Reducing graph matching to tree matching for XML queries with ID references

Huayu Wu[1], Tok Wang Ling[1], Gillian Dobbie[2], Zhifeng Bao[1], and Liang Xu[1]

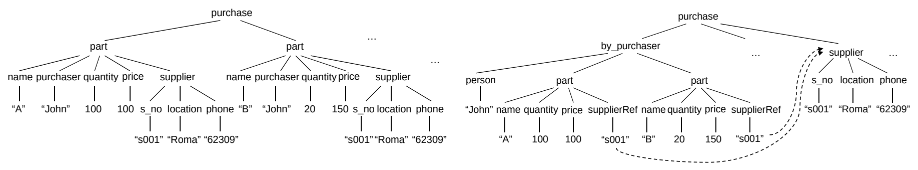
[1] School of Computing, National University of Singapore
{wuhuayu, lingtw, baozhife, xuliang}@comp.nus.edu.sg
[2] Department of Computer Science, The University of Auckland, New Zealand
gill@cs.auckland.ac.nz

Abstract. ID/IDREF is an important and widely used feature in XML documents for eliminating data redundancy. Most existing algorithms consider an XML document with ID references as a graph and perform graph matching for queries involving ID references. Graph matching naturally brings higher complexity compared with original tree matching algorithms that process XML queries. In this paper, we make use of semantics of ID/IDREF to reduce graph matching to tree matching to process queries involving ID references. Using our approach, an XML document with ID/IDREF is not treated as a graph, and a general query with ID references will be decomposed and processed using tree pattern matching techniques, which are more efficient than graph matching. Furthermore, our approach is able to handle complex ID references, such as cyclic references and sequential references, which cannot be handled efficiently by existing approaches. The experimental results show that our approach is 20-50% faster than MonetDB, an XQuery engine, and at least 100 times faster than TwigStackD, an existing graph matching algorithm.

1 Introduction

Because XML is an important standard format for data exchange over the Internet, it is important to remove redundant data from documents, which uses unnecessary storage and adds extra cost during data transfer. Consider the example shown in Fig. 1(a). Since both part A and part B are supplied by the same supplier, the information about supplier $s001$ is repeated twice. The most common way to reduce redundancy is to introduce ID and IDREF attributes [20]. ID and IDREF can be likened to primary key and foreign key constraints in relational databases. Using ID/IDREF, each object is stored once under the document root with a unique ID. A new structure for the document tree in Fig. 1(a) with data redundancies removed is shown in Fig. 1(b). The dotted arrows represent the references from IDREF value to the referenced object which will have the same value as its ID.

Despite the importance of ID/IDREF for good XML design, existing algorithms that process queries involving ID references in XML are still not efficient. To the best of our knowledge, all the existing algorithms consider both XML documents with ID/IDREF and XML queries with ID references as digraphs,



(a) XML tree with data redundancies (b) XML tree after reducing redundancies

Fig. 1. Example XML document in two schemas, with and without data redundancies

and perform graph matching to process queries. It is true that an XML document is modeled as a digraph if we consider the ID references as directed edges. However, transforming tree pattern matching to graph pattern matching naturally brings much higher complexity, because graph matching is more costly than tree matching with the same size input [12]. A simple question is whether we have to abandon many efficient tree pattern matching approaches (for XML and queries without ID references), and invent new, but less efficient graph pattern matching algorithms to process such queries with ID references. Fortunately, the answer is no. Unlike the graph model for social networks or other graph databases, ID reference in an XML document is not a random link between nodes. It has strong semantics, which always starts at an IDREF value and references an object with a same ID value. Surprisingly, no existing algorithm captures this semantics during query processing. They normally focus on how to enhance the efficiency of graph matching, but ignore the fact that using the semantic information of ID/IDREF, graph matching can be reduced to a less complex tree matching.

This paper focuses on incorporating semantics of ID/IDREF to reduce graph matching to tree matching to process XML queries with ID references. Besides significantly reducing the pattern matching complexity, our approach also makes all existing efficient tree pattern matching algorithms feasible for queries with ID references. The rest of the paper is organized as follows. We revisit some related work in XML query processing in Section 2. Our semantic approach to processing queries with ID references is presented in Section 3. Section 4 discusses how to handle special references in documents and queries. We present experimental results in Section 5 and conclude our paper in Section 6.

2 Related work

XML query processing has been studied for many years. Since in most XML query languages (e.g. [4][5]) queries are expressed as *twig* patterns, finding all occurrences of a twig pattern in an XML document is a core operation for XML query processing. In the early stage, a lot of work focused on storing and querying XML data using mature relational database systems [10][21][27][26]. Generally they shred XML data into relational tables, and convert XML queries into SQL to query the database. The advantage of these relational approaches is that they can manage and operate on values efficiently, e.g. performing range search for predicates, and they can make use of existing relational query optimizers to optimize SQL-style XML queries. However, the drawback of the relational approaches is also obvious. A twig pattern XML query may involve many table joins, which are costly. Sometimes it is not easy to decide what tables are to be joined and how many times to join them particularly for queries with "//"-axis (ancestor-descendant axis). Later how to process twig pattern queries natively without using relational databases became a hot topic. The structural join based approach is the most efficient native approach accepted by researchers. In particular, *TwigStack* [6] and subsequent work [9][15][19][8] bring in the idea of holistic structural join, which makes structural join very efficient. Finally, [25] complements structural join based approaches by introducing relational tables to process content search and content extraction.

