A Dual-strategy Compact Structure Model for Indexing XML Data

Yonggong REN †, Jialin WU, Yanyu ZHANG

School of Computer and Information Technology, Liaoning Normal University, Dalian 116029 China

Abstract

With data increasing continually, data of XML are queried complexly. Several years, there are several algorithms, for example, TJFast which is by scanning repeatedly and make a lot of middle routes in the process of merging; the XML datasets are too uncertain to impose the efficiency of the query well by filtering. In our article, a new efficient index structure—FSM is proposed, which can avoid merging the unnecessary routes, and impose the efficiency of filtering. Through a lot of experiments, the efficiency of FSM is better than the former ones, and FSM also imposes the query performance.

Keywords: Query Optimization; Index Structure; Merging

1. Introduction

XML can swap data by web, because of the feature of XML semi-structured. At the same time, XML Database can connect with HDB Heterogeneous Database, but with the increasing of the data quantity, data query is becoming more difficult. The method which is travelling the nodes of the XML Database effects the efficiency of the query. So the problem about the XML query has been followed extensively, and becomes a focused problem in the XML Database.

There have been a lot of excellent query algorithms, recent years. Such as the document proposed an integrated twig pattern ---TwigStack[1], and this method can get query results by means of merging the part of matching results. But its cost is that it needs several accessing the middle results. The document proposed a link algorithm—PathStack[2] which is based on the node stream and stack-link. The document proposed a XR tree index[3]. It can skip over all of the unmatching nodes in the descendant list and the ancestor list. the query efficiency is influenced, because of the uncertainty of XML data, and merging the unnecessary routes unavoidably, such as 1-index[4] and F&B-index[5]. These methods impose the query efficiency by filtering nodes and reducing the quantity of nodes in the structure. Our experiments reveal that filtering nodes can not increase the query efficiency effectively, because of the uncertainty of XML data.

There are two keys in the query XML document: (1) How to avoid merging the unnecessary routes. (2) How to resolve the problem of the query efficiency unconspicuously by filtering nodes, when the XML

† Corresponding author.

Email addresses: renyonggong@gmail.com (Yonggong REN)
document structure is complex, and the repetitive routes are less. About these two problems, this paper proposes a new algorithm—FSM. Using the route which has been decomposed by the entry path Vn, get the set Qn by pattern matching the XML document. Then FSM decomposes and filters the route of the set Qn, and gets the query results at last. The whole process effectively solves these problems.

2. FSM Encoding

In order to impose the query performance, we encode the XML document. Then we can quickly ascertain the relation of any two nodes in the XML tree structure.

A XML document tree encoding scheme can be divided into two categories: region-based encoding scheme and path-based encoding scheme. The encoding scheme mainly includes: bit-vector encoding[9], prefix encoding[10], region encoding[12], PBiTree encoding[11] and so on. The method of region encoding is that it gives nodes a region encoding [start, end], and then every node’s region encoding also has a region encoding of its descendant, namely a node u in tree T, its ancestor is v, if and only if start(v)<start(u)∧end(u)<end(v).

FSM uses <start, end, level, parent> to encode the XML document. Start represents ascending sort. Level represents the level of node, Parent represents ID of the node’s father, and parent can help us find the node’s father directly.

3. FSM Filtering

The main purpose of filtering nodes is to filter out the repetitive route in the XML document. So it can make avoiding querying the repetitive routes, decreasing nodes total of the structure and compacting the XML document. The method which FSM uses is based on FC-Index which is proposed on the document [13]. After estimating father and descendant of the node, FC-Index judges whether to filter nodes. Fig.1(a) is a XML tree. Fig.1(b) is the XML document structure which gets by using FC-Index to filter Fig.1(a).

**Property 1.** The storage structure of FC-Index travels the XML document by depth-first traversal. If ID of any node Q is greater than its pre-node’s ID and its next node’s ID, Q is a leaf node, and it is the last node on the route.

![XML tree](image1.png)

![FC-Index](image2.png)

Fig.1 Query Structure

Fig.1(b) is the tree structure which is filtered by using FC-Index. Its storage structure is A1/A2/B1/D6/C2/F5/C3/A1/B2……D5. If ID of C3 is greater than all of its pre-node’s node, and less than ID of its next node A1, C3 is a leaf node, and is the last node on the route M= A1/A2/B1/D6/C2/F5/C3.
4. Implementation of FSM

The XML document is encoded <start, end, level, parent> by FSM, and filtered by FC-Index. Then decompose the entry path and use these to match the XML document which has been filtered. Then put the results into the relevant subset Qn. Get rid of the repetitive routes in Qn. At last, combine the subsets and get the results.

