ISSN No. 0976-5697

Volume 3, No. 1, Jan-Feb 2012

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Optimization of Non Structured Query in Semantic Web by use of the Monitor Structure

Ahmad Kazemi*
MS Student mechatronic, Islamic Azad University, Majlesi Branch, Iran
Ahmadkazemi_2006@yahoo.com

Dr. Mehdi Sadeghzadeh
Professor Islamic Azad University, Mahshahr Branch, Iran
Sadegh_1999@yahoo.com

Dr. Amir Masoud Rahmani
Associate Islamic Azad University, Science and Research Branch Tehran, Iran

Dr. Houshang Kazemi
Professor Islamic Azad University, Majlesi Branch, Iran

Abstract: web queries are based on path expressions which are equal to a combination of some elements connected to each other in a tree pattern structure, called *query tree pattern (qtp).* The main operation in web query processing is to find the nodes that match the given qtp in the document. A number of methods have offered for qtp matching; but the majority of these methods process all of the query nodes to access all qtps in the document. A few methods such as *tjfast* process only nodes that satisfy leaves of qtp. All of above methods are trying to find a way just to optimizing direct comparing of nodes and to find the answer of query, directly via these comparisons. In this paper, we describe a novel method to find the answer of query without access to real data of the document blindly. In this method, first, the query will be executed on query guidance and this leads to a plan. Using this plan, it will be clear how to process leaf nodes and how to achieve query results, before processing of the document nodes. Therefore, none of document nodes will be processed blindly.

Keywords: Web, Twig Joins, Tjfast, Path Indexes And Evaluation.

I. INTRODUCTION

Query processing is an essential part of any WEB database. Both *XPath* and *XQuery*, the two most popular query languages in WEB domain, are based on *path expressions*. A path expression specifies patterns of predicates selection on multiple elements that has a tree structure named *Query Tree Pattern (QTP)*. Consequently, in order to process WEB queries, all occurrences of QTP in the WEB document should be found. This is an expensive task when huge WEB documents are involved. Consider the following query: *Q1: // A[.//B]//C//D;* The structure of an WEB query could be shown in a QTP, for example the QTP of query Q1 is presented in Figure 1.

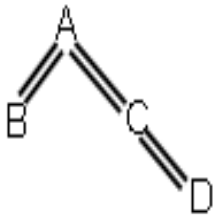

Figure 1. QTP of query Q1

The aim of all WEB query processing methods is to find all QTP instances in the WEB document. A number of methods are proposed to answer queries like *Q1*. We classify these methods into three groups:

Group A: Methods in this group are based on a famous method named *Structural Join* [1]. In structural join, query is decomposed into some binary join operations. Thus, a huge volume of intermediate results are produced in these methods.

Group B: *Holistic twig join* methods [2] do not decompose the query into its binary *Parent-Child (P-C)* or *Ancestor-Descendant (A-D)* relationships but they need to process all of the query nodes in the document.

***Group C*:** It is better to process only nodes that satisfy leaves of QTP. TJFast [12] is such a method.

Three Groups above called *containment joins*. Containment join methods use an index named *Name Indexes* to quick access to elements which have same tag name. for example to answer *Q1*, this index makes it possible to access to all *A, B, C* and *D* nodes in the document; but all of methods above, do not consider the place of elements. They are trying to find a way just for optimizing direct comparing of nodes and to get the answer of query, directly via these comparisons whereas many of these comparisons do not produce any part of the query answer.

On the other hand, there are some *path indexes* like Strong *DataGuide, Fabric Index, ToXin, APEX, Index1, A(k) Index,* and *F&B* which are indexing the path of document's nodes to facilitate access to nodes required in WEB query processing methods [3] [6] [7] [9] [10] [13][14].

These path indexes are other kinds of query processing methods which are against the *A, B* and *C* group methods. Path indexes usually have two parts:

a. Structural Summery (SS) that summarizes document structure and describes relation between elements.

b. Extend that keeps real data of the document based on Structural Summery.