When XML documents and queries involve ID references, most twig pattern matching based algorithms cannot handle the references between nodes. Many works focus on how to efficiently perform graph matching for such documents and queries that are modeled as graphs. Some traditional approaches [22] generate all possible mappings between each pair of nodes in two graphs and check for correctness. However, this sort of graph matching problem is NP-complete generally [12]. Moreover, these graph matching algorithms can hardly support "//"-axis queries. Later, [23] and [7] consider the structure of XML documents with ID/IDREF as a directed acyclic graph (DAG) and proposes algorithms to process queries on DAGs. However, XML documents with ID/IDREF may be a cyclic graph [13]. Recently, [17] and [16] extend twig pattern query to support queries involving ID references, and propose techniques to solve the extended twig. [24] proposes a new labeling scheme for document graph so that parent-child and ancestor-descendent relationships can be identified and thus queries can be processed. However, all these attempts consider ID references as a random link between nodes and match random graph queries to the document graph. As mentioned in Section 1, graph matching is normally more expensive than tree matching, thus this paper proposes a method to reduce graph matching to much less complex tree matching for XML queries.

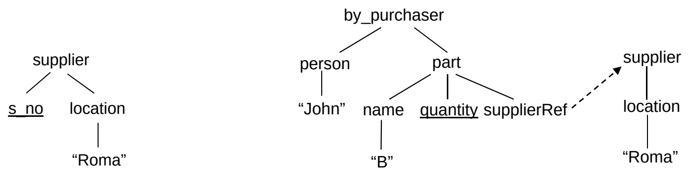
3 Semantic approach for queries with ID references

3.1 Reference pattern query

Twig pattern, or tree pattern, is considered the core XML query pattern when ID reference is ignored in documents and queries. We first extend the twig pattern expression to express ID references in a query, and propose a semantic approach based on this approach. Our extension mainly includes two parts: explicitly marking output nodes and introducing ID reference edges.

Definition 1. *(**Output node**) Output nodes in a twig pattern query are defined as a group of nodes in the twig pattern such that the query aims to find values for them, based on conditions on other nodes.*

Every query must contain at least one output nodes. For example, in the query shown in Fig. 2(a), *s_no* is an output node since we aim to find the value for c for each supplier that is located in 'Roma'. Besides noting output node, we also note the ID reference within a query. For example a query to find the value for *quantity* of part B which is purchased by John and supplied by some supplier located in 'Roma' is issued on the document shown in Fig. 1(b). From the document, we can see that full information about a certain *supplier* is stored separately from the object *part* and that an IDREF property *supplierRef* is used to reference the corresponding *supplier*. By considering both output node and ID reference, we propose the notion of *reference pattern query*. The above query can be issued as a reference pattern query as shown in Fig. 2(b).

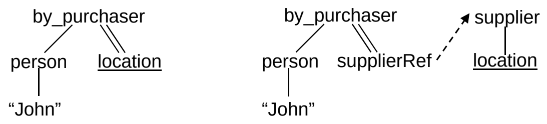


(a) Query without ID reference (b) Example reference pattern query

Fig. 2. Example twig pattern query and reference pattern query

Definition 2. ***(Reference pattern query)*** Reference pattern query *generalizes twig pattern query to express ID references between twigs using a dotted referencing arrow. In a reference pattern query, the main body where the referencing arrow starts is called the* referencing part*, and the part to which the referencing arrow points is called the* referenced part*. The referenced part normally corresponds to an object with an ID value. The output nodes in a reference pattern query are marked by underlining them.*

Note that a query issuer is expected to have some schematic information of the underlying XML document. Otherwise, she has no way to compose a structured query expression. Similarly, a query processor is also aware of the document structure. Although sometimes there is no formal schema available for a document, by parsing the document a program can easily summarize its structure. With such requirements, we assume that a user should be able to issue a reference pattern query, and the system can interpret the query, even if the query contains implicit ID references across a "//" relationship. For example, to process the query shown in Fig. 3(a) which finds all locations where John purchases things, the system will identify the ID reference across the "//" edge by consulting the document structural summary, and rewrite the query to be a reference pattern query as shown in Fig. 3(b). Sometimes, an object class may have a recursive reference, e.g. a *paper* cites another *paper*. This case is similar to element recursion in DTDs. The work in [18][11] discussed how to translate queries involving recursive elements into SQL. Since in our work, we use SQL to handle ID reference, these works can be adopted to solve recursive ID references.