4.1. Decomposing and Matching

Definition 1. If a query route Q=A[D/F]/B/C, the communal node is called the crossed node.

By using the crossed node of the query route, decompose the multiple route to the simple routes, until there isn’t the branch route in the query route. If there are several crossed nodes, decompose it from a small beginning into a mighty on the basis of ID of the crossed node.

For example Fig.2, Path Q=A[D/F]/B/C, and based on Definition 1, node A is a crossed node, so Path Q can be decomposed into A/D/F and A/B/C. In this paper, “/” denotes father relation, and “//” denotes forebearsdescendants relation.

Definition 2. If decompose the entry path into the simple routes, the first node of the simple route is called head-node.

After decomposing, allocate a storage space for every simple route. In this paper, this storage space is called subset which stores the path which matches with the simple route.

Filtering the XML document, and based on Definition 1, the first node of the simple route regards as the root-node. Travel the XML document deeply and match. Put the results into the relevant subset. For example, Fig.3 is a result which Fig.2 matches with Fig.1. Subset Q1 and Q2 are the results of Path W=A/B/C and Path E=A/D/F respectively.

4.2. Getting Rid of the Repetitive Route

Property 2. For Path u=A/B/C/D, the result which is got by using Path B/C/D and Path C/D is as the same as by using Path u. Namely, any sub-path of u is the repetitive route of u.

Think about that the simple path may appear several times in the XML document, so it can get a lot of results from one path in the process of matching. But these are the same results—the repetitive results. Fig.3, in the subset Q1, Path a=A2/B1/C2 and Path b= A1/A2/B1/C2 are all the query results of Path A//B//C. But Path a is a part of b, so Path a is the repetitive result. So FSM will filter every subset, before merging subsets.

In order to impose the efficiency of getting rid of the repetitive route and merging, FSM decomposes every route of the subset before filtering. Based on ID, every two nodes are stored in one block. The
block’s ID is in the minimum ID of nodes in the block. But the first block of every route have two ID. They are the head-node’s ID and the tail-node’s ID. The last block of every route also has two ID which is ID of the nodes in this block. Then store them in the quondam subset. Fig.4 is got from Fig.3 by decomposing and merging subset Q1.

After that, the length of the route becomes shorter. In the process of filtering, we can do that with the ID of the block directly. Our experiments reveal that the efficiency by decomposing and merging is better than without these.

Algorithm 1. remove_overlap()

Input: The subset Qn which is got from matching XML document

Output: The subset Qn without the repetitive routes

1. while(empty(Qn))
2. Createlist(Qn);        //decompose the route in the subset.
3. for(i<n)
4. Listi1=GetList(Q1); Listi2=GetList(Q1);  //get the routes from the i-th subset.
5. if(IS_overlap(Listi1,Listi2))  //judge whether List1 and List2 are the repetitive routes.
6. then(delete_list());        //delete the repetitive routes.
7. else GetList(Next(Q1));      //get the next route.
8. end while

Algorithm 1 is the core part in the process of filtering. In line 2, the purpose of Createlist() is to decompose the route in the subset, and gets the route. In line 2 and 3, by executing GetList(), get one route from the subset. Then judge whether these routes are the repetitive routes by executing IS overlap() in line 5. If it is true, execute delete_list() delete them, or execute GetList() get the next route in line 7, and continue to judge that.

4.3 Merging the Subset

Every subset is got from the simple route, so only need merging these subsets. But if going to merge, there must be a communal node in the route. We can get it by judging whether there are the nodes which have the same ID.

Property 3. If Path y in the subset Q and Path x in the subset R have a communal node, and the
head-node’s ID of y is \((y_1, y_2)\) and the head-node’s ID of x is \((x_1, x_2)\), \((y_1, y_2) \cap (x_1, x_2)\) must get an integer, and this integer is ID of the communal node.

Store ID \((a_1, a_2)\) in the head-node. Based on Property 3, we can judge whether there is a communal node, then avoid to operate the unnecessary results. Such as Fig.5, Path a in the subset List1 merges with Path b and Path c in the subset List2 respectively. The head-node’s ID of a, b and c are \((2,8),(1,9)\) and \((2,11)\). There has a communal integer 2 between a, b and c. It means that there is a communal node B2 which ID is 2. So a can merge with b and c respectively.

