

# BLAS : An Efficient XPath Processing System

Yi Chen  
University of Pennsylvania  
yicn@cis.upenn.edu

Susan B. Davidson  
University of Pennsylvania and  
INRIA-FUTURS (France)  
susan@cis.upenn.edu

Yifeng Zheng  
University of Pennsylvania  
yifeng@cis.upenn.edu

## ABSTRACT

We present *BLAS*, a Bi-Labeling based System, for efficiently processing complex XPath queries over XML data. *BLAS* uses *P-labeling* to process queries involving consecutive child axes, and *D-labeling* to process queries involving descendant axes traversal. The XML data is stored in labeled form, and indexed to optimize descendant axis traversals. Three algorithms are presented for translating complex XPath queries to SQL expressions, and two alternate query engines are provided. Experimental results demonstrate that the *BLAS* system has a substantial performance improvement compared to traditional XPath processing using *D-labeling*.

## 1. INTRODUCTION

XML is rapidly emerging as the de facto standard for exchanging data on the Web. Due to its complex, tree-like structure, languages for querying XML are based on path navigation (e.g. XPath [11]), and typically include the ability to traverse from a given node to a child node (the “child axis”) or from a given node to a descendant node (the “descendant axis”). XML query languages also give the ability to qualify traversals based on branches, wildcards and value predicates.

As an example, suppose a biologist is interested in proteins belonging to the “cytochrome c” family. He remembers that Dr. Evans, M.J. wrote an important paper about this family of proteins in 2001, but cannot remember the title of the paper. Using the XML protein repository shown in figure 1, the XPath query shown in figure 2 could be used to retrieve the desired information.

Since data sets may be large and complex, efficiently querying XML data is a major concern. To address this problem, several techniques for storing XML in relational databases have been proposed in order to leverage the power of relational technology for query processing. The simplest and most generic proposal is to treat an XML document as a graph [17], and generate a tuple for every XML node with its parent identifier. In this way, the parent-child relationship over two lists of XML nodes can be found out by a join. To reduce the number of joins required to answer an XML query, techniques which depend on a schema graph [27, 4, 15, 8, 7] have been proposed to inline each distinct child into the parent tu-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2004 June 13-18, 2004, Paris, France.  
Copyright 2004 ACM 1-58113-859-8/04/06 . . . \$5.00.

```
<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>
```

Figure 1: Sample XML protein repository

ple. However, in general many joins are needed to evaluate a single descendant axis. Furthermore, we have to rely on auxiliary code in some general-purpose program language together with SQL to express some XML queries (for example, descendant axis).

Recently, a labeling technique [16, 23, 1, 3, 31, 13] was proposed to efficiently handle descendant axis traversal. In this paper, we refer to the technique as *D-labeling*, where 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 intervals. The level of the node is also used to distinguish the parent-child relationship from the ancestor-descendant relationship. In this way, either a descendant or child axis can be processed by one join, and no auxiliary code except SQL is necessary to express an XML query. [13] shows the effectiveness of XQuery processing using *D-labeling* compared with other XQuery implementations.

However, *D-labeling* is not efficient for complex queries such as that in figure 2. To evaluate this query using *D-labeling*, we first find all nodes tagged with *proteinDatabase* and all nodes tagged with *proteinEntry*, and join them according to their *D-labels*. We must then join the result with all nodes tagged with *protein*, and so on. Essentially, any two tags connected by a descendant axis, child

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

Figure 2: Sample XPath Query Q

axis, or a branch will involve a join. Thus in our example a total of 8 joins are needed. Furthermore, all nodes whose tags appear in the query must be visited to answer the query.

Since joins are used extensively for query processing with D-labeling, [2, 6, 9, 10, 19, 29, 20] have proposed several new techniques to optimize this type of join. These techniques yield a significant performance improvement over the joins of a relational database. However, as seen in the example above, the sheer number of joins and disk accesses needed renders D-labeling inefficient for complex queries even using these techniques. Since the primary bottleneck for evaluating complex queries efficiently is the number of joins and disk accesses, in this paper we therefore address the problem of reducing the number of joins and disk accesses required for complex XPath queries, as well as optimizing the join operations.

Motivated by the compression scheme of XPRESS [26], we propose a labeling scheme called *P-labeling* (P stands for path) which optimizes an important class of queries called *suffix path queries*. Suffix path queries start with an optional descendant axis step followed by zero or more child axis steps.

Based on P-labeling and D-labeling, we build a system called BLAS (a Bi-LABELing based System) to efficiently process XPath queries which can be represented as trees. BLAS is composed of three parts: an index generator, a query translator and a query engine. The index generator stores the P-labeling, D-labeling as well as data values of an XML document. The query translator decomposes an XPath query into a set of suffix path queries, encodes each suffix path query using P-labeling, generates a corresponding SQL query for each suffix path query, and finally composes the SQL subqueries into a complete SQL query plan using D-labeling. The query engine can be either a RDBMS or can use the optimized join techniques of [2, 6, 9, 10, 19, 29, 20]. In particular, we focus on the holistic twig join of [6].

We propose and evaluate three query translation algorithms to translate a complex XPath query into SQL: Split, Push-Up and Unfold. The first algorithm splits an XPath query recursively according to the descendant axes and branches, resulting in a set of suffix path subqueries. Each suffix path subquery can be transformed into an efficient SQL subquery using P-labels. These SQL subqueries are then combined using D-labels to obtain the final SQL query plan. We show that the query plan generated by Split algorithm requires fewer joins, much fewer disk accesses and produces smaller intermediate results than approaches based solely on D-labeling. The next query translation algorithm, Push-up, enhances Split by producing more specific subqueries and further reducing disk accesses and the size of intermediate results. Finally, Unfold uses schema information to further improve the query processing of the Push Up algorithm. Within BLAS, we use Push-Up when schema information is not available, and Unfold otherwise.

The outline and contributions of this paper are:

1. Section 3: We present a new labeling scheme called P-labeling, which can efficiently evaluate suffix path queries using selections on P-labels.
2. Section 4: We present the BLAS system based on P-labeling and D-labeling as a generic framework for XML storage and query processing.
3. Section 4.1: Three query translation algorithms, Split, Push-Up and Unfold, are proposed to generate efficient SQL query plan from the input XPath queries.
4. Section 5: Experimental results on a variety of queries and data sets based on RDBMS and holistic twig join techniques

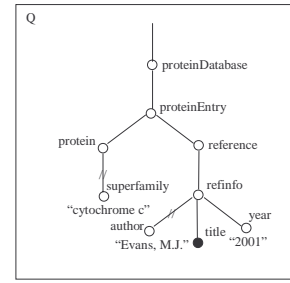


Figure 3: Query tree of  $Q$

[6] prove the efficiency of our approach.

We close by discussing related and future work.

## 2. QUERY LANGUAGE

In this paper, we focus on a commonly used subset of XPath queries consisting of child axis navigation ( $/$ ), descendant axis navigation ( $//$ ), and branches (or qualifiers, denoted as  $[..]$ ). Since such queries can be represented as trees, we call them *tree queries*. Queries without branches can be represented as paths, and are called *path queries*.

For example, the query in figure 2 is represented as the query tree of figure 3. We create a node for each tag in the query, and annotate the node with the tag. The return node 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 that it starts with axis  $/$  or  $//$ . If a node has more than one child then it is a *branching point*. For example, the nodes tagged with *ProteinEntry* and *refinfo* are two branching points. If the return node is not a leaf then it is also a branching point.

**Definition 2.1:** The *evaluation* of a path expression  $P$  returns the set of nodes in an XML tree  $T$  which are reachable by  $P$  starting from the root of  $T$ . This set of XML nodes is denoted as  $\llbracket P \rrbracket$ . ■

Since a path expression  $P$  can be evaluated to retrieve a set of XML nodes, we use “path expression” and “query” interchangeably.