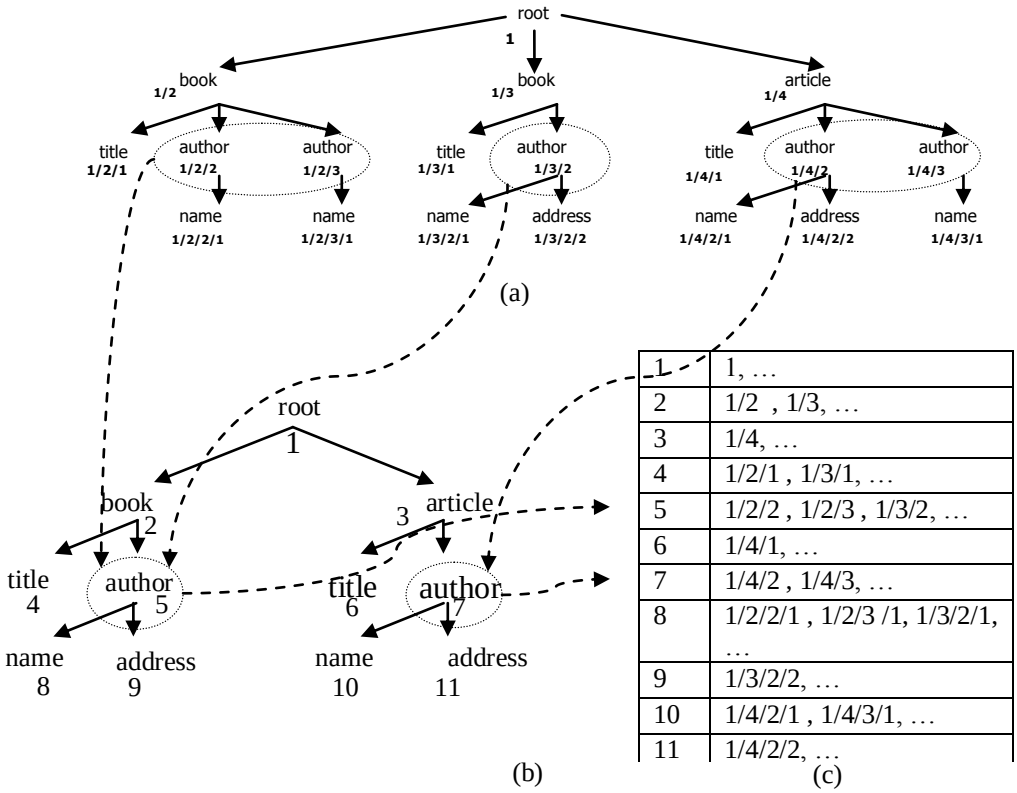

Figure 2. (a) A sample WEB document (b) Its Structural Summary (c) It's extend

Both of them are shown as *b* and *c* in figure 2. All of methods in this group behave as follows: At first, structural relationship (A-D or P-C) between query elements are compared with Structural Summary. As result, Extends of nodes that match with query is returned. For example in Match Processing of query *Book/author/name* on SS in figure 2, node number *8* matches with query. Therefore, all of its Extends will be returned as results. This method is considerable because it apply query on a small set named SS and to execute the query it doesn't need to access to real data of the document; But always queries are not such simple. For example to answer the queries such as *a//b[c]* or *a[.//b]/c* they need to access real data of the document. Therefore, this method has not enough performance.

None of the *containment join* methods uses full potential of path indexes or structural summaries, while these have great potential to guide us to sighed processing.

In this paper, we propose a compound method that uses structure summary as query guidance. In this method, query will be executed on structure summary that has very small size in comparison with the document. For this purpose, there is no need to access to real data of the document. Result of this execution is generation of a plan called *Monitor Structure (MS)*. MS shows leaf nodes of the query and the way of their processing in the document. This save us from direct and blind processing in the document

II. OVERVIEW OF OUR METHOD

Our method is similar to both *Containment Joins* and *Path Indexes*. In this method, we apply the query on Structural Summery of the document. SS is similar to schema of a document and has not close relation with size of the document. Its size and structure are usually stable or with a few variation.

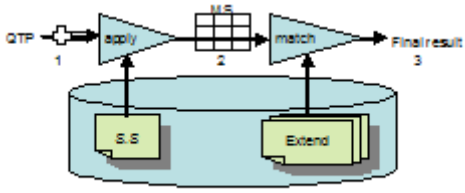

Figure 3. An Overview of Apply Query on SS.

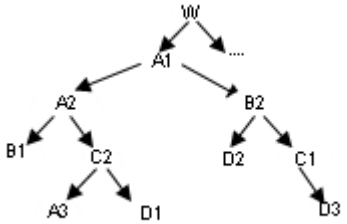

Figure 4. A Sample Structural Summary

***Step1*:** as shown in figure 3 and like Path Index methods, first, query is applied on SS; but here the query is not executed in its complicated form. It will be split in several single-branch queries that will be easily answered in all methods of path indexes [3] [6] [7] [9] [10] [13] [14].

