MQEB: Metadata-based Query Evaluation of Bi-labeled XML data

Rajesh Kumar A and P Sreenivasa Kumar

Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai 600036, India.
email: hererajesh2000@yahoo.com, psk@cs.iitm.ernet.in

Abstract

XML is gaining importance in representing and exchanging of documents in the World Wide Web. Since XML datasets may be large and complex, efficiently querying XML data is a major concern. We present Metadata-based Query Evaluation of Bi-labeled XML data(MQEB) to efficiently process XPath queries over XML data. We use two labelings, interval encoding and P-labeling. We present algorithms that use P-labels and metadata to evaluate path expressions without joins even if it contains ancestor-descendant operators. In case of branch queries, the query is split into path expressions which are evaluated individually and the results are joined using holistic twig join algorithm TwigStack. P-labeling is used to efficiently process path expressions containing consecutive parent-child operators. Interval encoding is exploited by the holistic twig join algorithm TwigStack. Since the number of joins needed to evaluate a query is reduced, MQEB gives better performance than BLAS for queries containing ancestor-descendant operators. We have implemented both BLAS and the proposed MQEB. Experimental results have shown that the proposed scheme performs better than BLAS.

1 Introduction

With the proliferating use of XML as a framework for exchange of data on the Internet and also as a means of capturing semi-structured data, there is great demand for devising efficient methods for storing, indexing and querying XML data. While certain standards for query languages for XML databases, such as XPATH [1] and XQUERY [11], are getting slowly adopted, the quest for the best storage scheme for XML data is still on. The query languages allow us to specify tree patterns where nodes in the query tree are connected by operators that specify parent-child, ancestor-descendant or preceding-following etc. relationships to be satisfied by the matching data nodes. Solving the query requires us to determine all the sub trees in the XML data tree that match the query tree. The development of efficient algorithms for carrying out basic XML data retrieval is intimately linked with the way nodes in the XML data tree are labeled. Several node labeling schemes have been devised recently [2][3][4][12]. One of the earliest techniques is to use intervals to encode tree nodes [3][9]. The intervals are assigned to nodes in such a way that the interval corresponding to a node would contain the intervals assigned to its children. Checking parent-child or ancestor-descendant relationships would then reduce to checking containment among intervals. However, the root-to-leaf paths in the query tree tend to be long and the approach of handling one edge at a time leads to a large number of joins, called the structural joins in the XML data context. One way of improving the situation is using twig joins [8], joins that match an entire tree pattern to the data. On the other hand, one can label entire paths instead of individual nodes of the tree. The bi-labeling scheme proposed recently in [2] and the reverse arithmetic encoding used in XPRESS [7] follow this approach. In this paper, we focus our attention on the bi-labeling method proposed in the system BLAS [2]. BLAS proposes to use both node encoding (using intervals) and path encoding. XML data tree nodes are labeled using encoding of the path from the root to the node. The query tree is segmented in such a way that each segment is a path with nodes connected by parent/child operators only. These segments are labeled using path encoding and matched with data node labels. To get the final result, structural join algorithms are used on the results of matching the query segments. We analyze the path encoding scheme and observe that though the set of labels required to encode all possible paths is very large in size, the set of labels that represent paths that actually exist in the XML data tree is small in size. We exploit

International Conference on Management of Data
COMAD 2005b, Hyderabad, India, December 20–22, 2005

this fact and propose a system called Metadata-based Query Evaluation of Bi-labeled XML data(MQEB). The contributions of the paper can be summarized as follows:

- Observation that the number of path labels that are needed in encoding of a typical XML dataset is far less than the number of possible path labels.
- Algorithms that exploit metadata to significantly reduce the number of joins required to process queries with ancestor-descendant operators.
- Implementation of the proposed MQEB system and experimentally demonstrating that the proposed algorithms indeed result in improved query evaluation performance on standard datasets[5] such as DBLP, SWISSPROT, NASA.