(a) Original query (b) Reference pattern query

Fig. 3. Case that "//" relationship contains ID reference

3.2 Parsing XML document with ID references

The hierarchical structure of XML data is normally modeled as a tree. However, when there are references from tree node to tree node, the data model is considered as a graph. Most existing algorithms handle XML queries over documents with ID reference in a graph matching manner, e.g. using an extra index to record ID references between nodes or inventinga new labeling scheme for graphs. As mentioned in Section 1, compared with tree matching, graph matching naturally brings higher complexity. ID reference in an XML document is not a random link between nodes. It always starts from an IDREF attribute and points to an object with the same ID value. If we use the semantics to avoid treating a document as a random graph, the performance could be improved. First, we present how our approach parses an XML document with ID references.

Most XML query processing algorithms assign a label to each document node, so that the parent-child or ancestor-descendant relationship between each pair of document nodes can be easily determined by their labels during query processing. In our approach, we ignore the ID references, and only label property nodes, object nodes and other internal nodes, but not value nodes. IDREF is an internal node, but its value is a value node (as shown in Fig. 4).

Definition 3. ***(Object, property)*** *In an XML document, the parent node (either an attribute or an element) of each value is a* property*. We consider the parent node of a property node (including IDREF node) as an object*[3].

In the document in Fig. 1(b), the nodes *name*, *quantity* and *supplierRef* are all properties as they have value children, and each *supplier* and *part* are objects as they are parents of certain properties. When we ignore the ID reference and the values, the label assignment of the document in Fig. 1(b) is shown in Fig. 4.

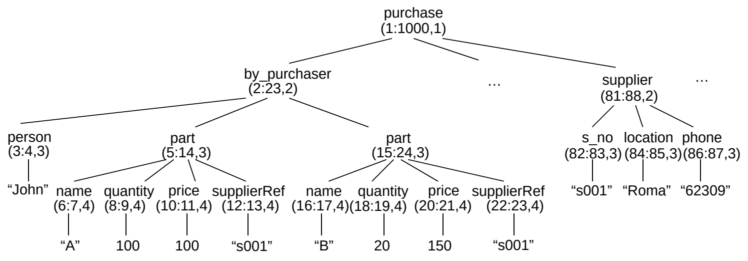


Fig. 4. The purchase document with internal nodes labeled

The labeled document nodes are organized as inverted lists based on different tags, which is the same as other approaches; whereas, the values that are not labeled are stored in object-oriented relational tables. In particular, for each object class there is a table, whose schema includes a label field to store the label of each object in this class, and a set of property fields to store the values of each property for a certain object. During this process, both ID value and IDREF value are treated in the same way as other properties. The reference between them is ignored, thus these indexes only keep the tree like nature of the

[3] It may not be semantically true for all cases, but it will not affect the correctness of query processing.

document. The example object tables for *supplier* and *part* are shown in Fig. 5, in which the ID and IDREF attributes are stored in the same way as other properties. Note that the IDREF attribute *supplierRef* in the XML document in Fig. 4 is represented directly with the same value in the attribute *supplierRef* in the *part* table. There is no redundancy or duplicated data in the tables.

$R_{supplier}$

Label	S_no	Location	Phone
(81:88,2)	s001	Roma	62309
...	...	...	...

(a) *Supplier* table

R_{part}

Label	Name	Quantity	Price	SupplierRef
(3:12,3)	A	100	100	s001
(13:22,3)	B	20	150	s001
...	...	...	...	...

(b) *Part* table

Fig. 5. Example object tables during document parsing

The implementation details of object table construction and the solution to some potential problems of object tables, e.g., how to store multi-valued property, are discussed in our previous report [25].

3.3 Query processing with tree matching

The ID reference in a reference pattern query is reflected by a dotted arrow. Such a dotted arrow always corresponds to an ID reference in the document, which makes the document a graph structure. During pattern matching, most algorithms consider ID references in both documents and queries as a normal edge, thus they have to perform graph matching. In our approach, we try to reduce the graph matching to a simple tree matching, to improve pattern matching performance. To do this, we treat a document as a tree, as mentioned in the previous section, and also ignore the dotted arrow in a reference pattern query when we perform pattern matching. However, the ID reference is a part of the query constraint which cannot be ignored. Our solution transforms the ID reference in a query to a table join, because (1) the semantics of ID/IDREF are such that all the ID references must be between ID and IDREF attributes, and they are not a random link, and (2) both the corresponding ID value and IDREF value are stored in relevant object tables and the reference between them is the same as the equi-join of the two tables.

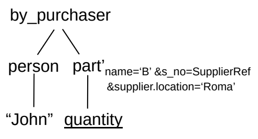
The general idea of our approach is to decompose a reference pattern query into a referencing part and a referenced part (the two parts are defined in Definition 2). The query is processed by a tree matching of the referencing part, with any existing twig pattern matching algorithms, and a join between the referencing part and the referenced part. In more detail, the join between the referencing part and the referenced part is eventually performed by a table join between two object tables with ID and IDREF attributes. In fact, a tree matching is a series of structural joins between each adjacent query node. In our heuristic we try to perform the join between the referencing part and the referenced part first, as this operation normally results in high selectivity due to the constraints in the referenced part. However, if the referencing part has no output node, the whole referencing part becomes a predicate, then we match the referencing part first before joining with the referenced part where output nodes are involved. The detailed query processing algorithm is presented in Algorithm 1, in which we suppose *o* and *obj* are two objects in the referencing part and referenced part of a reference pattern query respectively. First, we only consider the basic reference pattern query with only one referencing part and one referenced part.

