tag (lines 4–7). Then for each of x's children, x', if the concatenation of $x.SP$ and the tag of x' is longer than the current $x'.SP$, we replace $x'.SP$ with the concatenated path (lines 15–16).

Second, during the node list scan, we record for each node x in Q the nearest branching point $x.JP$ (lines 17–21). Note that in a query DAG, a node is a branching point if it has more than one incoming edge, more than one outgoing edge, or is the return node.

Third, if a node is a branching point or a leaf, we build an SQL subquery, $x.query$, according to its incoming suffix paths (lines 8–13).

Finally, for each branching point or leaf node x, we join its associated subquery, $x.query$, with its nearest branching point's subquery, $x.JP.query$, according to the axis between them to form the final SQL query.

Example 4.2. Consider the query in Fig. 11. First, we apply Algorithm 2 and get two subqueries: Q_1 `//Protein Database/ProteinEntry` and Q_2 `//protein/classi-`

Table 1
Axes and corresponding conditions on D-labels.

Node relationship	Conditions on D-labels
$y \in$ following(x)	$x.d_2 < y.d_1$
$y \in$ preceding(x)	$y.d_2 < x.d_1$
$y \in$ following-sibling(x)	following(x,y), parent(y) = parent(x)
$y \in$ preceding-sibling(x)	preceding(x,y), parent(y) = parent(x)
$y \in$ descendant-or-self(x)	$x.d_1 \le y.d_1$, $x.d_2 \ge y.d_2$
$y \in$ ancestor-or-self(x)	$x.d_1 \ge y.d_1$ and $x.d_2 \le y.d_2$.

`fication[./ancestor::*]/superfamily`. Note that node `classification` is a branching point with two parents in the original query; we use `*` to denote the one that has been cut by descendant axis elimination. Q_1 does not have branching points, so we apply Algorithm 5 on Q_2.

We first obtain a list of query nodes in their topological order: `protein, *, classification, superfamily`. For the first node in the list, `protein`, we obtain $SP =$ `//protein` (line 5), $JP = \varepsilon$ (line 6). For its child `classification`, we have $SP =$ `//protein/classification` (line 16), $JP = \varepsilon$ (line 20). Then we process node `*` and obtain $SP = JP = \varepsilon$. After processing the branching point `classification`, we have: `classification`.query = `//protein/classification` $\bowtie$ `//classification` (lines 9–12). For its child `superfamily`, we have $SP =$ `//protein/classification/superfamily` (line 16), $JP =$ `classification` (line 18). After processing the last node, `superfamily`, we get: `superfamily`.query = `//protein/classification/superfamily` (lines 9–12). Finally, the query translated from Q_2 is `superfamily`.query $\bowtie$ `classification`.query (line 25).

Other XPath axes can be evaluated as joins on D-labels using the techniques proposed in previous work [29]. The join conditions are listed in Table 1. Attributes are treated as a special type of children whose names start with '@'.

5. Approximate P-labeling

In Section 3, we discussed P-label construction for suffix paths whose lengths h satisfy $(n+1)^h \le m$, where $[0,m)$ is the domain size and n is the number of distinct tags appear in the XML data tree. In this section, we discuss how to extend P-labels to approximate P-labels which encode suffix paths of any lengths. We start by discussing how to construct approximate P-labels for XML nodes. We then discuss how to compute approximate P-labels for suffix path queries, and how to translate an XPath query correctly to an SQL query based on approximate P-labels.

5.1. Approximate P-labels for XML nodes

In exact P-labeling, we associate with each XML node a label which encodes the source path of the node. We do the same in approximate P-labeling for short source paths. However, if a source path is long and an exact P-label cannot be computed within the given domain, we assign an approximate P-label to the XML node. A k-approximate P-label encodes the suffix path of length k which enters the data node. More precisely, let the source path of an XML node x be $/t_s/t_{s-1}/\ldots/t_1$. The k-approximate P-label of x is the P-label for the path $//t_k/\ldots/t_1$. If we think of nodes as being grouped into equivalence classes according to their P-labels, then accurate P-labeling groups nodes according to their source paths, while k-approximate P-labeling groups nodes according to the paths of length k entering the nodes.

How should the level of approximation k for a given domain be chosen? If the given domain is $[0,m)$, we set k so that $(n+1)^k \le m < (n+1)^{k+1}$. Note that if $k \ge h$, then the P-labels for all XML nodes are accurate. To compute k-approximate P-labels, we use Algorithm 1.

Proposition 2. *Algorithm* 1 *computes k-approximate P-labels for nodes, where the domain for P-labels is* $[0,m)$ *and* $(n+1)^k \le m < (n+1)^{k+1}$.