The rest of the paper is organized as follows: In Section 2, we provide a summary of the background required for understanding the MQEB system. Section 3 gives the details of the proposed system. In Section 4, we present the implementation details of MQEB. Section 5, we report on the performance results obtained. Section 6 offers conclusions.

2 Background

2.1 Interval Encoding

XPath queries frequently contain ancestor-descendant operators. To process ancestor - descendant relationships efficiently, interval encoding was used in [3][9]. In interval encoding, each node is assigned a tuple $< start, end, level >$ where *start* and *end* are the DFS traversal numberings and *level* is the level of the node in XML tree. A global variable is maintained which is incremented whenever a node is visited. The given XML data tree is traversed in DFS order and the global variable is assigned to *start* of the node when the node is visited for the first time and to the *end* of the node when the node is revisited. An example XML document with interval encoding is given in Figure 1. One advantage

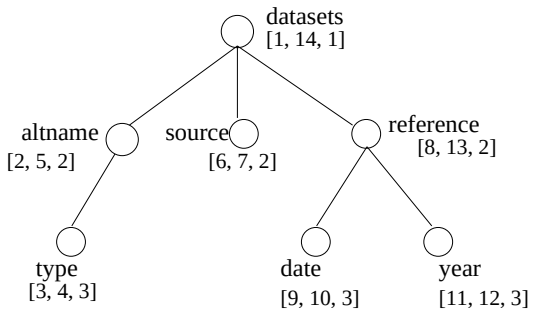

Figure 1: Illustration of interval encoding

with the interval encoding is that we can determine ancestor-descendant relationships and parent-child relationships easily. Let a and d be two nodes along with their encodings $<a.start, a.end, a.level>$ and $<d.start, d.end, d.level>$ respectively. a is ancestor of d iff $a.start < d.start$ and $a.end > d.end$. a is parent of d iff $a.start < d.start$ and $a.end > d.end$ and $a.level = d.level - 1$

This encoding alone is not efficient for query evaluation as the number of joins needed increases with the size of the query. For example, to evaluate the query $//datasets/reference/other/year$ we need to retrieve the lists of *datasets* nodes, $reference$ nodes, *other* nodes and *year* nodes and then apply join among the lists. This query needs 3 joins to get the result.
The number of joins needed to evaluate a query using interval encoding alone is (*Number of tags in the query - 1*)

2.2 BLAS:

This section gives the details of BLAS[2], which uses P-Labeling scheme to reduce the number of joins when consecutive parent-child operators are present in the query.

Terminology:
Source Path: *Source path* of a node is the path from the root of XML data tree to the node.
Suffix Path Expression: Any path expression that optionally starts with ancestor-descendant operator and is followed by any number of parent-child operators is called *Suffix path expression.*
Simple Path Expression: A path expression that has only parent-child operators is a *Simple path expression* i.e., a suffix path expression that doesn't start with ancestor-descendant operator.

2.2.1 P-Labeling Scheme:

The idea behind P-Labeling is to associate a unique label to every possible path in the given XML data tree. Let there be n distinct tags, say $t_1, t_2, ..t_n$ in the XML document. Here $'/'$ is assigned 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 in 0 to n. Let h be the length of longest path in the XML document. The range of P-Labels needed is $[0, m-1]$, where $m \geq (n+1)^h$. Note that the ordering among the tags $t_1, t_2, ..t_n$ is important. The labeling scheme is completed in $h+1$ steps.

In the first step, the range $[0, m-1]$ is divided into $n+1$ parts according to ratio vector $r_0, r_1..r_n$. The first partition $[0, m * r_0 - 1]$ is assigned to $'/'$ and the each remaining partition $[m * \sum_{j=0}^{i-1} r_j, m * \sum_{j=0}^{i} r_j - 1]$ is assigned to $//t_i$, where $i = 1$ to n. This means all paths that end with t_i have their P-Labels in the

range $[m * \sum_{j=0}^{i-1} r_j, m * \sum_{j=0}^{i} r_j - 1]$.