Algorithm 1 Query processor

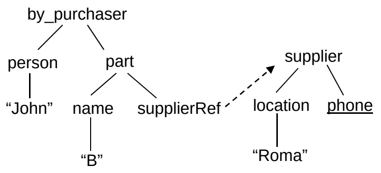
```
1: if there is no output node in the referencing part then
2:     match the referencing part to the document tree, to find the set $S$ of distinct values of the IDREF attribute
3:     join $S$ with $R_{obj}$ for the referenced part, based on the condition $S.IDREF=R_{obj}.ID$
4:     select values for the output node in the joined result, based on other constraints under $obj$.
5: else
6:     Join $R_{obj}$ and $R_o$ based on the condition $R_{obj}.ID=R_o.IDREF$, and select the labels of $o$ based on the predicates under $obj$ and $o$
7:     create a new inverted list $T_{o'}$ for $o$ and put the selected labels into $T_{o'}$
8:     rewrite the referencing part by changing $o$ to $o'$, which corresponds to $T_{o'}$
9:     match the rewritten referencing part to the document tree with $T_{o'}$ for $o'$ to find values of the output nodes in the referencing part
10:    if there are output nodes in the referenced part then
11:        find the set $S$ of distinct values of the IDREF attribute from the tree matching result
12:        join $S$ with $R_{obj}$ for the referenced part, based on the condition $S.IDREF=R_{obj}.ID$
13:        select values for the output node in the joined result, based on constraints under $obj$.
14:    end if
15: end if
```

There are three cases of reference pattern query, with respect to the position of output nodes: Case (1) output nodes reside in the referencing part, Case (2) output nodes reside in the referenced part and Case (3) output nodes reside in both the referencing part and referenced part. Now we use examples to illustrate our approach for the three cases.

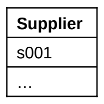
Example 1. Consider the Case (1) query shown in Fig. 2(b). The query asks for the quantity of part B which is purchased by John and supplied by some supplier in 'Roma'. In this query, only the referencing part contains an output node, *quantity*. The two objects involved in the ID reference are *part* and *supplier*. Using the algorithm, we join R_{part} and $R_{supplier}$ (as shown in Fig. 5) based on s_no=SupplierRef, filter the results by the predicate part.name='B' and supplier.location='Roma', and select the labels for *part*. These labels are put into a new inverted list for *part* and the referencing part is rewritten as shown in Fig. 6(a). In the new query, the subscript of node *part′* explains that this node corresponds to the new inverted list of parts, whose name is 'B' and has a supplier located in 'Roma'. The final step is to match the tree pattern referencing part with the new inverted list for *part′* to the document tree.



(a) Rewritten query for Example 1



(b) Query for Example 2



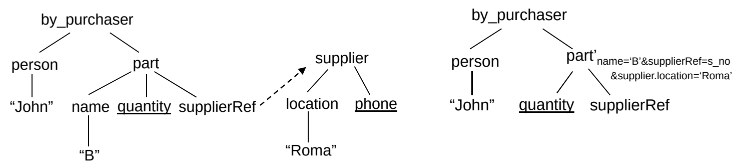
Supplier
s001
...

(c) IDREF values for Example 2

Fig. 6. Figures for Example 1 and 2

Example 2. Consider the Case (2) query shown in Fig. 6(b), in which only the referenced part has an output node, *phone*. By the algorithm, we first match the referencing part to the document tree, to find the labels of each matched *part* object. Then using these labels we can find the distinct values of *supplierRef* in R_{part}. The result is shown in Fig. 6(c). We join these tuples with table $R_{supplier}$ and select *phone* in $R_{supplier}$ based on *location*='Roma'.

Example 3. Consider the Case (3) query in Fig. 7(a). In this query both the referencing part and the object contain output nodes. We first join tables R_{part} and $R_{supplier}$ based on the equality of *supplierRef* and *s_no*, and select *part* labels based on the conditions that part's name is 'B' and supplier is in 'Roma'. Then a new inverted list $T_{part'}$ for *part* is constructed with the selected labels. The referencing part is rewritten by renaming the node *part* to be *part′* so that the new inverted list will take effect (shown in Fig. 7(b)). Using any tree matching algorithm to process the rewritten referencing part, we can find the labels for matched *part*. Then in R_{part} we can extract values for *supplierRef* and the output node *quantity*. To find the value for the other output node *phone*, we join the distinct values of *supplierRef* with the table $R_{supplier}$, and select the *phone* value based on the condition that *location* equals 'Roma'.