5.2. Query translation for approximate P-labeling

We begin with suffix path query evaluation using k-approximate P-labeling, and then discuss how to evaluate a general XPath query.

Since k-approximate P-labeling does not affect the P-labels for suffix paths of length $\le k$, we can restate Proposition 1 as:

Corollary 5.1. *Given a k-approximate P-labeling scheme, let Q be a suffix path query of length* $\le k$. *Then*

$$[\![Q]\!] = \{x | Q.p_1 \le x.plabel < Q.p_2\}$$

Furthermore, if Q is a simple path of length $\le k$, *then*:

$$[\![Q]\!] = \{x | Q.p_1 = x.plabel\}$$

To evaluate a suffix path query $Q : \alpha t_s/\ldots/t_1$, ... where $\alpha \in \{/, //\}$ and $s > k$, first we cut Q into shorter suffix paths, such that each of them has a length $\le k$ and therefore can be evaluated according to Corollary 5.1. Specifically, let g be an integer such that $g * k < s \le (g+1) * k$, and cut Q into a set of suffix paths: $Q_1 : //t_k/\ldots/t_1, \ldots, Q_g : //t_{g*k}/\ldots/t_{(g-1)*k+1}$, $Q_{g+1} : \alpha\, t_s/\ldots/t_{g*k+1}$, as shown in Fig. 12. Notice that we change the leading child axis to a descendant axis in each suffix path query Q_j, $1 \le j \le g$. We then retrieve the set of XML nodes reachable by Q_i ($[\![Q_i]\!]$), $1 \le i \le (g+1)$, by selecting all the nodes whose P-label is contained in the P-label of Q_i according to Corollary 5.1. Finally, we stitch together the query results of the Q_i's to compute the results of query Q based on the following proposition.

Proposition 3. *Let* Q_1 *be a simple path expression and* Q_2 *be a suffix path expression. For an XML node* $x \in [\![//Q_1]\!]$, *we have* $x \in [\![Q_2/Q_1]\!]$ *if and only if x has an ancestor y such that* $y \in [\![Q_2]\!]$ *and the depth difference between y and x is equal to the length of* Q_2.

Recall that D-labeling allows us to detect the ancestor–descendant relationship between XML nodes. To evaluate the suffix path query Q, we D-join the inter-

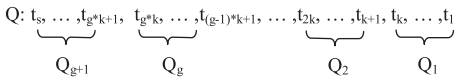

Fig. 12. Illustration for suffix path query decomposition.

mediate results $[\![Q_i]\!]$ with $[\![Q_{i+1}]\!]$, where $1 \le i \le g$, with a level difference of k as part of the join condition.

Example 5.2. Continuing Example 3.2, since there are 99 distinct tags, the domain of P-label $[0, 10^{12})$ supports a 6-approximate P-labeling scheme. To evaluate query Q: `/ProteinDatabase/ProteinEntry/reference /refinfo/xrefs/xref/db/url`, which has a length of 8, we first decompose Q into two shorter suffix paths: Q_1: `//reference/refinfo/xrefs/xref/db/url`, and Q_2: `/ProteinDatabase/ProteinEntry`. After performing a selection on P-labels to get the set of nodes reachable by Q_1 and Q_2, denoted as `url` and `PE`, respectively, we perform the following D-join to find the final query result:

```
select PE.start, PE.end, url.start, url.end
from PE, url
where PE.start ⟨url.start and PE.end⟩ url.end
  and PE.level = url.level + 6
```

To evaluate a general XPath query, we first use the algorithms presented in Section 4 to decompose the query into suffix path queries. If a suffix path query obtained from the decomposition is longer than k, we further decompose it to suffix path queries of length $\le k$. Then we translate each suffix path query to an SQL selection. Finally, the intermediate results of selections are stitched together by D-joins according to the ancestor–descendant relationship between XML nodes. The number of joins in the SQL query translated from an XPath query depends on the number of suffix paths of length $\le k$ after decomposition.

Using accurate P-labeling together with D-labeling (discussed in Section 3) and using D-labeling alone are two extreme cases of using k-approximate P-labeling together with D-labeling. Suppose that we set $k = 1$, then the P-label of each node in an XML tree encodes only the tag information of the node. The query translator in BLAS decomposes an XPath query Q to suffix path queries of length 1. In this case, the translated SQL query is the same as the previous approach using D-labeling only [2,4,20,34,44]. On the other hand, when we set $k \ge h$, where h is the height of the XML tree, the translated SQL query is the same as the one using accurate P-labeling and D-labeling. Thus, k-approximate P-labeling presents a spectrum of solutions for XPath-to-SQL query translation. To choose an appropriate value for k, note that the larger the value of k the fewer number of joins are produced in the translated SQL query. Therefore given the domain $[0, m)$ set by the machine word size to achieve maximal efficiency for P-label computation and comparison, we propose to choose the value of k such that $(n+1)^k \le m < (n+1)^{k+1}$.