Step2: all single-branch queries execute on SS separately. A plan that called Monitor Structure is build from execution result of single-branch queries. MS as process guidance shows the leaf nodes that are to process and the way of processing them in the document.

Step3: The document is numbered base on Dewey encoding.

Definition : In Dewey labeling method if node *U* is the n^{th} child of node *V*, the Dewey code of node *U* is the Dewey

code of node *V* as its prefix continue with *n*, Dewey(*U*)=Dewey(*V*)+'.'+'*n*'. For example suppose that Dewey(*V*)=<*1.3*> and node U_1 is the 7^{th} child of node *V*, then Dewey(U_1)=<*1.3.7*>

As shown in *C* of figure 2, Based on Dewey numbers, all nodes corresponding to each node of SS are sorted in Extend. Third step is similar to Containment Joins methods. Based on MS, leaf nodes of query that are placed in Extend will be compared and final result will be generated.

A. *Query Splitting And Execution Of Single-Branch Queries On Structural Summary:*

Queries are usually complicated and multi-branch. Before splitting a query in several single-branch queries, we should be familiar with *Joint point* concept.

- ***a. Definition-JP:*** Joint Point is a node in QTP which joins more than one branch to each other.

Example: suppose *A* and *B* are two branches of a query that have traversed path from query root *a1/a2/.../aj/ax1/.../an* and *a1/a2/.../aj/ax2/.../am* and *ax1* ≠ *ax2* then *J* is joint point of two branches with *a1/a2/.../aj* as its path. We do not mean parent-child relation by / between query elements and it can be interpreted as /, //, *.

To answer the multi-branch queries, we need to find joint points of branches that called JP. Complexity of multi-branch queries is because of JPs. We can easily find place of these nodes on SS; but we cannot definitely answer to this kind of query without access to the document. Query condition is as follow: a JP in a document is part of answer if it has all of query branches under itself, in other words, several query branches in the document can be part of answer if they are conjoint in same JP. This JP cannot be found just with access to SS and without comparing of branches in the document; because it is possible that one JP in the document has not one of query branches under itself.

Example: in *Q1*, *A* is joint point of two branches, *A//B* and *A//C//D*. in *Q1*, *A* nodes in document are part of answer if have both of *A//B* and *A//C//D* branches.

- ***b. Splitting Query:*** suppose *Q* is a multi-branch query with *n* JPs and m branches (leaves). *Q* split in single-branch queries $SQ_1, ..., SQ_m$ so that each SQ_i is a branch from root to leaf of one of branches and every two of SQ_i and SQ_j have same prefix from root to one of the JPs. Total Number of these different JPs is *n*.

Here our goal is description of algorithm functionality. For this reason, we explain our method on simple query of *Q1* and then we show how MS can answer to complicated queries.

- ***c. The procedure*:** As mentioned above, at first, we must split query. Query split into single-branch queries. Then each single-branch query will be executed on SS separately. Fortunately, in most of path index methods single-branch queries can be answered easily with SS and without access to the document data. Result of this execution will be a list of nodes in SS for each single-branch query. Path of these nodes will be absolute (from root to node in SS).
- ***d. Example:*** suppose we want to execute *Q1* query on SS of figure 4. at first, query split into two single-branch queries: *A//B* and *A//C//D*. we only need to keep and access to leaves of query for each branch because the document labeled with Dewey numbers and lower nodes have some information about upper nodes (path traversed from root) in themselves. Therefore, for *A//B* branch, *B1* and *B2* nodes and for *A//C//D* branch, *D1* and *D3* nodes are answers of single-branch queries. It is obvious that *D2* is not in the results because it has not condition of single-branch queries. These nodes, have absolute path *W/A1/A2/B1* ،*W/A1/B2* for *A//B* branch and *W/A1/B2/C1/D3, W/A1/A2/C2/D1* for *A//C//D* branch, in turn.