(a) Example query (b) Rewritten query

Fig. 7. Figures for Example 3

During query processing, any reference pattern query requiring graph matching on the document is eventually processed by tree pattern matching and table joins. Furthermore, the tree pattern to be matched is normally much simpler than the original query pattern with references. Since most relational systems can perform selection and join very efficiently with B^+ tree indexes, the overhead on table operations will not affect the benefit of reducing graph matching to tree matching. Our experiments also prove this. In particular, when the referenced part of a reference pattern query is in a complex pattern, e.g. enclosing other objects, we can perform pattern matching on the referenced part, before joining with the referencing part.

3.4 Correctness

The basic query processing idea in our approach is to replace the structural join using ID references, which is used in other algorithms, with a table join. Actually, an ID reference means the involved IDREF attribute has the same value as the ID attribute. On the one hand, we can visualize such a reference using a graph edge and perform a structural join; on the other hand, we can push the equality of IDREF and ID values to a table join. In this regard, both structural join and table join have the same effect of solving the constraints of ID references.

4 Special references

ID/IDREF in XML documents may lead to very complex patterns. In this section, we introduce two special cases of ID/IDREF and explain how our algorithm handles queries involving these cases.

4.1 Cyclic reference

ID references in an XML document may cause cycles if we consider parent-child relationships and ID references as directed edges in a document graph. Consider a document which contains such cycles, as shown in Fig. 8. In this document, Roy chooses Lisa as his first partner, while Lisa chooses Roy as her second partner. Then the references between *member* Roy and Lisa generate a cycle. When we process a query containing a reference cycle, e.g. the query shown in Fig. 9(b), many DAG-based graph matching algorithms, e.g. [23][7], are no longer effective. In our approach, two query objects involved in a reference cycle, i.e. two member nodes in this case, play both a referencing part and a referenced part. Thus we ignore the constraints under both of them during pattern matching, and handle these constraints by table joins. For the query in Fig. 9(b), we first match a rewritten query as shown in Fig. 9(c) to the document tree to get the labels of all satisfied members. By joining the result with itself through the member table (shown in Fig. 9(a)) twice, we can easily handle the cyclic situation, and output the desired values.

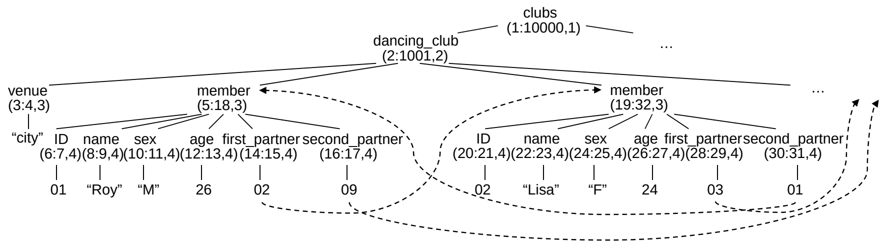
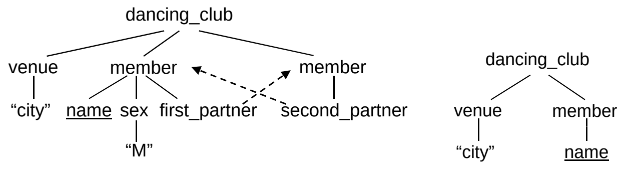


Fig. 8. XML document with cyclic reference

R_{member}

Label	ID	Name	Sex	Age	First_partner	Second_partner
(5:18,3)	01	Roy	M	26	02	09
(19:32,3)	02	Lisa	F	24	03	01
...	...	...	...	...	...	...

(a) Table for *member*



(b) Example query (c) Rewritten query

Fig. 9. Example query involving cyclic reference

4.2 Sequential reference

Sequential references happen when one object references another object, while that object also references a third object. One example document with sequential references is shown in Fig. 10. In this research community document, each *seminar* has a *chair* whose detailed information is stored in some other part of the document; and each *people*'s *affiliation* is also stored in detail separately. The ID/IDREFs between *seminar*, *people* and *affiliation* form a set of sequential references. A reference pattern query with a sequential reference is shown in Fig. 11(b). In this query, we try to find the *topic* of a *seminar* in the 'database' *research area*, which is chaired by a *people* from 'NUS'.

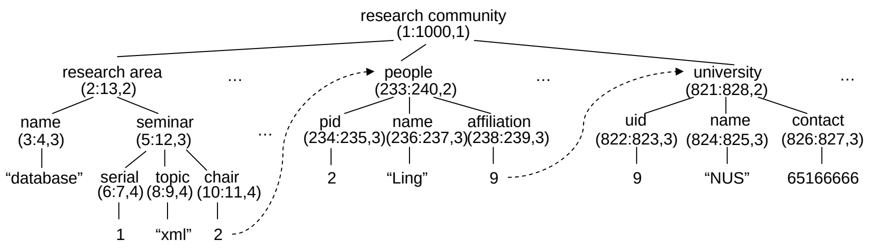


Fig. 10. XML document with sequential reference

$R_{seminar}$

Label	Serial	Topic	Chair
(5:12,3)	1	XML	2
...	...	...	...