**Definition 2.2:** A path expression  $P$  is *contained* in a path expression  $Q$ , denoted  $P \subseteq Q$ , if and only if for any XML tree  $T$ ,  $\llbracket P \rrbracket \subseteq \llbracket Q \rrbracket$ .

Path expressions  $P$  and  $Q$  are *non-overlapping*, denoted  $P \cap Q = \emptyset$ , if and only if for any XML tree  $T$ ,  $\llbracket P \rrbracket \cap \llbracket Q \rrbracket = \emptyset$ . ■

We process a complex XPath query by decomposing it to a set of subqueries called suffix path expressions. Suffix path expressions have special properties that enable efficient evaluation.

**Definition 2.3:** A *suffix path expression* is a path expression  $P$  which optionally begins with a descendant axis step ( $//$ ), 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. Another example is  $/proteinDatabase/proteinEntry/protein/name$ , which is also a simple path expression.

**Definition 2.4:** A *source path* of a node  $n$  in an XML tree  $T$ , denoted as  $SP(n)$ , is the unique simple path  $P$  from the root to itself.

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

### 3. THE LABELING SCHEME

In this section, we present a bi-labeling scheme which transforms XML data into relations, and XPath queries into SQL which can be efficiently evaluated over the transformed relations. The labeling scheme consists of two labels, one for speeding up descendant axis steps (D-label), and the other for speeding up consecutive child axis steps (P-label). We then build a generic  $B^+$  tree index on the labels. Using a combination of labeling and indexing, we achieve significant speed up for a large class of XPath queries.

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

An XPath query frequently contains descendant axes //. For example, the query  $//t_1//t_2$  asks for all nodes tagged with  $t_2$  which are descendants of some node tagged with  $t_1$ . Recently, D-labeling was applied [16, 23, 1, 3, 31, 13] to speed up descendant axis processing.

**Definition 3.1:** A *D-label* of an XML node is a triplet:  $\langle d_1, d_2, d_3 \rangle$ , such that for any two nodes  $n$  and  $m$ ,  $n \neq m$ :

**Validation:**  $n.d_1 \leq n.d_2$ .

**Descendant:**  $m$  is a descendant of  $n$  if and only if  $n.d_1 < m.d_1$  and  $n.d_2 > m.d_2$ .

**Child:**  $m$  is a child of  $n$  if and only if  $m$  is a descendant of  $n$  and  $n.d_3 + 1 = m.d_3$ .

**Nonoverlap:**  $n$  and  $m$  have no ancestor-descendant relationship if and only if  $n.d_2 < m.d_1$  or  $n.d_1 > m.d_2$ . ■

In this way, the ancestor-descendant relationship between any two nodes can be determined solely by checking their D-labels.

In this paper, we adopt the implementation of D-labeling suggested in [31, 13]. Let the *interval* of a node  $n$  denote the area between the start tag and end tag of  $n$ . The implementation is based on the following observation: an XML node  $m$  is a descendant of another node  $n$  if and only if  $m$  is nested within  $n$  in the XML document. Let  $d_1$  and  $d_2$  for a node  $n$  be the position of the start tag and end tag of  $n$  in the XML document, respectively. To distinguish child from descendant,  $d_3$  is set to be the level of  $n$  in the XML tree. Here, the *level* of  $n$  is defined as the length of the path from the root to  $n$ . For example, in figure 1 the first node tagged *classification* begins at position 7 and ends at position 11 (we treat each start tag, end tag and text as a separate unit). Its level is 4. It is easy to see that this implementation satisfies all the requirements of a D-label. In what follows, a D-label will be represented as  $\langle start, end, level \rangle$ .

To use this labeling scheme for processing descendant axis queries such as  $//t_1//t_2$ , we first retrieve all the nodes reachable by  $t_1$  and by  $t_2$ , resulting in two lists  $l_1$  and  $l_2$ . We then test for the ancestor-descendant relationship between nodes in list  $l_1$  and those in list  $l_2$ . Interpreting  $l_1$  and  $l_2$  as relations, this test is a join with “descendant” property as the join predicate. We therefore call this a *D-join*.

**Example 3.1:** As an example, consider the query  $//proteinDatabase//refinfo$  and let  $pDB$  and  $refinfo$  be relations which store nodes tagged by *proteinDatabase* and *refinfo*, respectively. The D-join could 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 ■
```

The key idea of D-labeling is to speed up descendant axis navigation. Since queries typically involve several child axis steps, it is also important to implement child axis navigation efficiently. For example, consider the following XPath query:

```
/proteinDatabase/proteinEntry/protein/name
```

This query retrieves the names of proteins for each protein entry in the XML file. Using D-labeling, each child axis is processed as a D-join, resulting in a query with 3 D-joins. Since the join operation is very expensive compared to relational select and project operations, an immediate question is whether we can reduce the number of D-joins in queries with multiple child axis steps.

#### 3.2 P-labeling

The P-labeling scheme is used to efficiently process consecutive child axis steps (a suffix path query). The intuition of P-labeling is that each XML node  $n$  is annotated with a label according to its source path  $SP(n)$ , and a suffix path query  $Q$  is also annotated with a label, such that the containment relationship between  $SP(n)$  and  $Q$  can be determined by examining their labels. Hence suffix path queries can be evaluated efficiently.

##### 3.2.1 P-labeling Properties

**Definition 3.2:** A *P-label* for a suffix path  $P$  is an interval  $I_P = \langle p_1, p_2 \rangle$ , such that for any two suffix path expressions  $P, Q$ :

**Validation:**  $P.p_1 \leq P.p_2$

**Containment:**  $P \subseteq Q$  if and only if interval  $I_P$  is contained in  $I_Q$ , i.e.  $Q.p_1 \leq P.p_1$  and  $Q.p_2 \geq P.p_2$ .

**Nonintersection:**  $P \cap Q = \emptyset$  if and only if  $I_P$  and  $I_Q$  do not overlap, i.e.  $P.p_1 > Q.p_2$  or  $P.p_2 < Q.p_1$ . ■

Using the definition of a suffix path expression, it is not hard to prove that for any two suffix paths  $P$  and  $Q$ , either  $P$  and  $Q$  have a containment relationship (that is,  $P \subseteq Q$  or  $Q \subseteq P$ ), or they are non-overlapping. Therefore if  $P.p_1$  is contained in the P-label of  $Q$  (an interval), then  $P.p_2$  is also contained in the P-label of  $Q$ .

Since the evaluation of a suffix path query  $Q$  entails finding all XML nodes  $n$  such that  $SP(n) \subseteq Q$ , the evaluation can be implemented as finding all  $n$  such that the P-label of  $SP(n)$  is contained in the P-label of  $Q$ . As discussed above, this is equivalent to finding all nodes  $n$  such that  $Q.p_1 \leq SP(n).p_1 \leq Q.p_2$ . Therefore, we call  $SP(n).p_1$  the *P-label for an XML node*, and evaluate suffix path query  $Q$  by obtaining the set of XML nodes whose P-labels are contained in the P-label of  $Q$ . Formally,

**Definition 3.3:** For an XML node  $n$ , such that  $SP(n) = \langle p_1, p_2 \rangle$ , the *P-label for this XML node*, denoted as  $n.plabel$ , is the integer  $p_1$ . ■

Notice that here the concept of P-label is overloaded for suffix paths and XML nodes. Though we could define the P-label of an XML node  $n$  to be the P-label of its source path  $SP(n)$ , using the start position of the interval ( $p_1$ ) saves space without affecting the result of query evaluation.

**Proposition 3.2:** Let  $Q$  be a suffix path query. Then  $\llbracket Q \rrbracket = \{n \mid Q.p_1 \leq n.plabel \leq Q.p_2\}$

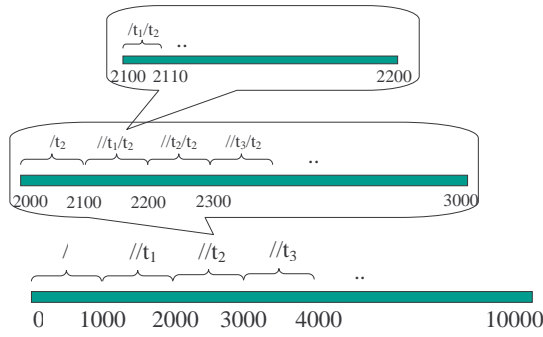


Figure 4: Illustration of interval partition

**Algorithm 1** P-Label( $q: \alpha \ /l_1/l_2/ \dots /l_n$  where  $\alpha \in \{/, //\}$ )