In the second step each such partition, except the first one, is again divided into $n + 1$ parts according to ratio vector as in first step. The partition $[m * \sum_{j=0}^{i-1} r_j, m * \sum_{j=0}^{i} r_j - 1]$ is divided into $n + 1$ parts and the partitions are assigned to $/t_i, //t_1/t_i, //t_2/t_i, ..., //t_n/t_i$ respectively where $i = 1...n$.

It is important to observe that the first partitions at each step i correspond to simple path expressions of length $i - 1$ and remaining partitions correspond to suffix path expressions of length i. In $(h + 1)^{th}$ step, all the first partitions are assigned to simple paths expressions of length h and the remaining partitions need not be considered as their length exceeds h.

Example: Let there be 3 distinct tags in XML document. Let the length of longest path in the XML document be 3. Let $m = 200$. Let the ratios be $r_i = 1/(n + 1) = 1/4$.

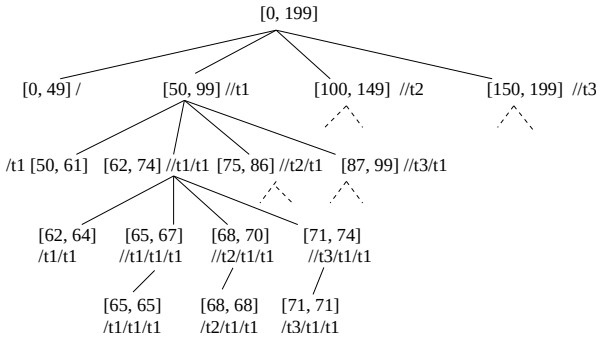

Figure 2: Illustration of P-Label scheme

In the first step the range [0, 199] is divided into 4 parts as shown in Figure 2. The first partition [0, 49] is allocated to $'/'$. The second partition [50, 99] is allocated to $//t_1$ and so on. In the second step each such partition is again divided into 4 parts. For example the partition [50, 99] is divided into four. The first partition [50, 61] is allocated to $/t_1$. This means that the simple path $/t_1$ has P-Label range [50, 61]. The second partition [62, 74] is allocated to $//t_1/t_1$. So all paths that end with t_1/t_1 have their P-Labels in the range [62, 74]. Observe that all first partitions are assigned to simple path expressions and are not partitioned further.

2.2.2 P-Label Construction for XML Tree:

The XML document tree is traversed in Depth First Search order and for each node encountered, the interval encoding $< start, end, level >$ and P-label p is found. To determine P-Label of a node, the P-Label range $[p1, p2]$ for its source path is determined and $p1$ is assigned as the P-label to the node. The algorithm to generate P-Labels of nodes in XML document is given in Algorithm 1.

Algorithm 1 Algorithm to generate P-labels of nodes in the XML document

Procedure: P-label(XML tree: T)

```
1: Stack s
2: for all i do
3:    < p_{i_1}, p_{i_2} >=< m * sum_{j=0}^{i-1} r_j, m * sum_{j=0}^{i} r_j - 1 >
4: end for
5: push(s, < 0, m - 1 >)
6: Depth-first search(T){
7: if current tag is < t_i > then
8:    < p_1, p_2 >= top(s)
9:    p_1 = p_{i_1} + p_1 * (p_{i_2} - p_{i_1} + 1)/m
10:   p_2 = p_{i_1} + (p_2 + 1) * (p_{i_2} - p_{i_1} + 1)/m - 1
11:   push(s, < p_1, p_2 >)
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: }
```

2.3 Holistic Join Algorithms

Holistic join algorithms were proposed in [8] which out perform binary structural join algorithms. Holistic path join algorithm PathStack generalizes the Stack-Tree-Desc binary structural join algorithm of [10]. Holistic twig join algorithm TwigStack refines PathStack to ensure that results computed for one root-to-leaf path of twig pattern are likely to have matching results in other paths of the twig pattern. TwigStack merges results for the different root-to-leaf paths in the query twig pattern to compute the desired output.

3 MQEB

In this section we present a new system MQEB, in which we propose algorithms that use metadata for efficient query evaluation.