	Shakespeare	Protein	Auction	TEI
Size	26MB	70MB	68MB	13.5M
Nodes	639500	2276620	1237800	1121370
Tags	19	66	77	314
Depth	7	7	12	50

Fig. 13. XML data sets.

6. Experimental evaluation

We compared the Split and Push-up approaches in BLAS with the approach of using D-labeling alone. Our experimental evaluation shows that Push-up outperforms Split, and that both have a substantial performance improvement over D-labeling.

6.1. Experimental setup

The experiments were performed on a 1.5 GHz Pentium 4 machine with 512 MB memory and one 40 GB hard disk (7200 rpm). All experiments were repeated 10 times independently on a cold cache, and the average processing time was calculated discarding the maximum and minimum values.

The data sets used are Shakespeare [10], Protein [25], Auction [1] and TEI [19], as summarized in Fig. 13. *Size* denotes the disk space used to store the original XML file. *Nodes* is the number of nodes in the XML file. *Tags* is the number of distinct tags. *Depth* is the length of the longest path in the XML tree.

We tested several types of XPath queries: suffix path queries (type 1), path queries (type 2), tree queries with child, descendant axes and predicates (type 3), and XPath queries containing parent and ancestor axes (type 4), as listed in Fig. 14. The names of the queries are encoded by "QXY", where "X" is one of "S" (Shakespeare), "P" (Protein) or "A" (Auction), and "Y" is the query type. For the Auction data set, we also tested a set of benchmark queries provided by XMark [1] listed as $Q1, \ldots, Q5$. For the TEI data set, we choose queries with length ranging from 1 to 34 to test the effectiveness of the approximate P-labeling scheme.

The XML data sets were stored in table form. We created two relations for each data set, one to implement our approach and the other to implement D-labeling. The schema of the relation used for our approach is *SP*(*plabel*, *start*, *end*, *level*, *data*), with primary key {*start*}. Attribute {*data*} stores PCDATA or attribute values. The relation is clustered by {*plabel*, *start*}. The schema of the relation *SD* implementing D-labeling is the same, except that the *plabel* attribute is replaced by a *tag* attribute. The relation is clustered by {*tag*, *start*}. Indexes are built for all the attributes involved in the queries to achieve the best possible performance for both approaches. The query engine for both approaches is based on the holistic twig join algorithm implemented in C++ [11].

QS1	/PLAYS/PLAY/ACT/SCENE/SPEECH/LINE
QS2	/PLAYS/PLAY/EPILOGUE//LINE/STAGEDIR
QS3	/PLAYS/PLAY/ACT/SCENE[./TITLE]//LINE
QS4	//PLAY//LINE[./ancestor::SCENE/parent::ACT]
QP1	/ProteinDatabase/ProteinEntry/protein/name
QP2	/ProteinDatabase/ProteinEntry//authors/author
QP3	/ProteinDatabase/ProteinEntry[./reference/refinfo[./citation and ./year]]/protein/name
QP4	/ProteinDatabase/ProteinEntry//authors[./ancestor::reference]/author
QA1	//category/description/parlist/listitem
QA2	/site/regions//item/description
QA3	/site/regions/asia/item[./shipping]/description
QA4	//item//mail[./ancestor::regions/asia]
Q1	/site/people/person[@id]/name
Q2	/site/open_auctions/open_auction/bidder/increase
Q3	/site/open_auctions/open_auction/bidder/personref[@person]
Q4	/site/closed_auctions/closedauction/price[text() $\geq$ 40]
Q5	count(/site/regions//item)

Fig. 14. Query sets.

6.2. Query processing time

Fig. 15 shows the query processing time of each approach on the Shakespeare, Protein and Auction data sets. Since the cost of output generation (XML tree reconstruction) is the same regardless of the algorithm applied, it is not included in the figure.