B. *Generation Of Ms:*

- ***a. Primary Definition:*** MS is a table with three columns. First two columns are leaf nodes of two query branches in SS and its third column is level of JP between two these branches. Leaf nodes in MS have absolute path. Therefore, each record of this table shows an operation called *Matching Process*.
- ***b. Definition:*** Matching Process is process of comparing two or more nodes in the document to achieve part of answer.
- ***c. The procedure:*** after splitting query into several single-branch queries and gaining corresponding nodes to leaves of single-branch queries in SS, now we have to achieve JP of these nodes. In Dewey encoding manner, each leaf indicate a branch. Result of single-branch queries execution on SS is a list of nodes for each single-branch query. The Path of these nodes are absolute (i.e, path of each node is completely specified from root to node). Now, to achieve JP of these nodes, we select a node from each list and compare their absolute paths with each other. If paths of selected nodes were same from root to level of query JP, we add those two nodes and level of JP to MS.
- ***d. Example:*** execution result of single-branch queries of *Q1* on SS in figure 4 are *B1* with absolute path *W/A1/A2/B1* and *B2* with absolute path *W/A1/B2* for *A//B* branch and *D1* with absolute path *W/A1/A2/C2/D1* and *D3* with absolute path *W/A1/B2/C1/D3* for *A//C//D* branch. Now, we compare elements of each branch with each other. If traversed paths from root to JP of query, - here *A* - are the same between each two comparing nodes, we add one record which contains those two nodes and JP level of two nodes. For example *B1* and *D3* have same path down to JP *A1* (Level of root is considered as *0*). Therefore, we add <*B1, D3, 1*> to MS. as a result MS has following records: <*B1, D1, 1*>, <*B1, D1, 2*>, <*B1, D3, 1*>, <*B2, D1, 1*> *and* <*B2, D3, 1*>.

Primary MS production algorithm for a two-branch query is shown in figure 5. Let us consider lines from 7 to 10 of algorithm. In this section, for each similar prefix from root to one JP located between two nodes, algorithm adds one record with JP level to MS.

This algorithm shows all records that must be added to MS for each JP; because one node can has more than one JP (line 7).

- ***e. Example*:** As is obvious in SS, *B1* has two JPs (*A1*, *A2*) and *D3* has one JP (*A1*); but these two nodes have same path just up to JP *A1*.

```
Algorithm MS_proc :

Input: Q as QTP
Output: MS as Result_Table
 1: Let A and B the two leaves of Q
 2: Let jp  = Joint point between A and B
 3: Let AL = list of SS nodes match A branch
 4: Let BL = list of SS nodes match B branch
 5: for each an ∈ AL do
 6:     for each bn ∈ BL do
 7:       for each jp1 in an , jp2 in bn do
 8:           if an.Prefix(jp1) = bn.prefix(jp2) then
 9:               MS.addREC(an, bn, jp1.level())
10:           end if
11:     end for
12:   end for
13: end for
```

Figure: 5 The Pseudocode of the R_T Production

```
Algorithm_1 : FinalResult-Proc

Input: ET as record of ResultTable
OutPut Outputlist as array of matched nodes
 1: Let L1= ET.field1.extend
 2: Let L2= ET.field2.extend
 3: Let node1= first node of L1
 4: Let node2= first node of L2
 5: Let L= ET.field3 // level of JP
 6: while –(one of L1 or L2 reach the end) do
 7:     for each a in L1, b in L2 do
 8:       if a.prefix(L) = b.prefix(L) then
 9:           (a, b) add to output
10:       else if node1.prefix(L)> node2.prefix(L) then
11:                   node2= L2.Jump(L)
12:       else
13:                   node1= L1.Jump(L)
14:       end if
15:     end for
16: end while
```

Figure 6: Final Result Production

FINAL RESULT

Final result is constructed based on the ResultTable. Each record in the ResultTable guides query processor to produce a part of the final result. Therefore, final result is the union of partial results produced for each record of ResultTable.

a. ***The procedure:*** Consider a given record in a ResultTable and its fields. Two first fields are two nodes in a Structure Summary. As mentioned in introduction, each node in Structure Summary has an ordered list of related nodes' Dewey number in the WEB document that called Extend. Elements of these two lists should be compared with each other to produce part of the final result. This process is called Matching Process. The matching process starts with comparing current node labels of lists (first ones at the beginning). If comparing nodes have same prefix up to JP level (third field), those are part of result.

b. ***Example:*** Consider record *B1*, *D1* of the previous ResultTable (the JP value of the record is assumed *2*). Suppose their related node labels form the below lists:

Level of *W* is assumed *ε*.