```

1:  $\langle p_1, p_2 \rangle = \langle 0, m - 1 \rangle$ 
2: for  $i = n; i \geq 1; i --$  do
3:   Find  $t_j$  such that  $t_j = l_i$ 
4:    $p'_1 = p_1 + (p_2 - p_1 + 1) * \sum_{k=0}^{j-1} r_k$ 
5:    $p'_2 = p_1 + (p_2 - p_1 + 1) * \sum_{k=0}^j r_k - 1$ 
6:    $p_1 = p'_1, p_2 = p'_2$ 
7: end for
8: if  $\alpha = /$  then
9:    $p_2 = p_1 + (p_2 - p_1 + 1) * r_0 - 1$ 
10: end if
11: return  $\langle p_1, p_2 \rangle$ 

```

Furthermore, if  $Q$  is a simple path, then:

$$\llbracket Q \rrbracket = \{n \mid Q.p_1 = n.plabel\} \quad \blacksquare$$

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-label 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, every child axis is evaluated one after another, and a total of  $(l - 1)$  D-joins are needed where  $l$  is the number of axis steps ( $/$  or  $//$ ) in  $P$ .

As we can see, D-labeling takes advantage of the well-nestedness of XML data and provides a “node containment” labeling scheme to detect ancestor-descendant relationship efficiently. P-labeling takes advantage of suffix path expressions, providing a “path containment” labeling scheme to implement child axis steps.

**Algorithm 2** P-Label(XML tree: T)

```

1: Stack  $s$ 
2: for all  $i$  do
3:    $\langle p_{i1}, p_{i2} \rangle = \text{P-Label}(//t_i)$ 
4: end for
5: push( $s, \langle 0, m - 1 \rangle$ )
6: Depth-first search(T){
7: if current tag is  $< t_i >$  then
8:    $\langle p_1, p_2 \rangle = \text{top}(s)$ 
9:    $p_1 = p_{i1} + p_1 * (p_{i2} - p_{i1} + 1) / m$ 
10:   $p_2 = p_{i1} + (p_2 + 1) * (p_{i2} - p_{i1} + 1) / m - 1$ 
11:  push( $s, \langle p_1, p_2 \rangle$ )
12:  label this node with  $p_1$ 
13: end if
14: if current tag is  $< /t_i >$  then
15:   pop( $s$ )
16: end if
17: }

```

### 3.2.2 P-labeling Construction

Our approach is described as follows. Suppose that there are  $n$  distinct tags  $(t_1, \dots, t_n)$ . We assign “/” a ratio  $r_0$ , and each tag  $t_i$  a ratio  $r_i$ , such that  $\sum_{i=0}^n r_i = 1$ . Let  $r_i = 1/(n + 1)$  for all  $i$ .

Define the domain of the numbers in a P-label to be integers in  $[0, m - 1]$ . The  $m$  is chosen such that  $m \geq (n + 1)^h$ , where  $h$  is the longest path in an XML tree. The length of a P-label of a suffix path expression is the number of integers contained in the P-label interval. Suppose there is an ordering for tags, where the particular ordering used is not important. Using the tag ratios and the order, we construct P-labels for suffix path expressions as follows:

1. Path  $//$  is assigned an interval (P-label) of  $\langle 0, m - 1 \rangle$ .
2. Partition the interval  $\langle 0, m - 1 \rangle$  in tag order proportional to  $t_i$ 's ratio  $r_i$  for each path  $//t_i$  and  $/$ 's ratio  $r_0$ . Assuming that the order of tags is  $t_1, t_2, \dots, t_n$ , this means that we allocate an interval  $\langle 0, m * r_0 - 1 \rangle$  to  $/$  and  $\langle p_i, p_{i+1} - 1 \rangle$  to each  $t_i$ , such that  $(p_{i+1} - p_i) / m = r_i$  and  $p_1 / m = r_0$ .  
Intuitively, we allocate  $\langle 0, p_1 \rangle$  to suffix paths starting with “/”, and  $\langle p_i, p_{i+1} - 1 \rangle$  to suffix paths starting with “ $//t_i$ ”.
3. 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  $/t_i$ ) is now assigned a subinterval, and the proportion of the length of interval of  $//t_j/t_i$  (or  $/t_i$ ) over the length of interval of  $//t_i$  is the ratio  $r_j$  (or  $r_0$ ).  
Intuitively, since  $\llbracket //t_j/t_i \rrbracket \subseteq \llbracket //t_i \rrbracket$  and  $\llbracket /t_i \rrbracket \subseteq \llbracket //t_i \rrbracket$ , we partition the interval for  $//t_i$  into subintervals according to the ratio of all tags  $t_j$  and the ratio for  $/$ .
4. Continue to partition over each subinterval as needed.

As an example, the partitioning procedure for  $m = 10001$  and tags  $t_1, t_2, t_3, \dots, t_9$  is illustrated in figure 4. The P-label assigned to path  $/t_1/t_2$  is  $\langle 2100, 2110 \rangle$ .

It is easily seen that this implementation of the P-labeling scheme is valid, i.e. it satisfies the properties in definition 3.2. The detailed algorithms for constructing the P-label of suffix paths and of XML nodes are presented in algorithms 1 and 2, respectively.

To evaluate a suffix path query  $P$ , we check whether the P-label of a node is contained in  $P$ 's P-label.

**Example 3.3:** Let us look at how to construct P-labels for the sample XML data in figure 1. For simplicity, assume  $m = 10^{12}$  and that there are 99 tags. Each tag is assigned a ratio 0.01. Suppose the order is  $/, ProteinDatabase, ProteinEntry, protein, name, \dots$ . Figure 5 shows how to construct a P-label for suffix path  $P = /ProteinDatabase/ProteinEntry/protein/name$  according to algorithm 1. We begin by assigning P-label  $\langle 4 \times 10^{10}, 5 \times 10^{10} - 1 \rangle$  to suffix path  $//name$ . Then we extract a subinterval from it according to the tag order and the ratio of tag *protein*, and get the P-label for path  $//protein/name$ , and so on. Finally we get the P-label for suffix path  $P$  as  $\langle 4.030201 \times 10^{10}, 4.03020101 \times 10^{10} - 1 \rangle$ . According to the definition 3.3, every node reachable by  $P$  is assigned a P-label  $4.030201 \times 10^{10}$ .

As an example of evaluating a suffix path query, suppose we wish to evaluate query  $//protein/name$ . First we compute its P-label:  $\langle 4.03 \times 10^{10}, 4.04 \times 10^{10} - 1 \rangle$  as shown in figure 5. Then we find all the nodes  $n$  such that  $4.03 \times 10^{10} \leq n.plabel \leq 4.04 \times 10^{10} - 1$ . Suppose all the XML nodes are stored in a relation *nodes*, and each node corresponds to a tuple. The attribute *plabel* records the P-label of the node. The suffix path query can be evaluated by the following SQL statement:

Path expression	P-label
//name	$\langle 4 \times 10^{10}, 5 \times 10^{10} - 1 \rangle$
//protein/name	$\langle 4.03 \times 10^{10}, 4.04 \times 10^{10} - 1 \rangle$
//ProteinEntry/protein/name	$\langle 4.0302 \times 10^{10}, 4.0303 \times 10^{10} - 1 \rangle$
//ProteinDatabase/ProteinEntry/protein/name	$\langle 4.030201 \times 10^{10}, 4.030202 \times 10^{10} - 1 \rangle$
/ProteinDatabase/ProteinEntry/protein/name	$\langle 4.030201 \times 10^{10}, 4.03020101 \times 10^{10} - 1 \rangle$

Figure 5: P-labels for some suffix path expressions

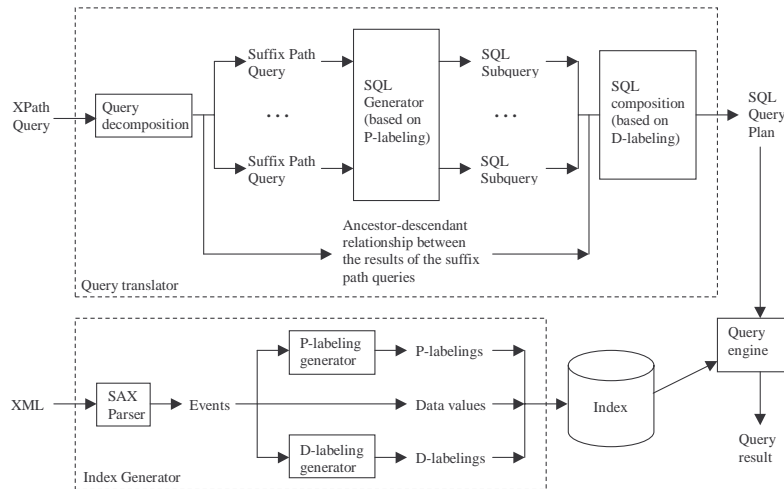


Figure 6: The architecture of BLAS system

**select \* from nodes**

**where** nodes.plabel  $\geq 4.03 \times 10^{10}$   
**and** nodes.plabel  $\leq 4.04 \times 10^{10} - 1$

As illustrated above, nodes with source path */ProteinDatabase/-ProteinEntry/protein/name* have a label  $4.030201 \times 10^{10}$ , and are therefore part of the answer to this query. ■

P-labeling works very well for suffix path queries as shown in this section. However, many practical examples have branch predicates and/or descendant axis navigation in the middle of the query. Used alone, P-labeling is not helpful for processing these complex queries.

## 4. BLAS SYSTEM

To efficiently answer complex queries, we propose BLAS, which is based on P-labeling and D-labeling schemes. The architecture of BLAS is presented in Figure 6. BLAS is composed of three parts: an index generator, a query translator and a query engine.

The BLAS index generator handles events generated by a SAX parser over an XML document. It builds P-labels and D-labels for each element node, and stores text values. Specifically, a tuple  $\langle plabel, start, end, level, data \rangle$  is generated for every node  $n$ , where *plabel* is the P-label of  $n$ , *start* and *end* are the start and end tag positions of  $n$  in the XML document, respectively, *level* is the level of  $n$ , and *data* is used to store the value of  $n$  if there is any (otherwise, *data* is set to null). Notice that  $\langle start, end, level \rangle$  is the D-labeling of the XML document. Furthermore, the tuples are clustered by  $\{plabel, start\}$ , and  $B^+$  tree indexes are built on *start*, *plabel* and *data* to facilitate searches.

The BLAS query translator translates an input XPath query into standard SQL. It is composed of three modules: query decomposition, SQL generation and SQL composition. The query decom-

position module generates a tree representation of the input XPath query, 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-labeling and generates a corresponding subquery in SQL. Finally, the subqueries are combined into a single SQL query plan by the SQL composition module based on D-labeling and the ancestor-descendant relationship between the suffix path queries results.

There are two alternative query engines. One is an RDBMS and the other is a file system implementing holistic twig joins [6].<sup>1</sup>

We have discussed the index generator in section 3; next we will present the query translator.

### 4.1 Query Translator

In this section, we will present three query translation algorithms – Split, Push-up and Unfold – that translate a complex XPath query into an efficient SQL query plan. Split is used only for purposes of exposition; for reasons that will become clear in section 5, Unfold is used in BLAS when schema information is present and Push-up when it is absent.

#### 4.1.1 Split Algorithm

The simplest query translator algorithm is called *Split*. The algorithm splits the query tree into one or more parts, where each part is a suffix path query.

Split consists of two steps: *descendent axis elimination* and *branch elimination*. Exchanging the order of these two steps will not affect the query result. The two steps can also be interleaved.

<sup>1</sup>The prototype implementation of holistic twig joins takes a tree pattern query as input. However, it is not hard to show that SQL input can be translated to a tree pattern query, so we can consider all input to the query engine to be SQL.

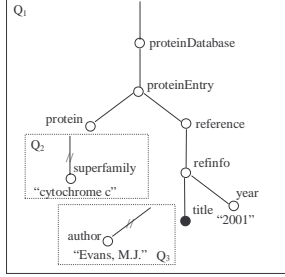
---

**Algorithm 3** D-elimination(query tree Q)

---