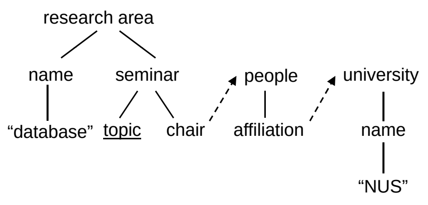
R_{people}

Label	Pid	Name	Affiliation
(233:240,2)	2	Ling	9
...	...	...	...

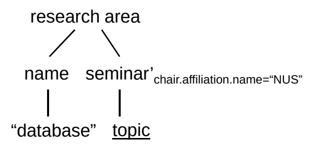
$R_{university}$

Label	Uid	Name	Contact
(821:828,2)	9	NUS	65166666
...	...	...	...

(a) Tables involved



(b) Original query



(c) Rewritten query

Fig. 11. Example query involving sequential reference

Processing queries involving sequential references in an XML document increases the complexity in many traditional subgraph matching algorithms. Sequential references lead to more cycles if we do not consider the directions of each reference. As we know, in traditional approaches, subgraph matching is done by generating all possible maps between nodes in two graphs and then filtering out incorrect answers. With more cycles, incorrect mappings cannot be pruned as early as that in graphs with fewer cycles. In our approach, we do not need to consider this aspect. For the query in Fig. 11(b), we just perform selection and join between tables for *seminar*, *people* and *university* (shown in Fig. 11(a)).

The selected *label* values for *seminar* will be used to construct a new inverted list for query node *seminar*. Then the original query is rewritten to a new query as shown in Fig. 11(c) by removing references and some other query nodes.

4.3 Complex reference

Theoretically, a general reference pattern query can be very complex with cyclic and sequential ID references. The last two sections show that table join is powerful to handle both kinds of references between two twig parts. When we deal with a reference pattern query with complex references, we simply decompose the query into twig parts, each of which contains an object referencing or being referenced by other parts. The reference between different parts is solved by table join, and when a twig part is complex itself, we can perform a pattern matching to solve it. We will further research on query optimization on queries with complex reference.

5 Experiments

In this section, we present experimental results, comparing our approach with an XQuery engine [2] and *TwigStackD* [7], which is a stack based approach for XML query processing involving ID/IDREF, which has proven more efficient than traditional graph matching methods. For convenience, we name our table based method as *TBM*. Another recent work on XML graph matching [24] may not be correct when it models a graph pattern query. Thus we do not do a comparison with it[4].

5.1 Experimental settings

Implementation: We implemented all algorithms in Java. The experiments were performed on a 3.0GHz Pentium 4 processor with 1G RAM.

XML Data Sets: We used three XML data sets for our experiments: Gene ontology data, purchase data and XMark data. Gene ontology data is a 70MB real-life data set, which is taken from a Gene Ontology Project [1]. Purchase data is a 12MB synthetic data set generated by our data generator. The schema of this document is similar to the schema of our example document shown in Fig. 1(a). The characteristics of this document is a large number of ID references, as every part has a supplier reference. We also use 9 XMark benchmark [3] documents with the size varying from 11MB to 111MB to compare document parsing time, and use one of them (23MB) to test execution time. XMark documents contain multiple types of ID references.

Queries: We randomly selected five meaningful queries with ID references for each data set. The queries are shown in Fig. 12. The last element in each query expression is the output node. We only consider the first two cases where the output node resides in either the referencing part or the referenced part in each query. The third case is just a combination of the first two cases. ID references in queries are denoted by '→', and some queries (Q1, Q5, Q11, Q15) contain sequential references.

5.2 Experimental results and analysis

Comparison with XQuery engine This experiment is done using *MonetDB* [2], which is a well known memory-based XQuery engine, and a relatively small XMark document (11MB) so that all the processing can be done in memory.

[4] More explanations are available at http://www.comp.nus.edu.sg/∼wuhuayu/problem.pdf

Query	Data Set	Path Expression
Q1	Gene	//term[n_associations=0][isa/resource→term/isa/resource →term/name=‘molecular_function’]/accession
Q2	Gene	//term[accession=‘GO0016329’]/isa/resource →term/association/evidence/evidence_code
Q3	Gene	//term[name=‘anticoagulant’]/isa/resource →term/dbxref[reference]/database_symbol
Q4	Gene	//term[accession=‘GO0016172’][name=‘antifreeze’][//resource →term/dbxref[reference]]/about
Q5	Gene	//term[n_associations=0][isa/resource→term/isa/resource →iterm/isa/resource→iterm/dbxref[reference]]/accession
Q6	Purchase	//part[name=‘phone’][supplierRef →supplier/location=‘Sydney’]/price
Q7	Purchase	//department[part[name=‘PC’]/supplierRef →supplier/phone=‘345’]/head
Q8	Purchase	//department[head=‘Fione’]/part[name=‘sofa’]/supplierRef →supplier/location
Q9	Purchase	//department[head]/part/supplierRef→supplier[location=‘London’]/phone
Q10	Purchase	//department[name=‘R&D’]//supplierRef→supplier/location
Q11	XMark	//open_auction[itemref →item[location=‘United States’]/incategory →category/name=‘Seeming mingle teach’]/current
Q12	XMark	//bidder[date=’11/13/2001’]/personref →person[address/city=‘Birmingham’][profile/gender=‘male’]/phone
Q13	XMark	//person[profile[education=‘High School’]/age=38][watches/watch →open_auction/initial=71.36]/name
Q14	XMark	//closed_auction[buyer/person→person[address/province=‘Haban’]][seller/ person→person[address/city=‘Lisbon’]]/price
Q15	XMark	//bidder[increase=‘21’][personref→person[age=’35’][//watch →open_auction/reserve=50.84]]/date