Valid P-label: A P-label is said to be valid if there exists at least one node with that P-label in the given XML document. In a typical XML document we observe the following:

- Even though the P-label range is large, the number of valid P-labels is very very less. We assign a P-label to every possible path. Even though there are $(n + 1)^h$ possible paths the number of distinct paths that occur in a typical XML document is very very less. For example, in SWISSPROT dataset the range of P-labels

needed is $[0, 10^{10}]$. But the number of valid P-labels is 264.

- There exist considerable number of nodes with same P-label. Since XML is semi-structured, considerable numbers of nodes occur in same context i.e. many nodes will have the same source path. For example, in SWISSPROT dataset there are nearly 2,977,031 elements and 2,189,859 attributes. But the number of distinct paths in the dataset is 264. So considerable number of nodes exist with same source path.

3.1 Query Evaluation Using Metadata:

Motivated by the above observations, we present MQEB that uses metadata for efficient query evaluation. While creating P-labels of the nodes in XML document, we store all distinct valid P-labels along with their *paths* as metadata.

3.1.1 Suffix Path Queries

Suffix path expressions are evaluated as in BLAS. To evaluate a simple path (that has only parent-child operators) query, we first find the P-label range $[p1, p2]$ for the simple path and retrieve all nodes with P-label $p1$. To evaluate a suffix path query that start with $'//'$, we find the P-label range $[p1, p2]$ for the suffix path and retrieve all nodes whose P-labels fall in that range. The method of finding P-label range for a suffix path expression is given in Algorithm 2.

Algorithm 2 Algorithm to find P-label range of a suffix path expression

Procedure: P-labelForPath($Ol_1/l_2/..../l_n$ where $O =' /'$or $'//'$)

1: **for** $i = n; i \geq 1; i--$ **do**
2: Find t_j such that $t_j = l_i$
3: $s_1 = p_1 + (p_2 - p_1 + 1) * \sum_{k=0}^{j-i} r_k$
4: $s_2 = p_1 + (p_2 - p_1 + 1) * \sum_{k=0}^{j} r_k - 1$
5: $p_1 = s_1$
6: $p_2 = s_2$
7: **end for**
8: **if** $O == /$ **then**
9: $p_2 = p_1 + (p_2 - p_1 + 1) * r_0 - 1$
10: **end if**
11: return $< p_1, p_2 >$

3.1.2 Complex Path Queries

A complex path query is one that optionally starts with an ancestor-descendant operator and is followed by at least one ancestor-descendant operator. In BLAS, complex path query is split up into suffix path queries which are evaluated as above and the resultant lists are joined using holistic twig join algorithm. For example, $//dataset/reference//source/year$ is split into $//dataset/reference$ and $//source/year$ and each part is evaluated and join algorithms are applied to the resultant lists to get the final result.

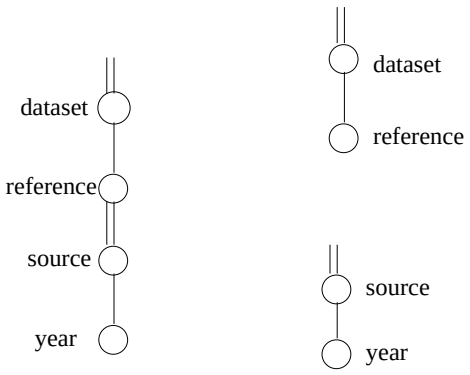

Figure 3: Processing complex path queries in BLAS

Where as in our system, MQEB, complex path queries are evaluated as follows: Query is split into two parts at the last ancestor-descendant operator. The first part $//datasets/reference$ is called Prefix part and the second part $//source/year$ is called Suffix part. we find the P-label range $[p1, p2]$ for the suffix part and retrieve all the valid P-labels in the range $[p1, p2]$ from meta-data. For each of these P-labels, we check whether the prefix part exists in the path corresponding to the P-label. If it exists, we retrieve all nodes with the corresponding P-label, else, we discard it. The procedure for evaluating path expressions is given in Algorithm 3.