```
1: List intermediate-result
2: Depth-first search(Q){
3: if current node reached by a // edge then
4:   Q' = the subtree rooted at the current // edge
5:   Cut Q' from Q;
6:   intermediate-result.add(D-elimination(Q'))
7: end if
8: }
9: result = answer(Q)
10: for all r in intermediate-result do
11:   result = D-join(result,r)
12: end for
13: return result
```

---



**Figure 7: Descendant-axis elimination for Q**

The basic operation of descendant-axis elimination (shown in algorithm 3) is to take a query as input, do a depth-first traversal and split any descendent axis of form  $p//q$ , into  $p$  and  $//q$ . In algorithm 3, *answer* is an abstract function which invokes the next step (B-elimination) if there are branching points in the query tree  $Q$ ; otherwise, it evaluates  $Q$  using P-labeling. A *D-join* is then used to join intermediate results by their D-labels, as discussed in section 3.1.

The basic operation of branch elimination is to take a query as input, do a depth-first traversal and split any branch axis of form  $p[q_1, q_2, \dots, q_l]/r$  into  $p, //q_1, //q_2, \dots, //q_l, //r$  (see algorithm 4). As in algorithm 3, *answer* is an abstract function. If  $Q$  is a suffix path query,  $Q$  is evaluated using P-labeling. Otherwise, if  $Q$  contains descendant axes, D-elimination is called to further decompose it into suffix path queries. A *D-join* is then used to join intermediate results by their D-labels. Different than the D-joins discussed in section 3.1, we use level information in the where clause of a SQL statement to specify the level difference between the intermediate results.

---

**Algorithm 4** B-elimination(query tree Q)

---

```
1: List intermediate-result
2: Depth-first search(Q){
3: if current node has more than one child then
4:   for all child of Q: Q' do
5:     cut Q' from Q
6:     Q' = //Q'
7:     intermediate-result.add(B-elimination(Q'))
8:   end for
9: end if
10: }
11: result = answer(Q)
12: for all r in intermediate-result do
13:   result = D-join(result,r)
14: end for
15: return result
```

---

---

**Algorithm 5** 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 has more than one child then
5:   Path = false
6:   for all child of Q: Q' do
7:     cut Q' from Q
8:   end for
9:   SP = P/Q
10:  for all child of Q: Q' do
11:    intermediate-result.add(
12:      PushUp B-eliminate-sub(Q', SP))
13:  end for
14: end if
15: }
16: if Path then
17:   SP = P/Q
18: end if
19: result = answer(SP)
20: for all r in intermediate-result do
21:   result = D-join(result,r)
22: end for
23: return result
```

---

**Example 4.1:** As an example, suppose we want to translate the query in figure 3. Figure 7 illustrates how to eliminate the descendant axis. Since  $Q_1$  contains branching points, the branch elimination procedure must be invoked to further decompose it. Branch axis is further eliminated in Figure 8. After that, we can see that each resulting subquery is a suffix path query, which can be evaluated directly using P-labeling as discussed in section 3.2.