Fig. 12. Experimental queries

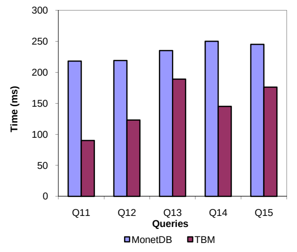
MonetDB uses an optimized node-based relational approach [14] to process XML queries. First using *MonetDB* and *TBM*, we process the queries Q11-Q15 which include sequential reference cases.The XQuery expression of Q11 is shown in Fig. 13(a). Other queries can also be expressed as XQuery expressions in a similar way. Due to the space limitations, we do not show the XQuery expression of every query. The experimental result is shown in Fig. 13(b). In the second step, we test queries with cyclic references. In XMark data, we observe that each *person* has several *watches*, each *watch* contains an *open_auction*, each *open_auction* has *bidders*, and each *bidder* is a *person*. We randomly compose five queries within this cycle and the execution time for the two methods is shown in Fig. 13(c).

TBM outperforms *MonetDB* for all the queries by 20-50%. The reason is that XQuery cannot express queries involving ID references using a single path. Instead, XQuery has to do a multiple retrieval for the referencing part and the referenced part of the query, and then do a join between the retrieved results. However, using our method, we handle the reference separately, solving the reference constraint, and also simplifying the query structure and search space.

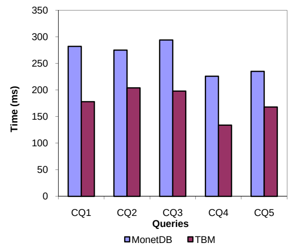
Comparison with TwigStackD Our experiments mainly compare the query processing time and document parsing time between *TwigStackD* and our table based methods (*TBM*). We used B^+ trees to organize inverted lists for both approaches to ensure high performance of inverted list accessing. The execution time for *TBM* includes the time of processing ID reference with table joins and

```
Q11:
FOR $o IN doc("XMark.xml")//open_auction
FOR $i IN doc("XMark.xml")//
        item[location="United States"]
FOR $c IN doc("XMark.xml")//
        category[name="Seeming mingle teach"]
WHERE $o/itemref/@item=$i/@id AND
        $i/incategory/@category=$c/@id
RETURN <open_auction current={$o/current}/>
```

(a) Q11 in XQuery

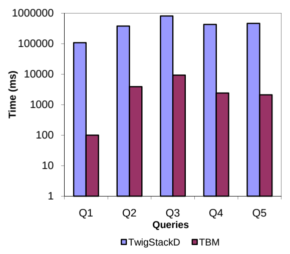


(b) Result for Q11-Q15

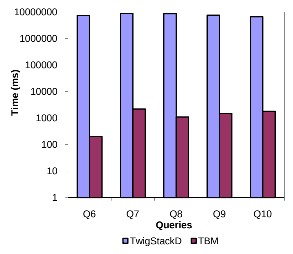


(c) Result for cyclic cases

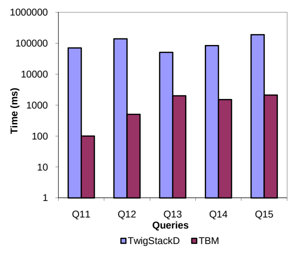
Fig. 13. Query performance comparison between *MonetDB* and *TBM*



(a) Gene Ontology



(b) Purchase



(c) XMark

Fig. 14. Execution time by *TwigStackD* and *TBM*

the time of structural joins during tree pattern matching. The results of execution time are shown in Fig. 14 (the Y-axis is in logarithmic scale). From the results, we can see for all queries the performance of *TBM* is more than 100 times faster than *TwigStackD*. The reason why *TwigStackD* is worse is that their approach uses graph matching rather than tree matching. As a result, in *TwigStackD* they maintain an index to store position relationships between nodes. Each time they process a query, partial solutions are expanded based on the index. This stage is very costly and seriously affects the performance.