Algorithm 3 Algorithm to evaluate a path expression

Input: $o_1t_1o_2t_2...o_rt_r$ where o_i is an operator and t_i is a tagname
Output: Retrieve all nodes that satisfy the input path expression.
$//path(p)$ returns the path corresponding to the P-label p.
$//match(prefixPart, path(p))$ checks if the prefixPart satisfies the path corresponding to p.
Procedure:evalPathExp($o_1t_1o_2t_2...o_rt_r$)

1: Split the path expression $o_1t_1o_2t_2...o_rt_r$ into two parts at the last ancestor-descendant operator o_k.
2: $prefixPart = o_1t_1o_2t_2...o_{k-1}t_{k-1}$
3: $suffixPart = o_kt_k..o_rt_r$
4: Find the P-label range for the $suffixPart$. Let it be $[p1, p2]$
5: **for** each valid P-label p in the range $[p1, p2]$ **do**
6: **if** $match(prefixPart, path(p))$ **then**
7: Retrieve all nodes with P-label p
8: **end if**
9: **end for**

Procedure match() is given in Algorithm 4. In this, Prefix part is taken in a linked list with each node containing the name of the tag and preceding operator(P or A). Path is taken in another linked list with each node containing the name of the tag. Let $prefixPart$ and $path$ point to the heads of prefix part and path linked lists respectively. Let $lastAncInPrefix$ be the pointer to the recently encountered node in prefix part linked list with operator A. Let $lastAncInPath$ be the pointer to the corresponding match node in path linked list. If the current node in prefix part list has operator P, then we keep on comparing the successive nodes in both the lists. If the comparison fails, then we track back to $lastAncInPrefix$ and $lastAncInPath$->$next$ and start matching again(lines 4 to 16). If the current node in prefix part linked list has operator A, then we traverse the path list until we find a match to the current node in prefix part list(lines 17 to 24).

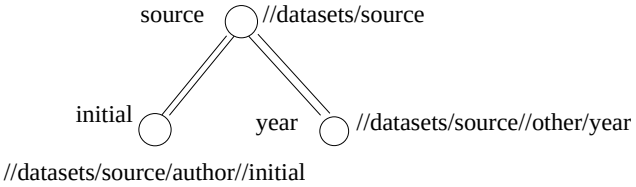

Figure 4: Processing of branch queries in MQEB

Since the number of valid P-labels in the P-label range of the suffix part is very very less, the time taken to do the matching of valid P-label paths with prefix part is considerably less when compared to BLAS case in which joins are needed. Moreover, since the metadata is small it can be maintained in memory so that the time taken for $match()$ method will be less. For example, the size of metadata for SWISSPROT dataset is 6KB. So for path expressions I/O disk access is done only to retrieve nodes that are part of final result. Thus any path expression is evaluated without a single join but with some processing done in $match()$ method to check if the valid P-label path satisfies the prefix part. In BLAS, the path expression is split into suffix path expressions and each suffix path expression is evaluated and the resultant lists are joined using holistic join algorithm. So a number of nodes that are not part of final solution are retrieved which are eliminated by join algorithm. Where as, in our case all the nodes that are retrieved are part of final solution i.e., no unnecessary node is retrieved.

3.2 Processing Branch Queries

In BLAS, to evaluate a branch query, the query is segmented into suffix path expressions by splitting at branch points and ancestor-descendant operators. Each of these suffixpath expressions is evaluated individually and the results are joined using holistic join algorithm. Algorithms are presented in [2] to split a query at branch points and ancestor-descendant operators. For example, the query shown in Figure 5(a) is split into four parts as shown in Figure 5(b). Each part is evaluated by taking the suffix path expression specified at the partition. Then holistic twig join algorithm is applied to the resultant lists to get the final result.The number of joins needed to evaluate a query is given by

No. of ancestor-descendant operators + No. of outgoing branches not annotated with '//'