Suppose the evaluation of  $Q_4$  results in a list of nodes *pEntry* that are reachable by path  $/ProteinDatabase/ProteinEntry$ , and the evaluation of  $Q_7$  results in a list of nodes *refinfo* that are reachable by path  $//reference/refinfo$ . *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
```

Note that we have  $pEntry.level = refinfo.level - 2$  in the where clause, which is different from the general D-join algorithm where no level predicate is needed in the where clause. This is because the paths *ProteinDatabase/ProteinEntry* and *reference/refinfo* are connected directly in the original query (figure 3), therefore the *ProteinEntry* nodes returned by the query *ProteinDatabase/ProteinEntry* are grandfather of the *refinfo* nodes returned by the query *//reference/refinfo* rather than an general ancestor. This information about level difference can be obtained from the branch elimination procedure. Also the D-labels of both *pEntry* and *refinfo* are recorded since they may be involved in D-joins with other intermediate results. ■

#### 4.1.2 Push-up Algorithm

As illustrated in section 4.1.1, Split decomposes a complex query into a set of subqueries, each of which is a suffix path query. Observe that the branch elimination algorithm, which eliminate a branch of form  $p[q_1, q_2, \dots, q_l]/r$  into  $p, //q_1, //q_2, \dots, //q_l, //r$ , ignores the fact that the root of  $q_i, 1 \leq i \leq l$  and  $r$  is a child of the

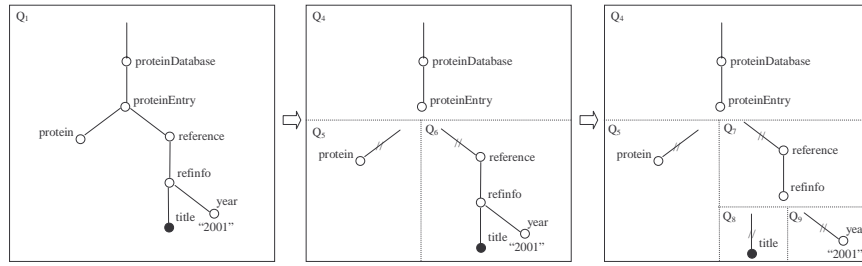


Figure 8: Branch elimination for  $Q_1$

leaf of  $p$ . Rather than evaluate  $//q_i$  and  $//r$ , we should therefore evaluate  $p/q_i$  and  $p/r$ . Since  $p/q_i$  and  $p/r$  are more specific than  $//q_i$  and  $//r$  (recall that we cluster XML data by  $\{plabel, start\}$ ), the number of disk accesses and the size of the intermediate results is reduced without affecting the final result.

This observation causes us to redesign the basic operation of branch elimination so as to split a query of form  $p[q_1, q_2, \dots, q_l]/r$  into  $p, p/q_1, p/q_2, \dots, p/q_l, p/r$  (see algorithm 5). The intuition is that during branch elimination, for each branching point  $n$  we *push up* the path expression of  $n$ 's children toward the root. We call this step *push-up branch elimination*, and the whole query processing algorithm the *Push-up* algorithm. The key difference between algorithm 4 and algorithm 5 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  $SP/Q'$ .

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 figure 9, each  $SP/Q'$  is a suffix path expression and can be evaluated using P-labeling. This is because the input of push-up branch elimination is a subquery tree obtained from the descendant-axis elimination algorithm, which eliminates all descendant axis steps in the middle of query. 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.

### 4.1.3 Unfold Algorithm

A further optimization of descendant-axis elimination is possible when schema information is available. For non-recursive schemas, path expressions with wildcards can be evaluated over the schema graph and wildcards can be substituted with actual tags. Thus a query of form  $p//q$  can be enumerated by all possibilities  $(p/r_1/q, p/r_2/q, \dots, p/r_l/q)$ , and the result of the query is the union of the results of  $p/r_1/q, p/r_2/q, \dots, p/r_l/q$ . For a recursive schema, given statistics about the depth of the XML tree, queries can be unfolded to this depth and the occurrences of  $//$  can be eliminated. Since descendant axis traversals are substituted with child axis traversals, we call this optimization *unfold descendant-axis elimination*, and the resulting algorithm which splits a query  $Q$  into suffix path subqueries the *Unfold* algorithm.

One advantage with Unfold is that we replace D-joins with a process that first performs selections on P-labels and then unions the results. This is very efficient because selections using an index are cheap, and the union is very simple since there are no duplicates.

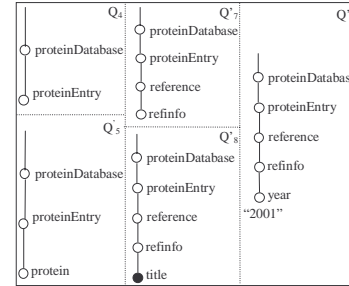


Figure 9: Push branch elimination for  $Q_1$

Another advantage is that the subqueries are all simple path queries, which can be implemented as a select operation with equality predicates instead of range predicates.

Furthermore, we also reduce the number of disk accesses. For example, to process query  $t_1//t_2$ , let the list of nodes tagged with  $t_1$  be  $l_1$ , and the list of nodes tagged with  $t_2$  be  $l_2$ , where the cardinality of  $l_1$  and  $l_2$  are  $n_1$  and  $n_2$ , respectively. A D-join of  $l_1$  and  $l_2$  requires  $O(n_1 + n_2)$  node accesses. However, unfold  $t_1//t_2$  produces queries  $t_1/p_1/t_2, \dots, t_1/p_s/t_2$ , where  $p_1, \dots, p_s$  are simple path expressions. The query can then be implemented as a select operation over  $l_2$ , resulting in at most  $n_2$  node accesses.

**Example 4.2:** For our sample query  $Q$ , first we apply push-up branch-elimination and get the following set of subqueries:  $Q_4, Q'_5, Q'_7, Q'_8, Q'_9$  and  $Q''_2 = /ProteinDatabase/ProteinEntry/protein//superfamily="cytochrome c", Q''_3 = /ProteinDatabase/ProteinEntry/reference/refinfo//author="Evans, M.J."$ .

Applying the unfold descendant-axis elimination to subqueries  $Q''_2$  and  $Q''_3$ , we get  $Q'''_2 = /ProteinDatabase/ProteinEntry/protein/classification/superfamily="cytochrome c", Q'''_3 = /ProteinDatabase/ProteinEntry/reference/refinfo/authors/author="Evans, M.J."$ .

Now all the subqueries of  $Q$  are simple path queries and can be evaluated by a select operation on their P-labels with equality predicates. ■

## 4.2 Efficiency of the Algorithms

We claim that our algorithms are more efficient than an approach which only uses D-labeling because:

1. The number of joins are reduced. Recall that with D-labeling, a query which contains  $l$  tags requires  $(l - 1)$  D-joins. However, if  $b$  is the number of outgoing edges, which are not annotated with  $//$ , of a branching point and  $d$  is the number

QS1	/PLAYS/PLAY/ACT/SCENE/SPEECH/LINE
QS2	/PLAYS/PLAY/EPILOGUE//LINE/STAGEDIR
QS3	/PLAYS/PLAY/ACT/SCENE[TITLE='SCENE III. A public place.']/LINE
QP1	/ProteinDatabase/ProteinEntry/protein/name
QP2	/ProteinDatabase/ProteinEntry//authors/author='Daniel, M.'
QP3	/ProteinDatabase/ProteinEntry[reference/refinfo[citation and year ]]/protein/name
QA1	//category/description/parlist/listitem
QA2	/site/regions//item/description
QA3	/site/regions/asia/item[shipping]/description

Figure 10: Query Sets

of descendant axis steps, then the number of D-joins in our Split and Push-up algorithms is bounded by  $(b + d)$ , which is always less than  $(l - 1)$ . In the presence of schema information, we can apply the Unfold algorithm and reduce the number of D-joins to  $b$ .

- The number of disk accesses are reduced. Since we cluster XML data by  $\{p_{label, start}\}$ , the number of disk accesses of BLAS is less than that of D-labeling. For example if we have a query  $Q = /t_1/t_2/ \dots /t_n$ , using D-labeling we should get all the tuples with tag  $t_i$  where  $1 \leq i \leq n$ . Using BLAS, we only access the tuples whose P-label is contained in that of  $/t_1/t_2/ \dots /t_n$ , which is bounded by the number of tuples with tag  $t_n$ .