B1-extend = {**1/3**/6, **1/7**/1}

D1-extend = {**1/2**/2/1, **1/2**/2/2, **1/3**/3/1, **1/3**/5/7, **1/6**/2/2, **1/7**/1/2}

Applying the algorithm of matching process on these lists forms the below output list:

OutputList = {(1/3/1, 1/3/3/1), (1/3/1, 1/3/5/7), (1/3/6, 1/3/3/1), (1/3/6, 1/3/5/7), (1/7/1, 1/7/1/2)}

Lines number 8 and 9 of figure 6, give us nodes that have same prefix up to JP level and are part of Matching Process results. For nodes such as *1/2/2/1* which have not successful matching process, we should jump to next first node that is just greater in this level (look at *jump(L)*). For example in level *2*, if node *1/2/2/1* is current node, then next node will be *1/3/3/1*.

III. MS AND COMPLICATED QUERIES

In previous sections, overall procedure of algorithm to answer a two-branch query is shown; but there are queries that are more complicated in database' world. In this section, we show MS flexibility and applicability in these queries so that we can answer these queries with processing of leaf nodes just once.

A. Jps With More Than Two Branches:

As mentioned in primary definition, MS is a table with three columns that first two columns are nodes of each branch and its third column is common level between two branches; but in the world, it is possible that several branches were joined together in one JP. For example, assume *Q2: //A[./C][./D]/B;*

```
Algorithm_2 : FinalResult-Proc

LET A1, A2..., An = field1, field2,..., fieldn
LET L=ET.field(n+1)
For each a1∈ A1, a2∈A2,....,an∈An do
    IF a1.prefix(L)= a2.prefix(L)=.....= an.prefix(L) then
      Add( a1,a2,...,an) to output
    Else
      Min(a1,a2,...an).next()
    Endif
Endfor
```

Figure: 8 Pseudo code of Matching Process

```
Algorithm_3 : FinalResult-Proc

1: Assume Order process in MS_Model is:
2:      MS1→MS2→...→MSn
3: Match_Proc(MS1, 1)
4: Match_Proc(MS,i)
5:    If(i=n+1) then
6:      Successful match
7:    Elseif (Match_Proc in MSi is Successful) ther
8:      Match_proc(MSi+1,i+1)
9:    Else
10:     Min(MSi, MSi+1).jump
11:     Match_Proc(MSi+1, i-1)
```

Figure: 10 Pseudo code of Matching Process

Here it is enough that we change primary definition of MS as follows:

Secondary Definition of MS: MS is a table with *M+1*columns for a JP with *M* sub-branch so that its 1^{st} to M^{th} columns are leaves of branches and last column is common level of JP between all nodes.

We also need to change pseudo code of figure 6 as figure 7 to generate final result for each MS record.

B. Queries With Several Jps:

In a query, each MS will be used for one JP. Therefore, for queries with *M* JPs we need *M* MSs; but these MSs cannot be used independently and there is relationship between them. Therefore, we need two changes: first, we use *MS_model* instead of MS in figure 3.

a. ***Definition:*** MS_model shows a set of n MS for a query with n JP along with their relations.

b. Example: suppose that we want to build an MS_model for query of figure 8. This QTP has three branches *(A//B/C, A//B/D, A/E//F)*. The first joint point is *B* which joins two first branches *A//B/C* and *A//B/D*. *A* is another JP between two first and third branch. As shown in figure 9, *A* uses output of *B*. Therefore, output of *MSB* will be used as a field of *MSA*.

Second change must be in sequence of nodes processing to generation of Final Result. This change illustrated in figure 10. Pseudo code of figure 10 show a bottom-up processing. This means that at first it process those JPs that are in lower position in QTP tree. The procedure is as follows: when there are orders of process between several JP, it begins with first JP (line No.3). A recursive procedure called *Math_Proc* is used that consider orders of process. If matching process was successful for a MS (line No. 7), this procedure tests next MS (line No. 8). This process continues while matching process is successful for all MSs, (line No. 5, 6). If matching process was not successful for one MS, we must do jump from either that or previous MS (line No. 10) and matching process begin from previous MS (line No. 11).