Algorithm 4 Algorithm to match the prefixPart with Path

Input: prefixPart and path
Output: Returns TRUE or FALSE depending on the match
Procedure: match(Node * $prefixPart$, Node * $path$)

```
1:  lastAncInPrefix = NULL;
2:  lastAncInPath = NULL;
3:  while (prefixPart ≠ NULL && path ≠ NULL) do
4:    while (prefixPart->operator=='P' && prefixPart ≠ NULL && path ≠ NULL) do
5:      if strcmp(prefixPart->name,path->name) then
6:        prefixPart = prefixPart->next;
7:        path = path->next;
8:      else
9:        if lastAncInPrefix == NULL then
10:         return FALSE;
11:       else
12:         prefixPart = lastAncInPrefix;
13:         path = lastAncInPath->next;
14:       end if
15:     end if
16:   end while
17:   if (prefixPart->operator=='A' && prefixPart ≠ NULL && path ≠ NULL) then
18:     while (!strcmp(prefixPart->name, path->name) && prefixPart ≠ NULL && path ≠ NULL) do
19:       path = path->next;
20:     end while
21:     lastAncInPrefix = prefixPart;
22:     lastAncInPath = path;
23:   end if
24: end while
25: if prefixPart == NULL then
26:   return TRUE;
27: else
28:   if path == NULL then
29:     return FALSE;
30:   end if
31: end if
```

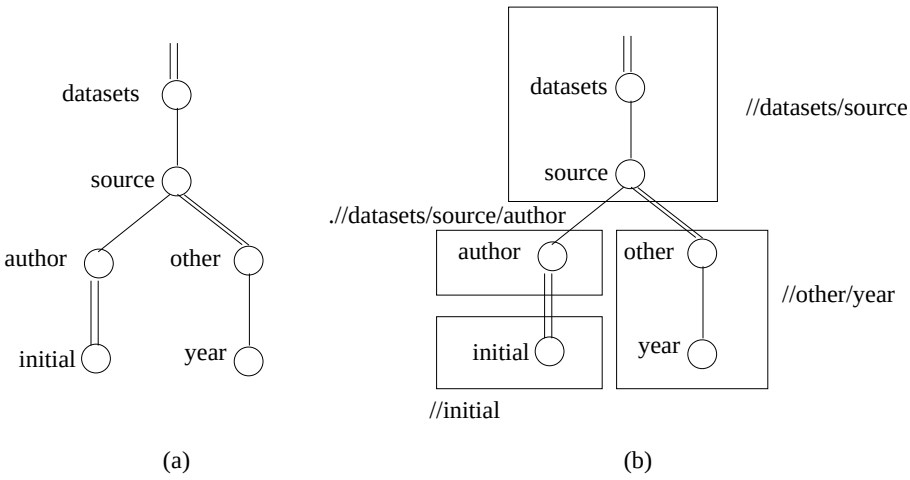

Figure 5: Processing of branch queries in BLAS

In MQEB, to evaluate a branch query, first the given query tree is reduced into one retaining only leaf nodes and branch points as well as the structural relationships among them. Now each node in the reduced query tree is evaluated using Algorithm 3 by taking the path from root to the node in the original query tree. Holistic join algorithm is applied to the reduced query node lists to get the final result. Algorithm to reduce given query tree is given in Algorithm 5.

For example, the query shown in Figure 5(a) is reduced to one shown in Figure 4, retaining only leaf nodes and branch points. Nodes *source*, *initial* and *year* are evaluated by taking path specified in the Figure 4. Now holistic twig join algorithm is applied on the above reduced query tree to get the final result. Here, we need not determine ancestor-descendant relationship between *year* nodes and *source* nodes because the *year* nodes are already evaluated by taking the entire path from root of the query to it. Same is the case with *initial* nodes also. The purpose of join algorithm is to find which *year* and *initial* nodes have a *source* node as their common ancestor.