As we can see, for a query with l-axis steps, the conventional approach using D-labeling requires $(l-1)$ D-joins. The number of D-joins needed in BLAS depends on the number of predicates and descendant/ancestor axis steps rather than on the total number of axis steps, and therefore fewer D-joins are needed. Furthermore, BLAS does not need to access XML nodes whose source path expressions are not contained in the decomposed suffix path expressions, and therefore requires fewer disk accesses. The size of intermediate results is also smaller due to the fact that more specific SQL subqueries are generated. In particular, BLAS uses a single select operation with fewer disk accesses over the P-labels for suffix path queries (query type 1) compared with D-labeling. It uses a single D-join and two selections for the path queries tested (query type 2). Also, observe that for suffix path queries and path queries, Split and Push-up are the same: they generate the same subqueries, which are stitched together using the same number of joins, and therefore have the same execution time. As for general queries (query type 3), consider the SQL queries generated for `QS3`. D-labeling requires five D-joins, whereas BLAS only requires two D-joins. Furthermore, since Push-up restricts each subquery to be as specific as possible, it further reduces disk accesses and the size of intermediate results, and performs better than Split. For `QS3`, Split requires two range selections and one equality selection; Push-up requires one range selection and two equality selections with a smaller intermediate result generated compared with Split. Therefore Push-up has better performance than Split. Experiments show that for all test queries and data sets, BLAS is more efficient than the traditional D-labeling approach.

6.3. Scalability

To test scalability, we replicated the Auction data set. Fig. 16 shows the execution time and the number of elements read by each approach for suffix path query `QA1` on different data set sizes. As the file size increases, the difference between the execution time of D-labeling and that of BLAS increases. Performance results for tree query QA3 are shown in Fig. 17; again Split and Push-up outperform D-labeling. For this query, Push-up outperforms Split: although Push-up uses the same number of joins as Split, the select operations are more selective. Therefore the number of disk accesses is fewer and the execution time is smaller for Push-up. We also observe that the performance difference increases with the file size.

6.4. Approximate P-labeling

To test the effectiveness of approximate P-labeling, we used 6-approximate P-labeling and tested path queries of length ranging from 1 to 34 on the TEI data set. As shown in Fig. 18, when the query length increases, the time used by the Push-up algorithm increases in a stair-wise fashion.

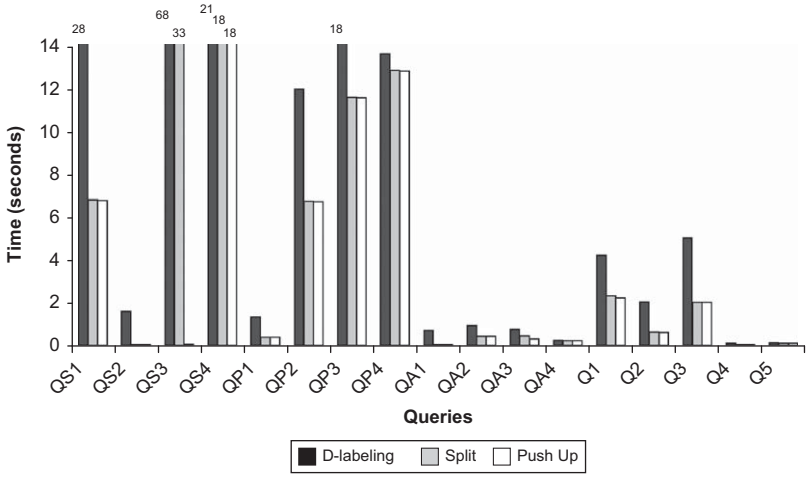

Fig. 15. Query execution time of D-labeling, Split and Push-up on different data sets.

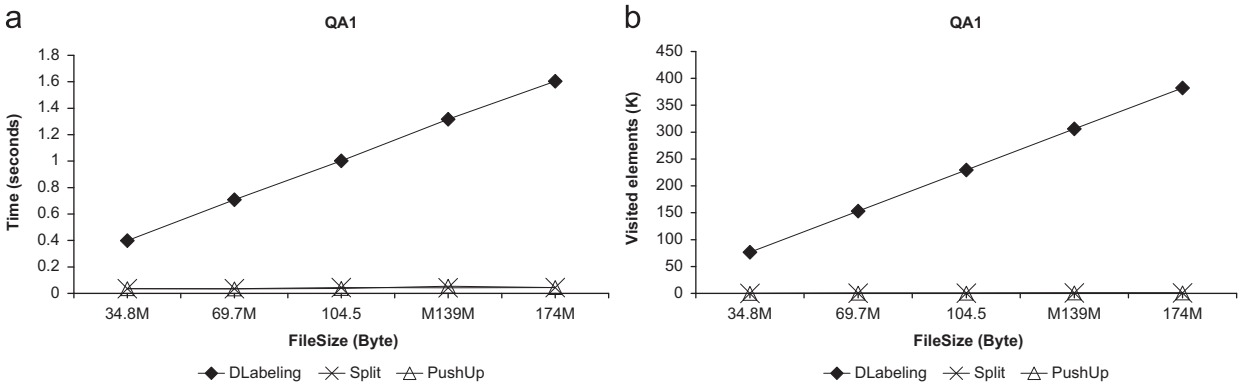

Fig. 16. The performance of D-labeling, Split and Push-up for a suffix path query. (a) Execution time. (b) Number of elements read.