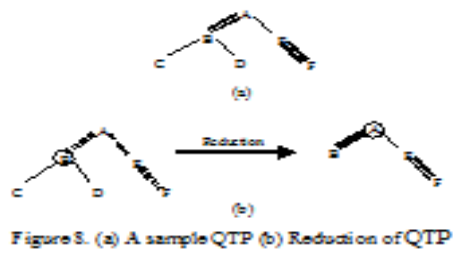

Figure 8. (a) A sample QTP (b) Reduction of QTP

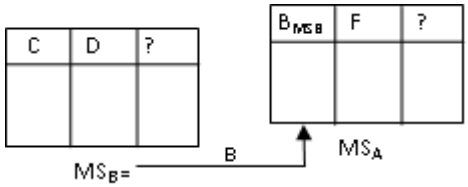

Figure 9. MS_Model of Figure 9

*C. Queries with *, ?, // and /:*

MS method is similar to both Path Index and Containment join methods. For single-branch queries, path indexes undertake the responsibility of query conversion to absolute path. Fortunately, some of them such as *YAPI*[19] have acceptable performance on various operators (*, ?, // and /) in single-branch queries and don't need to access to real data of the document and just with access to SS can answer to various kind of single-branch queries.

IV. EXPERIMENTAL RESULTS

In this section we present the result of our experiments. As discussed above, we categorize the existing WEB query processing method into three groups. We compared our MS methods with *Twig²Stack* and *TJFast*. *Twig²Stack* is selected as the representative of holistic twig join algorithms of Group *B* and *TJFast* as the representative of Group *C*, the methods which only access leaf nodes of QTP in the WEB document. As mentioned above our MS methods are classified into the Group *C* too.

a. Our path index: In second step our method needs to one of path indexes to convert single-branch queries to absolute path of result nodes in SS. There are many path index methods to choose; but each method tries to answer to complicated queries by itself. Therefore, for many of queries they need to access to real data of the document and thus they have not enough performance whereas in our method a path index is used just on SS and to answer to single-branch queries. Therefore, it must have only two below properties:

i. Its SS is small and it answers to single-branch queries quickly.
ii. It is applicable for all single-branch queries with all possible operators (*, ?, //)

Among all path index methods, the best option that provides two above properties is *YAPI* [19]. It is quickest and cheapest method to answer to single-branch queries.

b. Data sets: We use four datasets *TreeBank [15], XMark [17]* and *DBLP [11]* and a Random dataset in our experiments. *DBLP* is a famous dataset which is a shallow and wide document. Against DBLP, we use well-known TreeBank dataset which is a deep document.

c. Random dataset: We build random dataset with the depth of *12* and width of node – maximum number of children of a node – *10*. The elements tags of this dataset are only *A, B, C, D, E* and *F*. In this way, one element could have one or some homonymous nodes as children. As a result, the Structural Summery of the document could be complex and nested. Here, the numbers, types and orders of children of nodes are chosen accidentally.

***d. Original Dewey*:** In our experiments, the extended Dewey labels are not stored by the dotted-decimal strings displayed (e.g.*\1.2.3.4"*), but rather a compressed binary representation. In particular, we used *UTF-8* encoding as an efficient way to present the integer value, which was proposed by Tatarinov et al. [8].

***e. Queries*:** In order to compare our MS method with TJFast, we use queries that are listed in the Table 1. Each query has its distinguished property. The query *XQ1* is a single query with *P-C* relationships. For this kind of queries we do not need to generate MS. The queries *XQ4* and *XQ5* are multi-branch queries with A-D relationships. The query *XQ3* is also a multi-branch query but with P-C relationships and *XQ2* is combination of A-D and P-C relationships.

We choose three parameters to compare our MS method with TJFast: i) number of elements read, ii) Size of disk files scanned and iii) execution time

***f. Number of elements read*:** In both methods, just leaves of QTP will be processed; but there are two fundamental differences: 1) in TJFast at first, each node will be checked whether it has single-branch condition or not; but in our method, we only access those nodes, which are member of one query branch. 2) TJFast try to answer the query by direct comparing of each branch leaves in document and it compares many leaves that have not any structural relation with each other; but in our method with considering MS, only those leaves will be compared that have structural relation with each other and many nodes don't need to be accessed because they have no counterpart in other branch.

This difference is more obvious in parent and child queries.

g. ***Size of disk files scanned:*** In TJFast method when we do comparisons, we need to save some nodes because it is possible that they can produce part of answer in comparison with another node in the future. This is because TJFast try to answer the query by direct comparing of nodes blindly; but in our method, we do not need to save any intermediate data because the way of node processing and answering the query are specified in MS.

h. ***Execution time:*** the execution time of TJFast seems to be more than MS. TJFast needs to decode the labels to their paths and then compare them but in our method, there is no need to decode node labels. Figure 12 confirms the discussion. Our experiments run on a PC with 2.2 GHz Intel Pentium IV processor running Red Hat Linux 8.0 with 2 GB of main memory.