Algorithm 5 Algorithm to reduce given query tree

Input: Root of the query tree
Output: Root of reduced query tree
$//q.noOfChildren()$: returns the number of children of node q.
$//q.child(i)$: returns the i^{th} child of node q.
Procedure: reduce(q)

```
1: while q is not a branch point and not a leaf do
2:    q = q.child(1);
3: end while
4: for i = 0 to q.noOfChildren() do
5:    reduce(q.child(i));
6: end for
7: return(q);
```

Algorithm 6 Algorithm to evaluate a query

Input: Root of query tree
Output: List of nodes that satisfy the input query
Procedure: processQuery(q)

```
1: for each node q' in the query do
2:    if q' is leaf node or branch point then
3:       evalPathExp(path from root q to q');
4:    end if
5: end for
6: q= reduce(q);
7: Twigstack(q);
```

The number of joins needed to evaluate a query is *Number of out-going branches in the query tree.*

In MQEB, we evaluate leaf nodes and branch points by taking the entire path from root of the query to the node. Where as in BLAS the path from nearest ancestor-descendant operator to the node is considered. As a result we get more qualified nodes than BLAS. Consequently the time taken for join algorithm is also less when compared to that in BLAS.

4 Implementation Details

4.1 Offline Processing

Offline processing involves parsing the XML document, index creation for the nodes and B^+-Tree creation for storing the node tuples. SAX parser is used for parsing the XML document. When SAX parser raises events, interval encodings and P-labels are calculated for the nodes. Attributes of a node are treated as children of the node. Data values are represented as leaves and are assigned the P-label of their parent. For each node we generate the tuple <*P-label, start, end, level, datavalue*>.Berkeley DB[6] B^+-Tree API is used for creating B^+-Trees to store the node tuples.

Table 1: Description of Data Sets

Dataset	Size in MB	No. of tags(n)	Max. height(h)	Size of P-labels
NASA	24	68	8	15 digits
SWISSPROT	115	99	5	10 digits

Table 2: Queries for NASA dataset along with no. of joins in each case

No.	XPath Query	BLAS	MQEB
Q1	//dataset/altname/type="ADC"	0	0
Q2	//dataset//tableLinks/tableLink//title	2	0
Q3	/datasets//reference//other//year=1936	3	0
Q4	/datasets//source[//author/initial][other//date]	4	2
Q5	/dataset[reference/source//year][tableHead//tableLink/title="The catalogue"]	4	2

Data structures : We use two-level B^+-Trees for storing the node tuples. All the distinct valid P-labels are inserted into top-level B^+-Tree. For the top-level B^+Tree:
Key is <*P-label*>
Data is <pointer to low-level B^+-Tree corresponding to the P-label>
For each distinct valid P-label we create a low-level B^+-Tree which is pointed by the corresponding P-label node in the top level B^+Tree. For the low level B^+-Tree:
Key is $< datavalue, start >$
Data is $< end, level >$
Since we use $< datavalue, start >$ as key in low-level B^+-Tree, value based queries can be processed efficiently. In case of internal nodes with no data value, the *data value* field will be $NULL$.

Metadata: Metadata contains the list of all distinct valid P-labels along with their corresponding path. Metadata is stored in a separate B^+-Tree with P-label as key and the corresponding path as data.

4.2 Online Processing

Online processing involves parsing the input query, P-labels generation and evaluation using the proposed algorithms. For joining, we use holistic twig join algorithm TwigStack proposed in [8].

5 Experimental Results

In our experiments, we compared the query performance of BLAS and MQEB for a set of queries on different XML datasets. We implemented BLAS and MQEB in C language. We used Berkeley DB B^+-Trees for storing the node tuples.

5.1 Experimental Setup

All experiments were conducted on 1.5GHz Pentium IV processor with 256MB RAM running Redhat Linux 7.2. Berkeley DB B^+-Trees were stored on secondary storage, 40GB hard disk.

Table 4: Time of execution for queries on NASA dataset

Q No.	Time(sec) in BLAS	Time(sec) in MQEB
Q1	0.002	0.002
Q2	0.091	0.048
Q3	0.028	0.004
Q4	0.045	0.022
Q5	0.047	0.019

5.2 Datasets

The datasets[5] used in the experiments are shown in Table 1.