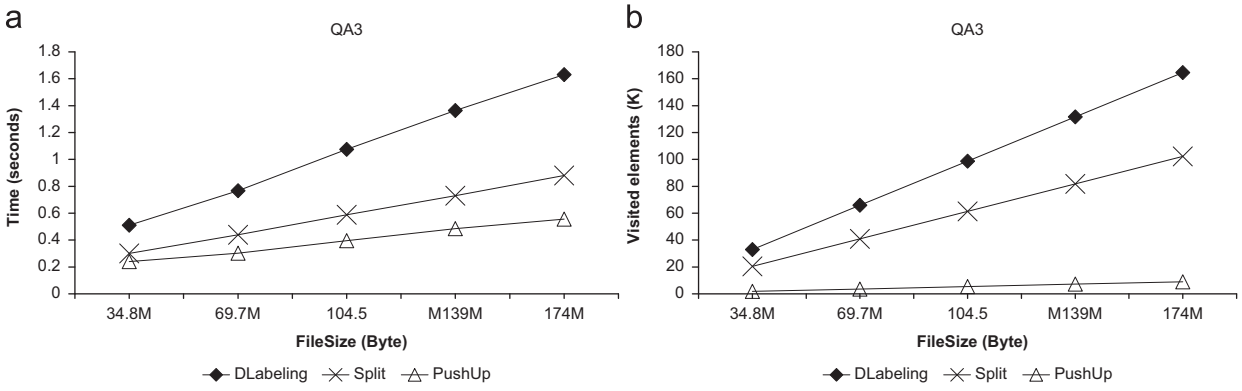

Fig. 17. The performance of D-labeling, Split and Push-up for a tree query. (a) Execution time. (b) Number of elements read.

Within each group of six queries, Push-up takes roughly the same time since the SQL queries generated for the XPath queries in the same group have the same number of selections and joins. On the other hand, since D-labeling requires one more join when the query length is increased by one, the time it takes is proportional to the length of the query.

As shown in the experiments, BLAS outperforms the traditional approach which only uses D-labeling. The performance enhancement is achieved by reducing the

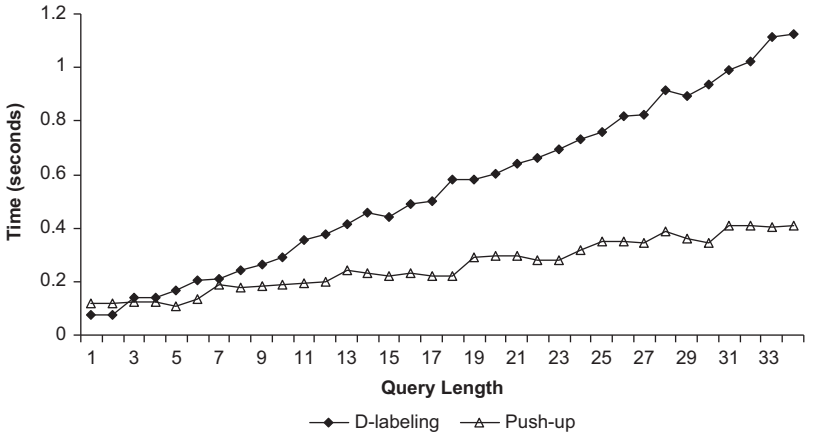

Fig. 18. Query time for TEI data set when query length increases.

number of joins and disk accesses. BLAS gracefully handles data sets with a large number of tag names and long suffix path queries using approximate P-labeling. Furthermore, Push-up should be chosen over Split as an implementation strategy for BLAS.

7. Related work

XML storage and query processing using relational databases: A lot of research has been done on storing XML data using relations and translating XML queries to SQL queries, in order to leverage the mature indexing and query processing techniques of relational databases. The "edge approach" was the first proposal, where an XML document is treated as a graph and a tuple is generated for every edge [24]. It also provides a simple and general approach to map an XML query to an SQL query. However, since a self-join is required to determine the parent–child relationship over two lists of XML nodes, an XML query is typically translated to an SQL query with many joins.