Table 1. Queries used to compare MS with TJFast

Query Name	Query	Data Base
XQ1	/site/people/person/gender	XMARK
XQ2	/S[.//VP/IN]//NP	TreeBank
XQ3	/S/VP/PP[IN]/NP/VBN	TreeBank
XQ4	//article[.//sup]//title//sub	DBLP
XQ5	//inproceedings//title[.//i]//sup	DBLP

Table 2. Queries used to compare MS with Twig^{2}Stack

Query Name	Query	Data Base
XQ5	//dblp/artcle[author]/[.//title]//year	DBLP
XQ6	//people//person[.//address/zipcode]/profile/education	XMark
XQ7	//S//VP/PP[IN]/NP/VBN	TreeBank

i. ***Twig^{2}Stack*:** In this section, we compare our method with Twig^{2}stack method as representative of *B* group methods. We compare our method with Twig^{2}stack in two criteria of *i) number of elements read* and *ii) execution time*. Queries are in table 2, Twig^{2}stack like all of methods in its group will access to all QTP nodes to answer the query. Therefore, it will have more node access than TJFast method to answer the query; but it does not need to convert Dewey numbers to path elements' name, as a result, in some cases it operates better than TJFast in execution time factor. Figure 13 confirms the discussion.

j. ***Random Dataset:*** Here we execute our queries on Random Dataset that is described before. This dataset has many namesake elements and a non-uniform structure. Therefore, it shows efficiency of methods clear.

k. ***Single-branch queries:*** Both *MS* and *Twig^{2}stack*, execute *8* single-branch queries *A1, A2, …, A8* with *2, 3 , …, 9* length respectively. All queries are Partial, i.e, they begin with //, As shown in figure 14, as many as number of single-branch queries' nodes increase, number of elements to be accessed in the document in MS decrease.

l. ***Multi-branch queries*:** Both *MS* and *TJFast*, execute *A1*, *A2*, *A3* and *A4* queries which have *2*, *3*, *4*, *5* branches respectively. As shown in figure 14 in both methods when number of branches increases, number of node accesses will increase whereas growth rate of MS is very less than growth rate of TJFast.