5.3 Queries

The queries used in experiments are given in Table 2 and Table 3. some of these queries are suffix path expressions, some are complex path expressions and some are branch queries.

5.4 Performance Analysis

The number of joins needed for each query in BLAS and MQEB for NASA and SWISSPROT datasets are given in the Table 2 and Table 3 respectively. For the complex path queries and branch queries, the number of joins needed in MQEB is less than that in BLAS. Consequently the time taken for evaluation of queries Q2, Q3, Q4 and Q5 is less in MQEB than that in BLAS. The times for execution of queries in Table 2 and Table 3 using BLAS and MQEB are given in Table 4 and Table 5 respectively.

6 Conclusion

We proposed and implemented MQEB for efficient XML query evaluation. We created interval encoding labels and P-labels for the nodes in the XML document while parsing the XML document with SAX parser. We used Berkeley DB software to create B^+-Trees for storing node tuples as well as metadata of the XML document. We proposed and implemented algorithms to split a complex query into path expressions and evaluate them individually and then join the

Table 3: Queries for SWISSPROT dataset along with no. of joins in each case

No.	XPath Query	BLAS	MQEB
Q1	//Entry/Ref/Author="Baumeister"	0	0
Q2	/root/Entry//Features//DISULFID/from=31	2	0
Q3	/root//Features//CONFLICT/to=73	2	0
Q4	/root//Features[//CARBOHYD/Descr="POTENTIAL"][CHAIN/from=25]	3	2
Q5	//Features//DOMAIN[Descr][to=254]	3	2

Table 5: Time of execution for queries on SWISSPROT dataset

Q No.	Time(sec) in BLAS	Time(sec) in MQEB
Q1	0.004	0.004
Q2	0.033	0.005
Q3	0.021	0.005
Q4	0.025	0.021
Q5	0.065	0.043

results using holistic join algorithm. The proposed algorithms make use of metadata and reduce the number of joins as well as the size of input lists for join algorithms thus reducing the time for evaluation of query. We observed that MQEB performs significantly better than BLAS for the queries containing more number of ancestor-descendant operators.

We used B^+-Trees for storing the node tuples as the Berkeley DB software doesn't support XB-Trees. So we used TwigStack algorithm for join operation even though a better algorithm TwigStackXB[8] was available. Using XB-Trees for storing node tuples and TwigStackXB for join operation gives better performance.

References

[1] A.Berglund, S.Bong, D.Chemberlin, M.F.Fernandez, M.Kay, J.Robie, and J.Simon. XML path language(XPATH)2.0 W3C working draft. Technical report, World Wide Web Consortium, August 2002.

[2] Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: An efficient xpath processing system. In *SIGMOD*, pages 47–58, 2004.

[3] D. DeHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A comprehensive xquery to sql translation using dynamic interval encoding. In *SIGMOD*, pages 623–624, June 2003.

[4] P. F. Dietz. Maintainig order in a linked list. In *ACM Symposium on Theory of Computing*, pages 122–127, June 1982.

[5] http://www.cs.washington.edu/research/xmldatasets.

[6] http://www.sleepycat.com.

[7] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A queriable compression for xml data. In *SIGMOD*, pages 122–133, 2003.

[8] N.Bruno, N.Koudas, and D.Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, pages 310–321, June 2002.

[9] Q.Li and B.Moon. Indexing and querying XML data for regular path expressions. In *VLDB*, pages 361–370, September 2001.

[10] S.Al-Khalifa, H.V.Jagadish, N.Koudas, J.M.Patel, D.Srivastava, and Y.Wu. Structural joins: A primitive for efficient XML query processing. In *ICDE*, pages 141–152, Feb 2002.

[11] S.Bong, D.Chemberlin, M.F.Fernandez, D.Florescu, J.Robie, and J.Simon. XQuery1.0: An XML query laguage W3C working draft. Technical report, World Wide Web Consortium, August 2002.

[12] S. D. Viglas, K. Beyer, JayavelShanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD*, pages 204–215, June 2002.