To enable efficient query processing, one theme is to reduce the number of joins in the translated SQL queries. The "inlining" approach inlines the information of unique child into its parent tuple based on the data schema [8,15,21,32,41]. Query translation in the presence of recursive data schemas has also been studied [23,33]. Compared to the edge approach, inlining eliminates the joins between a node and its unique child. However, the mapping from an XML query to an SQL query is complex and requires schema information. To process a descendant and/or ancestor axis traversal, both the edge and inlining approaches require several joins, depending on the depth of the XML tree.

D-labeling was proposed to translate either a descendant or child axis to a single join, and has been shown to reduce the number of joins as compared to the edge and inlining approaches when the XML query involves descendant axis traversals [3,11,17,20,29,30,34,43,44]. For an XML query with l-axis traversals, D-labeling requires $(l-1)$ joins over D-labels. The effectiveness of leveraging D-labeling in processing XQuery queries compared with other XQuery engines has been shown [20].

Our work further decreases the number of joins in the SQL query translated from an XML query by eliminating the joins translated from consecutive child axis traversals. For an XML query containing l-axis traversals, the number of joins that BLAS requires is equal to the number of suffix paths that are in the XML query, which is in general fewer than (and in the worst case equal to) $(l-1)$.

In Georgiadis and Vassalos's work [26], the source path of each XML node is recorded as a string and a path query is processed as a complex regular expression matching over strings. Another related approach is to record the "reversed" representation of the source path of each XML node and process a suffix path query using "prefix match" of the reversed source paths [7,39]. The results of the subqueries are joined based on node ancestor–descendant relationships using Dewey labeling or its variant [38]. While the implementation of D-labeling can be either based on node regions or Dewey labeling, selections involving integer equality/range testing on P-labels in the SQL queries generated by BLAS are more efficient than selections involving string matching.

This paper extends our earlier work [14] to consider general XPath query processing on XML data. Since accurate P-labeling presented in our earlier work cannot handle XML data with large tag sets and/or tree depth, we propose in Section 5 an approximate P-labeling scheme and an extended query translation algorithm. BLAS with approximate P-labeling presents a spectrum of XPath-to-SQL query translation algorithms, in which existing work [3,11,17,20,29,34,44] and our earlier work [14] represent two extreme cases. Furthermore, we extend our previous work to support all XPath axes in Section 4.3, rather than just child and descendant axes.

There are several orthogonal research themes in speeding up the performance of XML query processing. Since the join operation based on D-labeling or Dewey-labeling is common in the translated SQL queries, the problem of efficiently implementing this type of join has been extensively studied [3,17,29,34,44]. To process tree queries that contain only descendant axes, algorithms with time complexity linear in the size of the input and output have been proposed [11]. Specialized indexes for optimizing joins over D-labels have also been designed

[12,17,30,43]. The problem of optimizing joins translated from XQuery queries was studied [9], such as handling joins embedded in nested for-loops, recognizing join patterns in a way that is immune to syntactic variance in the query, and avoiding expensive sorting operations. Problems such as generating pipelineable plans and picking the right physical organization, have also been studied [27]. Schema information has been exploited so that the wildcards and descendant axes in XPath queries can be expanded to speed up evaluation [35]. Techniques for generating an information-preserving mapping scheme from XML to relational databases that satisfies losslessness and validation properties have also been proposed [5]. A survey on how to efficiently query XML data repositories was done by Gou and Chirkova [28].

Labeling schemes: Several D-labeling implementations have been proposed [2,4,22,34,36]. The problem of building D-labels with the smallest label size has been studied [2,4]. XPRESS [37] proposes an XML data compression technique that uses reverse arithmetic encoding to encode paths as a distinct interval within [0,1). Our P-labeling borrows the idea of labeling a path, but focuses on the optimization of query processing instead of compression. During P-label construction, the intervals are partitioned uniformly from an integer domain. In Kannan et al. [31], labels of size logarithmic in the data size have been designed to encode a graph structure, such that testing the adjacency of any two nodes can be performed in time linear in the size of the labels. Techniques for dynamically updating D-labels when the underlying XML data is updated have also been proposed [16,38,42].

As with approximate P-labeling, the A(k)-index [32] and D(k)-index [40] exploit the idea of designing an approximate index for paths of length k entering data nodes. However, there are several salient differences. First, an A(k) or D(k)-index is a structural summary of a data graph, and a query is evaluated by traversing the index graph. In BLAS, by augmenting XML data with P-labels and D-labels, a query is evaluated by accessing B+-trees that index integer labels. Second, an A(k) or D(k)-index does not support queries containing predicates, and it provides complete but not sound answers for queries containing descendant axes. In contrast, BLAS supports those types of queries correctly. On the other hand, an A(k) or D(k)-index can handle graph-structured XML data, while BLAS processes XML trees.