## 5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of BLAS, we compare its performance with the traditional D-labeling scheme. We test the performance on two alternative query engines: a relational database query engine, and a file system using the holistic twig join algorithm [6] for joins. The experiments were performed over three XML data sets with different schema characteristics. The XPath queries used were chosen to represent suffix path queries, path queries with descendant axis traversal, and general tree queries. A benchmark query set was also used for the Auction data set. Experimental results show a substantial performance improvement of BLAS over the traditional D-labeling scheme for both query engines tested. Furthermore, they show that Unfold outperforms both Split and Push-up when XML schema information is available, and Push-up outperforms Split in the absence of such information.

### 5.1 Experimental Setup

The experiments were performed on a 1.5GHz Pentium 4 machine running Windows 2000, with 512MB memory and one 40GB hard disk (7200rpm). All experiments were repeated 10 times independently on a cold cache, and the average processing time was calculated disregarding the maximum and minimum values.

#### 5.1.1 Data Sets

The three data sets are Shakespeare [5], Protein [18] and Auction [30], and are described below:

	Shakespeare	Protein	Auction
Size	1.3MB	3.5MB	3.4MB
Nodes	31975	113831	61890
Tags	19	66	77
Depth	7	7	12

Figure 12: XML Data Sets

**Shakespeare:** This data set represents Shakespeare’s plays in XML format. The DTD of this data is a graph.

**Protein:** This data set is part of the integrated collection of functionally annotated protein sequences from the Georgetown Protein Information Resource. The DTD of this data is a tree.

**Auction:** This data set contains information about auctions. It is a synthetic benchmark data set generated by the XML Generator from XMark. The data generated conforms to the default benchmark DTD provided. Its DTD is recursive, and the instance data is relatively deep (12 levels).

Characteristics of these data sets are summarized in figure 12. *Size* denotes the disk space used to store the original XML file. *Nodes* is the number of nodes in the XML file, including element and attribute nodes. *Tags* is the number of distinct tags. *Depth* is the length of the longest simple path in the XML file.

#### 5.1.2 Queries

For each data set, we tested several types of XPath queries. The first is a suffix path query, in which a descendant axis appears only at the beginning (if it presents) and branches are not allowed. The second is a path query in which a descendant axis can appear anywhere in the query, but branches are not allowed. The third is a general tree query in which branches and descendant axes are both allowed to appear anywhere in the query.

We choose these types of queries for the following reasons. The first type of query tests the performance of P-labeling versus D-labeling for long suffix path queries. The second type of query evaluates how to use the bi-labeling scheme of BLAS to optimize path query processing. The third type, twig queries, occur frequently in practice.

For the Auction dataset, we also tested a set of benchmark queries provided by XMark [30] which only contains “/”, “//” and branches.

The non-benchmark queries are listed in figure 10. The names of the queries are encoded by “QXY”, where ‘X’ is one of ‘S’(Shakespeare), ‘P’(Protein) or ‘A’(Auction), and ‘Y’ is one of ‘1’(type 1, suffix path), ‘2’(type 2, path with no branching) or ‘3’(type 3, general tree query). We also use the original query names Q1, ..., Q6 for the benchmark queries.

## 5.2 Relational Database Implementation

Although holistic twig joins have been shown to be much more efficient than relational database joins for implementing twig joins with no value predicates [6], relational databases are still heavily used for storing XML data. We therefore start by comparing the system with the Split, Push-up and Unfold query translators with the D-Labeling algorithm using a relational database query engine.

QS3	D-labeling	$\pi_{T6.start}(\rho(T1, \sigma_{tag='PLAYS' \wedge level=1}(SD)))$ $\bowtie_{T1.start < T2.start \wedge T1.end > T2.end \wedge T1.level = T2.level - 1} \rho(T2, \sigma_{tag='PLAY'}(SD))$ $\bowtie_{T2.start < T3.start \wedge T2.end > T3.end \wedge T2.level = T3.level - 1} \rho(T3, \sigma_{tag='ACT'}(SD))$ $\bowtie_{T3.start < T4.start \wedge T3.end > T4.end \wedge T3.level = T4.level - 1} \rho(T4, \sigma_{tag='SCENE'}(SD))$ $\bowtie_{T4.start < T5.start \wedge T4.end > T5.end \wedge T4.level = T5.level - 1} \rho(T5, \sigma_{tag='TITLE' \wedge data='SCENEIII.Apublicplace.'}(SD))$ $\bowtie_{T4.start < T6.start \wedge T4.end > T6.end \wedge T4.level = T6.level - 1} \rho(T6, \sigma_{tag='LINE'}(SD))$
	Split	$\pi_{T3.start}(\rho(T1, \sigma_{plabel=345830491796013056}(SP)))$ $\bowtie_{T1.start < T2.start \wedge T1.end > T2.end} \rho(T2, \sigma_{plabel \geq 396316767208603648 \wedge plabel \leq 432345564227567616}(SP))$ $\bowtie_{T1.start < T3.start \wedge T1.end > T3.end} \rho(T3, \sigma_{plabel \geq 576460752303423488 \wedge plabel \leq 612489549322387456}(SP))$
	Push up	$\pi_{T3.start}(\rho(T1, \sigma_{plabel=345830491796013056}(SP)))$ $\bowtie_{T1.start < T2.start \wedge T1.end > T2.end \wedge T1.level = T2.level - 1} \rho(T2, \sigma_{plabel=407123970077229056}(SP))$ $\bowtie_{T1.start < T3.start \wedge T1.end > T3.end} \rho(T3, \sigma_{plabel \geq 576460752303423488 \wedge plabel \leq 612489549322387456}(SP))$
	Unfold	$\pi_{T3.start}(\rho(T1, \sigma_{plabel=345830491796013056}(SP)))$ $\bowtie_{T1.start < T2.start \wedge T1.end > T2.end \wedge T1.level = T2.level - 1} \rho(T2, \sigma_{plabel=407123970077229056}(SP))$ $\bowtie_{T1.start < T3.start \wedge T1.end > T3.end} \rho(T3, \sigma_{plabel=579050277206753280}(SP))$

Figure 11: Relational algebra expressions generated for QS3 by D-labeling, Split, Push up and Unfold

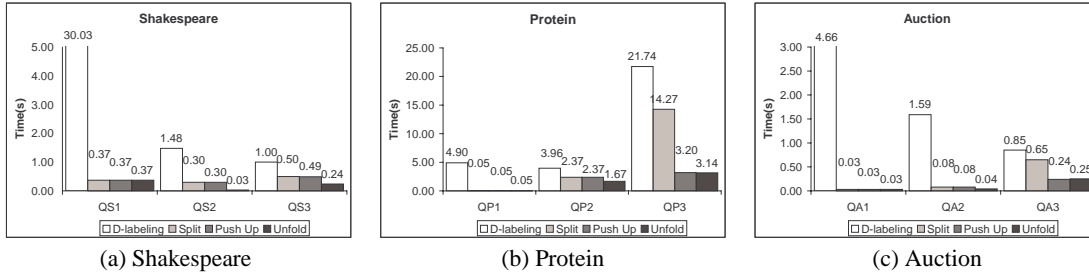


Figure 13: Query time for Shakespeare, Protein and Auction data sets

### 5.2.1 Storage Setup

The XML data sets were stored in DB2 universal version 7.2 using the high-performance option installed in the same machine. 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 for our approach is  $SP(plabel, start, end, level, data)$ , with primary key  $\{start\}$ . Attribute  $\{data\}$  stores string (PCDATA) 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.

### 5.2.2 Query Translation

First we compare the SQL queries generated for schemas  $SD$  and  $SP$ . The queries for  $QS3$  are shown in figure 11, where the big numbers are the labels of the corresponding paths for Split, Push-up and Unfold. (We use relational algebra for the queries instead of SQL to conserve space). As we can see, D-labeling requires 5 D-joins, whereas Split, Push-up and Unfold only require 2 D-joins. Furthermore, Split requires two range selections and one equality selection, Push-up requires one range selection and two equality selections, and Unfold requires three equality selections. Since equality selection has better performance and generates a smaller intermediate result than range selection, Push-up has better performance than Split, and Unfold has the best performance, as shown in section 5.2.3. The other queries have similar results and are omitted

here.