In the purchase document, the schema is simple but contains lots of references between *part* and *supplier*. When *TwigStackD* expands partial solutions in a pool, the pool size and amount of checking is very large. That is why the execution time of *TwigStackD* in purchase document is much greater than that in the other two documents where fewer references are involved. The XMark document contains four types of ID/IDREF on objects *item*, *category*, *person* and *open_auction*. Due to the complex references, the index in *TwigStackD* is very large (nearly 3 times greater than the original document size) and it takes quite a long time to build such an index.

Finally we conducted experiments on document parsing time between *TBM* and *TwigStackD*. This parsing time includes node labeling, inverted list constructing and other index building. All these operations are required for structural join based native XML query processing algorithms. In our experiment, we took 12 XMark documents with different numbers of nodes. The results given in Fig. 15, show that when the size of the document grows, the parsing time for *TwigStackD* increases quickly, while using our approach the parsing time is more acceptable for different document sizes.

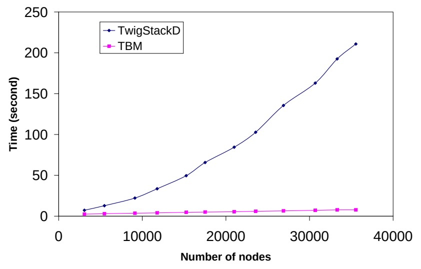


Fig. 15. Document parsing time comparison

6 Conclusions and further work

In this paper we analyze the drawbacks of existing work for query processing in XML documents with ID/IDREF. In particular, most existing algorithms consider XML documents and queries as graphs, and perform graph matching to process queries. However, graph matching is generally believed to be less efficient than tree matching, which is a widely accepted approach to process XML queries without considering ID references. Motivated by this finding, we propose a table based semantic approach to reduce graph matching to tree matching to process XML queries involving ID references. When we parse an XML document, we only consider the native hierarchical structure, and do not treat it as a graph with ID references. We build relational tables for each object that may contain ID or IDREF attributes. During query processing, we decompose a reference pattern query into a referencing part and a referenced part using the ID reference involved. Now the referencing part will be a simple tree structure that can be matched to the document tree. The reference between the referencing part and the referenced part is eventually transformed to a table join between the two parts. The experimental results show that our approach is 20-50% more efficient than MonetDB and more than 100 times faster than TwigStackD, a structural join based graph matching algorithm. Furthermore, our approach can also handle complex ID/IDREF relationships such as cyclic references and sequential references, which are bottlenecks for many existing works.

For further work, we will further investigate how to generate a better query plan when dealing with both tree pattern matching and table joins.

References

1. http://www.geneontology.org/.
2. MonetDB. http://monetdb.cwi.nl/.
3. XMark. An XML benchmark project. http://www.xml-benchmark.org, 2001.
4. A. Berglund, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language XPath 2.0. W3C Working Draft, 2003.
5. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query. W3C Working Draft, 2003.

6. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In SIGMOD, pages 310-321, 2002.
7. L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In VLDB, pages 493-504, 2005.
8. S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over XML documents. In VLDB, pages 283-294, 2006.
9. T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In SIGMOD, pages 455-466, 2005.
10. A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In SIGMOD Conference, pages 431-442, 1999.
11. W. Fan, J. X. Yu, H. Lu, J. Lu, and R. Rastogi. Query translation from XPath to SQL in the presence of recursive DTDs. In VLDB, pages 337-348, 2005.
12. M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.
13. G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. IEEE Trans. Knowl. Data Eng., 19(10):1381-1403, 2007.
14. T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. ACM Trans. Database Syst., 29(1):91-131, 2004.
15. H. Jiang, H. Lu, and W. Wang. Efficient processing of twig queries with or-predicates. In SIGMOD, pages 59-70, 2004.
16. M. Jiang. Querying XML data: efficiency and security issues. Ph.D. Thesis. The Chinese University of Hong Kong.
17. B. Kimelfeld, and Y. Sagiv. Twig patterns: from XML trees to graphs. In WebDB, pages 26-31, 2006.
18. R. Krishnamurthy, V. T. Chakaravarthy, R. Kaushik, and J. F. Naughton. Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In ICDE, pages 42-53, 2004.
19. J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of XML twig pattern matching. In VLDB, pages 193-204, 2005.
20. J. Morgenthal and J. Evdemon. Eliminating redundancy in XML using ID/IDREF. XML Journal, 1(4), 2000.
21. 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 VLDB, pages 302-314, 1999.
22. D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In PODS, pages 39-52, 2002.
23. Z. Vagena, M. M. Moro, and V. J. Tsotras. Twig query processing over graph-structured XML data. In WebDB, pages 43-48, 2004.
24. H. Wang, J. Li, J. Luo, and H. Gao. Hash-based subgraph query processing method for fraph-structured XML documents. In VLDB, pages 478-489, 2008.
25. H. Wu, T. W. Ling, and B. Chen. VERT: A semantic approach for content search and content extraction in XML query processing. In ER, pages 534-549, 2007.
26. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Internet Techn., 1(1):110-141, 2001.
27. C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In SIGMOD Conference, pages 425-436, 2001.