8. Conclusions

This paper presents BLAS as a generic and efficient system for XML storage and XPath query processing by leveraging relational databases. It maps XML data to relations and translates an input XPath query to an SQL query. BLAS has several advantages compared to existing work. First, the number of selections in the translated SQL query is reduced. The number of selections required by BLAS is equal to the number of suffix paths in an XPath query, while the number of selections required in existing work is equal to the number of axis steps in the query. Second, as a consequence of reducing the number of selections, the number of disk accesses required to execute the SQL query is reduced. Third, BLAS requires fewer joins; the number of joins is one less than the number of selections in the generated SQL query. Finally, the intermediate results that participate in joins in BLAS are smaller since the SQL subqueries generated in BLAS are longer and more specific.

To achieve these benefits, BLAS uses a bi-labeling scheme which consists of P-labeling to speed up suffix path query evaluation, and D-labeling to speed up queries involving descendant axis traversals. To translate an XPath query to an SQL query, we first decompose an XPath query into a set of suffix path subqueries. P-labels of these subqueries are then calculated, based on which each suffix path subquery is translated to an SQL selection. The final SQL query is obtained by stitching together the set of selections using joins on D-labels.

We also propose approximate P-labeling and corresponding query translation algorithm when we run out of enough precision to compute exact P-labels. Approximate P-labeling represents a spectrum of XPath-to-SQL query translation, in which BLAS with exact P-labeling and existing work that only uses D-labeling represent two end points.

BLAS efficiently evaluates XPath expressions, which are a building block of XQuery. We are extending BLAS to support XQuery by addressing the additional technical challenges of evaluating FLWOR expressions (For, Let, Where, Order By and Return clauses).

References

[1] XMARK the XML-benchmark project ⟨http://monetdb.cwi.nl/xml/index.html⟩.
[2] S. Abiteboul, H. Kaplan, T. Milo, Compact labeling schemes for ancestor queries, in: Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001, pp. 547–556.
[3] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu, Structural joins: a primitive for efficient XML query pattern matching, in: Proceedings of International Conference on Data Engineering (ICDE), 2002, pp. 141–154.
[4] S. Alstrup, T. Rauhe, Improved labeling scheme for ancestor queries, in: Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 947–953.
[5] D. Barbosa, J. Freire, A.O. Mendelzon, Designing information-preserving mapping schemes for XML, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2005, pp. 109–120.
[6] C. Barton, P. Charles, D. Goyal, M. Raghavachari, V. Josifovski, M. Fontoura, Streaming XPath processing with forward and backward axes, in: Proceedings of International Conference on Data Engineering (ICDE), 2003, pp. 455–466.
[7] K. Beyer, R.J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B.V. Linden, B. Vickery, C. Zhang, System RX: one part relational, one part XML, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2005, pp. 347–358.
[8] P. Bohannon, J. Freire, P. Roy, J. Simeon, From XML schema to relations: a cost-based approach to XML storage, in: Proceedings of International Conference on Data Engineering (ICDE), 2002, pp. 64–80.
[9] P. Boncz, T. Grust, M.V. Keulen, S. Manegold, J. Rittinger, J. Teubner, MonetDB/XQuery: a fast XQuery processor powered by a relational engine, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2006, pp. 479–490.
[10] J. Bosak, Shakespeare ⟨http://www.ibiblio.org/xml/examples/shakespeare/⟩.