### 5.2.3 Performance Analysis

Figure 13 shows the query processing time for queries 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 contained in any of our measurements.

For each query, we compare the conventional approach using D-labeling, the Split algorithm, the Push-up algorithm and the Unfold algorithm. The results of these experiments show that significant speed-up is achieved using our approach.

For the first type of query, suffix path queries, our approach is 100 times faster than the conventional approach using D-labeling. This is because D-labeling requires  $(l - 1)$  D-joins with a lot of disk accesses to answer a suffix path query with  $l$  tags, while our approach uses a select operation with fewer disk accesses over the P-labels. Observe also that for suffix path queries, the Split, Push-up and Unfold algorithms are the same and therefore have the same performance.

For the second type of query, the conventional approach using D-labeling again requires  $(l - 1)$  D-joins. Split and Push-up are the same algorithms; both involve one D-join and two selections with fewer disk accesses, which is more efficient than D-labeling. On the other hand, Unfold translates the query into a select operation, and is therefore the fastest.

Similarly, for the third type of query the conventional approach using D-labeling requires  $(l - 1)$  D-joins. The number of D-joins needed in Split and Push-up depends on the number of branches

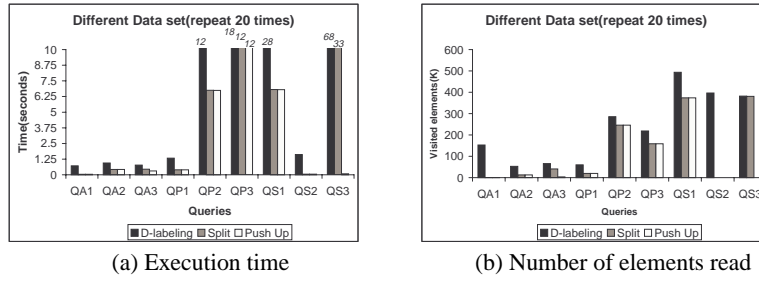


Figure 14: The performance of D-labeling, Split and Push up for the queries on different data sets

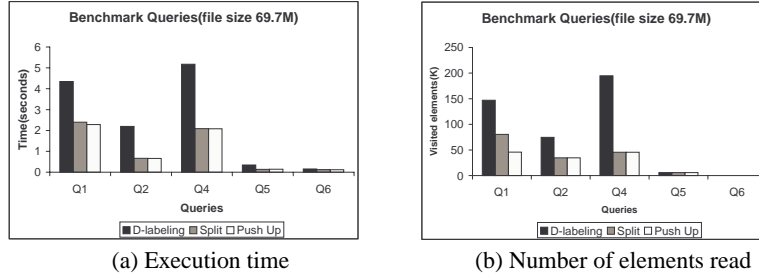


Figure 15: The performance of D-labeling, Split and Push up for the benchmark queries

and descendant axis steps rather than the number of tags, and therefore require fewer D-joins and disk accesses. The size of intermediate results is also smaller. Hence Split and Push-up are both more efficient than the D-labeling approach. 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. Unfold removes D-joins which are related to descendant axes, and therefore has fewer D-joins and disk accesses than either Split or Push-up. Overall, it has the best performance for the third type of query, and is 3-7 times faster than the D-labeling algorithm.

### 5.3 Holistic Twig Join Implementation

In the second experiment, we compared the Split, Push up and Unfold algorithms with the D-Labeling algorithm based on the holistic twig join technique of [6], using a file system as the storage engine. All queries ran significantly faster in this implementation than in the relational database implementation, concurring with the results of [6]. These speed-ups are not presented since the purpose of our experiments is to show the benefit of BLAS over D-labeling in either implementation rather than to show the benefit of holistic twig joins.

#### 5.3.1 Modifications to Query Set

Holistic twig join techniques focus on the core operations in XML query languages, that is, path and twig queries. They have not been developed to support other operations such as value predicates and unions. In this experiment, we therefore removed value predicates from the queries. Furthermore, since the Unfold algorithm uses unions, we only compared the Split and Push-up algorithms with the D-Labeling algorithm.

#### 5.3.2 Queries of Different Types

We studied the efficiency of D-labeling, Split and Push-up across all three data sets: Auction (a recursive DTD), Shakespeare (a graph DTD) and Protein (a tree DTD). In all cases, we test queries on larger data sets by repeating the original data set 20 times to

highlight the differences in running time. The result is shown in Figure 14. Experiments show that for all test queries and data sets, our algorithms are more efficient than the traditional D-labeling algorithm.

#### 5.3.3 Benchmark Queries

We also tested the benchmark queries using the Auction data with size 69.7M. As seen by the results in Figure 15, Push-up has as good or better performance than Split, and Split has better performance than D-labeling.

#### 5.3.4 Scalability

To test the scalability of the algorithms, we replicated the Auction data set between 10 and 60 times to get increasingly large experimental data sets.

Figure 16(a) shows the execution time for suffix path query QA1 on different data set sizes. The Split and Push-up algorithms have the same performance since they share the same query plan on suffix path queries (recall the analysis of Section 5.2.2). Furthermore, the execution time for Split and Push-up is almost constant because they only use selection to get the result. On the other hand, D-labeling needs to read all the data with a tag appearing in the query and do a join for each axis in the query. Notice that as the file size increases the difference between execution time of D-labeling and that of split and Push up algorithm increases. This proves that for suffix path queries, Split and Push up algorithm have better performance and scalability comparing to the D-labeling algorithm. The number of elements read by each approach is shown in the Figure 16(b).

Performance results for D-labeling, Split and Push-up on path query QA2 are shown in Figure 17. Compared to suffix path queries, Split and Push-up need more time to answer path queries since joins and more disk accesses are necessary. However, they still outperform D-Labeling. One reason is that Split and Pushup use fewer joins (recall the analysis of Section 5.2.2), and another is that D-labeling accesses up to 4 times as many elements as Split and Push (see Figure 17(b)). Figure 17 also shows that the difference

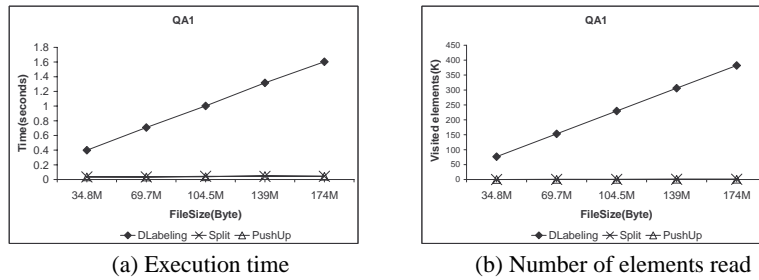


Figure 16: The performance of D-labeling, Split and Push up for the suffix path query

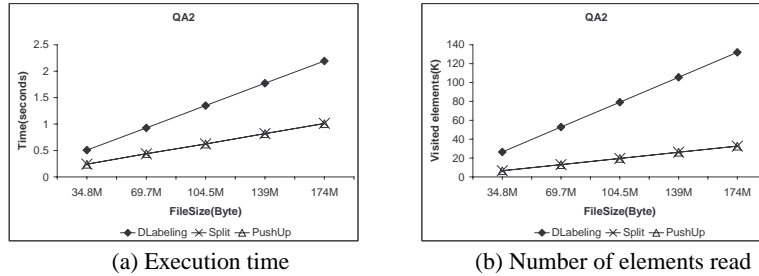


Figure 17: The performance of D-labeling, Split and Push up for the path query

between Split/Push-Up and D-labeling increases as the file grows larger, hence Split/Push-Up is more scalable than D-labeling.

Performance results for twig query QA3 are shown in Figure 18, and again Split and Push-Up outperform D-labeling. An importance difference between this result and the result for the path queries, however, is that Push-up outperforms Split. The reason for this is the difference in query plans (see Section 5.2.2): Although Push-Up uses the same number of joins as Split, the select operations are more selective. Therefore the number of disk access is fewer (see Figure 18(b)), and the execution time is smaller for Push-Up. Figure 18 also shows that the performance differences increase with the file size.

As shown in the experiments, BLAS outperforms the traditional approach which only uses D-labeling scheme on various data sets and queries for both query engines we have tested. The performance enhancement is achieved by reducing the number of joins and disk accesses.