Algorithm 2. find_result()

Input: The subset Qn after merging

Output: Query results

1. while(\(\emptyset(Q1)\))
2. \(Vk=GetList(Q1); Vg=GetList(Qn)\);
3. if(IDList(Vk) \cap IDList(Vg)) then  //if have a communal node between these two route.
4. \(K=FindNode()\);  //get the communal node.
5. GetResult(O);  //merge and get query results.
6. else
7. \(Vg=GetList(Qn)\);  //merge the next route in the subset Qn.
8. end while

Algorithm 2 returns query results. In line 1 and 2, execute GetList() and get the route \(Vk\) from Q1 and the route \(Vg\) from Qn. In line 3, we can judge whether there has a communal node by executing IDList(). If there has, execute line 4 and 5. Get the communal node K of Vk and Vg by executing FindNode. Using K, execute GetResult(), merge Vk and Vg and get the query results. If we don’t get K, execute line 8, and continue to merge the next route in the subset Qn.

5. Performance Evaluation

5.1. Experimental Data

In order to test the properties of FSM, Experiments were run on a 2.66 GHz PC-compatible machine with 1GB of RAM running Windows XP. We implemented FSM, TwigStack and TwigList[8] in VC++6.0 for XML indexing. To validate the efficiency of FSM, we tested two datasets XMark[6] and TreeBank[7]. XMark is got from Data Generator, 113MB size and has 1 hundred million elements. TreeBank is 82MB size, and has more than 2 hundred million elements. Experiments are divided into two groups. The first
group used XMark, and divided into 5 query tests. The second group used TreeBank, and divided into 4 query tests. We used queries as shown in Table 1.

Table 1: Queries Used in the Experiment

<table>
<thead>
<tr>
<th>Query NO.</th>
<th>Dataset</th>
<th>XPath expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>KQ1</td>
<td>XMark</td>
<td>site/item//description//text//bold//name</td>
</tr>
<tr>
<td>KQ2</td>
<td>XMark</td>
<td>site/africa//mailbox/mail//date//description//text</td>
</tr>
<tr>
<td>KQ3</td>
<td>XMark</td>
<td>site/item/name//mailbox//date</td>
</tr>
<tr>
<td>KQ4</td>
<td>XMark</td>
<td>site/open_auctions//bidder//time/personref</td>
</tr>
<tr>
<td>KQ5</td>
<td>XMark</td>
<td>site/regions//mailbox//text//name</td>
</tr>
<tr>
<td>BQ1</td>
<td>TreeBank</td>
<td>//S[NP][JJ]//VP/VBN</td>
</tr>
<tr>
<td>BQ2</td>
<td>TreeBank</td>
<td>//S[NP]//VP//VBN//PP</td>
</tr>
<tr>
<td>BQ3</td>
<td>TreeBank</td>
<td>//S[NP][DT]//PP//IN</td>
</tr>
<tr>
<td>BQ4</td>
<td>TreeBank</td>
<td>//S[NP]//VP//NP//JJ</td>
</tr>
</tbody>
</table>

5.2. Experimental Results and Analysis

We used FSM, TwigStack and TwigList to match XMark and TreeBank, and got the time which was shown on Fig.6. From Fig.6, we can find that FSM is more efficient than the others. In the XMark, it used less time to query, because of the relative simple structure of XMark. Fig.6(a) shows that FSM uses 20.78ms which is the maximum time, and TwigStack and TwigList use 80.25ms and 22.65ms which is the minimum time. In the relatively large number and complicated structure database---TreeBank, tested the query. FSM is also more efficient. On Fig.6(b), The time which TwigStack takes is the time which FSM takes 4.8~20.5 times, and The time which and The time which TwigList takes is the time which FSM takes 4~14.5 times. From

![Fig.6 Comparison of Running Time](image)

The experimental tests, the efficiency of TwigStack isn’t well, because of merging. And it creates a lot of middle routes in the process of query both TwigStack and TwigList, so it must merge these unnecessary routes. Experiments show that FSM can effectively resolve the problems which are the unnecessary routes by merging and the query efficiency unconspicuously by filtering nodes, and impose the query performance.
6. Conclusions and Future Work

In this paper, we research that how to impose the query efficiency of the XML document. Especially, think about the problem which is the unnecessary routes by merging and the query efficiency unconspicuously by filtering nodes. And study these problems deeply and propose a new algorithm—FSM. Through the experiments show, FSM can effectively resolve these problems and impose the query performance.

Acknowledgement

The authors would like to thank National Natural Science Foundation of China (Grant No.60603047), Science and Technology Plan Projects of Liaoning Province (Grant No.2008216014), Science Foundation for The Excellent Youth Scholars of Dalian (Grant No.2008J23JH026), The Project-sponsored by SRF for ROCS, SEM.

References