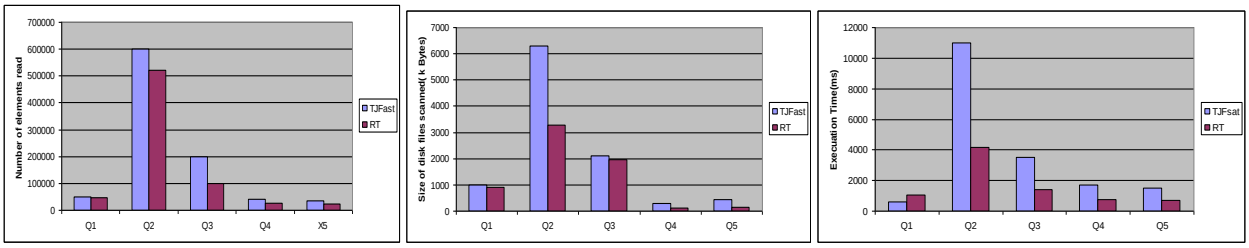

Figure 12. MS in Comparison with TJFsat

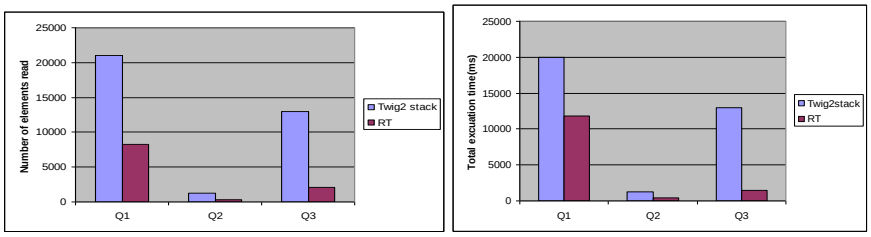

Figure 13. MS in Comparison with T2S

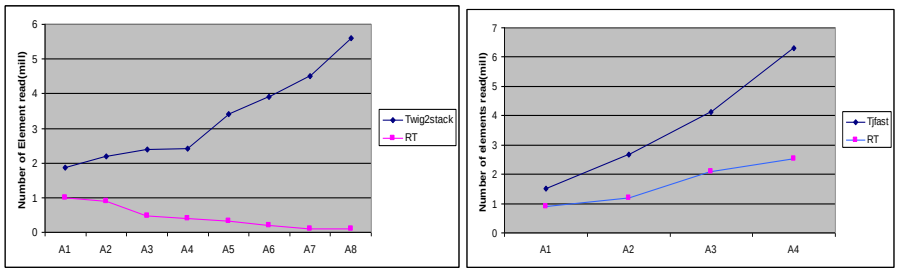

Figure 14. MS and Random DataSet

V. CONCLUSION

In this paper we have presented to the viewer. Using this graph, nodes can be compared to the blind did. We are trying to answer the question for us to process all the nodes that are produced or processed in other words the number of nodes equal to the number of nodes to be answered. But we are still far from perfect equality. We try to improve the papers and later works with the following methods to achieve this equality:

- Provide a new method for answering queries branch, a branch of our process.
- Provide a new method for the query execution time Btvanymnmayshgr that build the structure using the display screen to be able to reach the structure.

Provide complete optimization techniques on the screen.

VI. REFERENCES

[1]. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu., Y. Structural Joins: A Primitive for Efficient WEB Query Pattern Matching. In Proc. ICDE: 141-152(2002)

[2]. Bruno, N., Koudas, N, Srivastava, D. Holistic Twig Joins: Optimal WEB Pattern Matching, In Proc. SIGMOD Conference: 310–321(2002)

[3]. Chung, C., Min, J., Shim, K. Apex: An adaptive path index for xml data. In Proc ACM Conference on Management of Data SIGMOD: 121 - 132(2005)

[4]. Dewey, M. Dewey Decimal Classification System. http://www.mtsu.edu/~vvesper/dewey.html

[5]. Fontoura, M., Josifovski, V., Shekita, E., Yang, B. Optimizing Cursor Movement in Holistic Twig Joins, In Proc. CIKM Conference: 784 – 791(2005)

[6]. Garofalakis, M. N., Gionis, A., Rastogi, R., Seshadri, S., Shim. K. XTRACT: A system for extracting document type descriptors from WEB documents. In Proc. ACM SIGMOD Conference: 165 - 176 (2000)

[7]. Goldman, R., Widom, J. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Proc. 23rd VLDB Conference: 436—445(1997)

[8]. Haerder, T., Haustein, M, Mathis, C., Wagner, W. Node labeling schemes for dynamic WEB documents reconsidered, In Proc Data & Knowl. Engineering, Elsevier (2006)

[9]. Kaushik, R., Bohannon, P., Naughton, J., and Korth, H. Covering Indexes for Branching Path Queries, In Proc. 11rd SIGMOD Conference, 2005, 133–144

[10]. Kaushik, R., Krishnamurthy, R., Naughton, J., and Ramakrishnan, R. On the integration of structure indexes and inverted lists, In Proc SIGMOD Conference, 2002, 779-790

[11]. Ley., C. DBLP Computer Science Biblography, http://www.informatik.unitrier.de/ley/db/index.html

[12]. Lu. J., Ling, T. W., Chan, C. Y., and Chen, T. From region encoding to extended dewey: On efficient processing of WEB twig pattern matching. In Proc VLDB Conference, 2005, 193–204

[13]. Milo, T., and Suciu, D. Index Structures for Path Expressions. In Proc. ICDT, 1999, 277-295

[14]. Rizzolo. F. and Mendelzon. A., Indexing WEB Data with ToXin, in Proc.5th. WebDB conference (2001)

[15]. Schmidt, A. R., et al. The WEB Benchmark Project. Technical Report, INS-R0103, CWI, 2003.

[16]. Tatarinov, I. , Viglas, S. , Beyer, K. S., Shanmugasundaram, J. , Shekita, E. J. and Zhang, C. Storing and querying ordered WEB using a relational database system. In Proc. of SIGMOD, 2002, 204-215.

[17]. U. of Washington WEB Repository. http://www.cs.washington.edu/research/xmldatasets/.

[18]. Yu, T., Wang, W., and Lu, J. TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-predicates on WEB Data. In Proc. ICDE: 141-152(2005)

[19]. Zamato, G., Debole, D., Zezula, P and Faust YAPI: Yet Another Path Index for WEB searching In Proc. Springer: 141-152(2003)