## 6. RELATED WORK

**XML storage and query processing.** One approach is to store XML data natively as a file [28]. However, since the whole file needs to be traversed whenever we process a query, it is not efficient for large XML data sets. There are many ideas of how to store XML using commercial RDBMS, leveraging its indexing and querying processing capabilities [17, 27, 4, 15, 8, 22]. [17] treats an XML document as a graph, and generates a tuple for every edge. The main advantage is that the approach is simple and general, and the mapping between an XML query and SQL can be automatically generated. However, an XML query may involve many self-joins. [27, 4, 15, 8, 22] eliminate joins between a node and its distinct child by inlining the distinct child information into the parent tuple. However, the mapping from an XML query to SQL is very complex, and needs schema information. In all above approaches, we typically need to rely on auxiliary code in a general-purpose programming language together with SQL to express an XML query.

**Indexing.** Various indexing techniques have been proposed for XML query processing and optimization. Structural indexes [24,

25, 22, 12, 21] create a structural summary which is extracted from the XML document as a directed graph. Queries can then be evaluated over the structural summary by pruning the search space. [24, 22, 12] only support path queries. [25] supports tree queries which match some predefined template. [21] discussed covering indexes for branching path queries and proposes to restrict the class of queries being indexed to achieve performance benefit. [14] addresses the XML query optimization problem in the presence of materialized views.

**Labeling.** Several D-labeling implementations have been proposed [16, 23, 1, 3]. [1, 3] addresses the problem of how to build D-labels with the smallest label size. [23, 31, 13] apply D-labeling for answering XML queries. Since D-labeling does not depend on the query workload or features of a document, a generic  $B^+$  tree index can be built over D-labels to support tree queries. The most recent work, [13], shows the effectiveness of D-labeling for translating XQuery to SQL as compared with other XQuery processing techniques. XPRESS [26] proposes an XML data compression technique which uses reverse arithmetic encoding to encode label paths as a distinct interval within  $[0.0, 1)$ . Furthermore, it supports query evaluation over the compressed document using the containment relationship among the intervals. Our P-labeling borrows the idea of labeling a path, but focuses on the optimization of query processing. We use integers rather than floating point numbers to enable efficient process. During P-label construction, the intervals are partitioned uniformly in order to compute the P-label of a query efficiently. Since the join operation is commonly used for query processing with D-labeling, [2, 6, 9, 10, 19, 29, 20] study various join optimization techniques.

## 7. CONCLUSIONS

We present the BLAS system for processing complex XPath queries over XML data. The system is based on a labeling scheme which combines P-labeling (for processing suffix path queries efficiently) with D-labelings (for processing queries involving the descendant axis). Since we use 4 numbers in our labeling scheme to replace tag names, the space used to represent an XML document

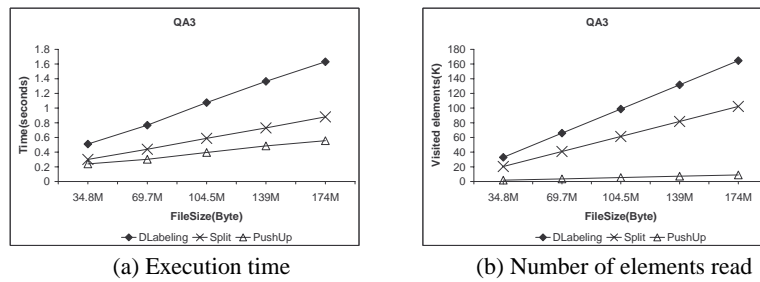


Figure 18: The performance of D-labeling, Split and Push up for the tree query

is comparable to the size of the original document.

Three query translator algorithms were considered for BLAS : Split, Push-up and Unfold. In these algorithms, an XPath query is first decomposed into a set of suffix path sub-queries. P-labels of these sub-queries are then calculated, and the sub-queries translated into SQL expressions. The final SQL query plan is obtained by taking their D-join. BLAS provides a generic and efficient implementation by creating special indexes ( $B^+$  tree and/or  $R$  tree) for optimizing D-joins.

Our experiments show that BLAS improves a large class of XML data sets and queries with comparable storage cost when compared to the D-labeling strategy, using both RDBMS and the holistic twig join technique of [6]. Our experiments also show that the Push-up algorithm is the best query translator when there is no schema information, and that when such information is available Unfold should be used.

In future work, we plan to extend the techniques to handle more complex XPath queries.

## 8. ACKNOWLEDGMENTS

We would like to thank Val Tannen and Wang-Chiew Tan for their constructive and valuable comments. Furthermore, we thank Nicolas Bruno for sharing his efficient implementation of the holistic twig join algorithms. Research is partially supported by NSF DBI-9975206.

## 9. REFERENCES

- [1] S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proceedings of SODA*, 2001.
- [2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002.
- [3] S. Alstrup and T. Rauhe. Improved labeling scheme for ancestor queries. In *Proceedings of SODA*, 2002.
- [4] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML-Schema to Relations: A Cost-Based Approach to XML Storage. In *Proceedings of ICDE*, 2002.
- [5] J. Bosak. Shakespeare. <http://www.ibiblio.org/xml/examples/shakespeare/>.
- [6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of SIGMOD*, 2002.
- [7] Y. Chen, S. Davidson, C. Hara, and Y. Zheng. RRXS: Redundancy reducing XML storage in relations. In *Proceedings of VLDB*, 2003.
- [8] Y. Chen, S. B. Davidson, and Y. Zheng. Constraint Preserving XML Storage in Relations. In *WebDB*, 2002.
- [9] S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Efficient complex query support for multiversion XML documents. In *Proceedings of EDBT*, 2002.
- [10] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of VLDB*, 2002.
- [11] J. Clark and S. DeRose. XML Path language (XPath), November 1999. <http://www.w3.org/TR/xpath>.
- [12] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of VLDB*, 2001.
- [13] D. DeHaan, D. Toman, M. Consens, and M. T. Oszu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of SIGMOD*, 2001.
- [14] A. Deutsch. An Experimental Evaluation of the MARS System. In *Excerpt from PhD Thesis Alin Deutsch*, 2002.
- [15] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 431–442, 1999.
- [16] P. F. Dietz. Maintaining order in a linked list. In *Proceeding of STOC*, 1982.
- [17] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.
- [18] Georgetown Protein Information Resource. Protein Sequence Database, 2001. <http://www.cs.washington.edu/research/xmldatasets/>.
- [19] T. Grust. Accelerating Xpath location steps. In *Proceedings of SIGMOD*, 2002.
- [20] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of ICDE*, 2003.
- [21] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of SIGMOD*, 2002.
- [22] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proceedings of ICDE*, 2002.
- [23] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.
- [24] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Stanford University, 1998.
- [25] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of ICDT*, 1999.
- [26] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A queriable compression for XML data. In *Proceedings of SIGMOD*, 2003.
- [27] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *The VLDB Journal*, pages 302–314, 1999.
- [28] J. Simon and M. Fernandez. Galax. <http://db.bell-labs.com/galax>.
- [29] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree Coding and Efficient Processing of Containment Joins. In *Proceedings of ICDE*, 2003.
- [30] XMARK the XML-benchmark project, April 2001. <http://monetdb.cwi.nl/xml/index.html>.
- [31] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.