[11] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2002, pp. 310–321.
[12] T. Chen, J. Lu, T.W. Ling, On boosting holism in XML twig pattern matching using structural indexing techniques, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2005, pp. 455–466.
[13] Y. Chen, S. Davidson, C. Hara, Y. Zheng, RRXS: redundancy reducing XML storage in relations, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2003, pp. 189–200.
[14] Y. Chen, S. Davidson, Y. Zheng, BLAS: an efficient XPath processing system, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2004, pp. 47–58.
[15] Y. Chen, S.B. Davidson, Y. Zheng, Constraint preserving XML storage in relations, in: Proceedings of International Workshop on the Web and Databases (WebDB), 2002, pp. 7–12.
[16] Y. Chen, G.A. Mihaila, R. Bordawekar, S. Padmanabhan, L-tree: a dynamic labeling structure for ordered XML data, in: Lecture Notes in Computer Science, EDBT Workshop: Database Technologies for Handling XML-Information on the Web (DataX), vol. 3268, 2004, pp. 209–218.
[17] S. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, C. Zaniolo, Efficient structural joins on indexed XML documents, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2002, pp. 263–274.
[18] J. Clark, S. DeRose, XML path language (XPath) ⟨http://www.w3.org/TR/xpath⟩.
[19] The TEI Consortium, Text Encoding Initiative—The XML Version of the TEI Guidelines ⟨http://www.tei-c.org/Consortium⟩.
[20] D. DeHaan, D. Toman, M. Consens, M.T. Ozsu, A comprehensive XQuery to SQL translation using dynamic interval encoding, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2003, pp. 623–634.
[21] A. Deutsch, M. Fernandez, D. Suciu, Storing semistructured data with STORED, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 1999, pp. 431–442.
[22] P.F. Dietz, Maintaining order in a linked list, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 1982, pp. 62–69.
[23] W. Fan, J.X. Yu, H. Lu, J. Lu, R. Rastogi, Query translation from XPATH to SQL in the presence of recursive DTDs, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2005, pp. 337–348.
[24] D. Florescu, D. Kossmann, Storing and querying XML data using an RDBMS, IEEE Data Engineering Bulletin 22 (3) (1999) 27–34.
[25] Georgetown Protein Information Resource, Protein Sequence Database ⟨http://www.cs.washington.edu/research/xmldatasets/⟩.
[26] H. Georgiadis, V. Vassalos, Improving the efficiency of XPath execution on relational systems, in: Proceedings of International Conference on Extending Database Technology (EDBT), 2006, pp. 570–587.
[27] H. Georgiadis, V. Vassalos, XPath on steroids: exploiting relational engines for XPath performance, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2007, pp. 317–328.
[28] G. Gou, R. Chirkova, Efficiently querying large XML data repositories: a survey, IEEE Transactions on Knowledge and Data Engineering 19 (10) (2007) 1381–1403.
[29] T. Grust, M.V. Keulen, J. Teubner, Accelerating XPath evaluation in any RDBMS, ACM Transactions on Database Systems (TODS) 29 (1) (2004) 91–131.
[30] H. Jiang, H. Lu, W. Wang, B.C. Ooi, XR-tree: indexing XML data for efficient structural joins, in: Proceedings of International Conference on Data Engineering (ICDE), 2003, pp. 253–263.
[31] S. Kannan, M. Naor, S. Rudich, Implicit representation of graphs, in: Proceedings of ACM Symposium on Theory of Computing (STOC), 1988, pp. 334–343.
[32] R. Kaushik, P. Shenoy, P. Bohannon, E. Gudes, Exploiting local similarity for efficient indexing of paths in graph structured data, in: Proceedings of International Conference on Data Engineering (ICDE), 2002, pp. 129–140.
[33] R. Krishnamurthy, V.T. Chakaravarthy, R. Kaushik, J.F. Naughton, Recursive XML Schemas, Recursive XML queries, relational storage: XML-to-SQL query translation, in: Proceedings of International Conference on Data Engineering (ICDE), 2004, pp. 42–53.
[34] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2001, pp. 361–370.
[35] Z.H. Liu, M. Krishnaprasad, V. Arora, Native XQuery processing in Oracle XMLDB, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2005, pp. 828–833.
[36] J. Lu, T.W. Ling, C.Y. Chan, T. Chen, From region encoding to extended Dewey: on efficient processing of XML twig pattern matching, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2005, pp. 193–204.
[37] J. Min, M. Park, C. Chung, XPRESS: a queriable compression for XML data, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2003, pp. 122–133.
[38] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury, ORDPATHs: insert-friendly XML node labels, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2004, pp. 903–908.
[39] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, V. Zolotov, Indexing XML data stored in a relational database, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 2004, pp. 1134–1145.
[40] C. Qun, A. Lim, K.W. Ong, D(k)-index: an adaptive structural summary for graph-structured data, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2003, pp. 134–144.
[41] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, J.F. Naughton, Relational databases for querying XML documents: limitations and opportunities, in: Proceedings of International Conference on Very Large Data Bases (VLDB), 1999, pp. 302–314.
[42] A. Silberstein, H. He, K. Yi, J. Yang, BOXes: efficient maintenance of order-based labeling for dynamic XML data, in: Proceedings of International Conference on Data Engineering (ICDE), 2005, pp. 285–296.
[43] W. Wang, H. Jiang, H. Lu, J.X. Yu, PBiTree coding and efficient processing of containment joins, in: Proceedings of International Conference on Data Engineering (ICDE), 2003, pp. 391–404.
[44] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, G.M. Lohman, On supporting containment queries in relational database management systems, in: Proceedings of ACM SIGMOD International Conference on Management of Data, 2001, pp. 425–436.