SemanticTwig: A Semantic Approach to Optimize XML Query Processing

Zhifeng Bao[1], Tok Wang Ling[1], Jiaheng Lu[2], and Bo Chen[1]

[1] School of Computing, National University of Singapore
{baozhife, lingtw, chenbo}@comp.nus.edu.sg
[2] University of California, Irvine, jiahengl@uci.edu

Abstract. Twig pattern matching (TPM) is the core operation of XML query processing. Existing approaches rely on either efficient data structures or novel labeling/indexing schemes to reduce the intermediate result size, but none of them takes into account the rich semantic information resided in XML document and the query issued. Moreover, in order to fulfill the semantics of the XPath/XQuery query, most of them require costly post processing to eliminate redundant matches and group matching results. In this paper, we propose an innovative semantics-aware query optimization approach to overcome these limitations. In particular, we exploit the functional dependency derived from the given semantic information to stop query processing early; we distinguish the output and predicate nodes of a query, then propose a query breakup technique and build a query plan, such that for each distinct query output, we avoid finding the redundant matches having the same results as the first match in most cases. Both I/O and structural join cost are saved, and much less intermediate results are produced. Experiments show the effectiveness of our optimization.

1 Introduction

XML is emerging as a standard for information exchange and representation. Efficient processing of XML queries attracts wide research attentions recently [2, 4, 10–12, 3]. The structure of an XML query [7, 13] is generally modeled as a twig pattern (i.e. a small tree), while the values of XML elements or attributes are used in selection predicates. Twig pattern edges are either parent-child (P-C) or ancestor-descendant (A-D) relationships, denoted by "/" and "//". E.g. an XML query *book[title="XML"]/author* returns the authors of all books titled "XML". *author* is the output, and *title="XML"* is a value-based selection predicate.

So far, many twig pattern matching (TPM) methods have been proposed. Bruno et al. proposed a holistic *TwigStack* join algorithm to avoid producing large intermediate results [2]. Several following approaches [11, 12, 10, 4] suggest different ways to optimize *TwigStack*. However, there are two important issues that the existing TPMs haven't addressed. Firstly, existing TPMs do not distinguish the output nodes and predicate nodes of a query and they return the entire twig matches. However, many of them contribute to the same output result, and the entire twig matches are unnecessary for most XPath or XQuery queries. After all twig matches are found, they have to apply a post-processing, which includes eliminating redundant matches having the same query result (and grouping part of results into a set for the variables in *let* clause of XQuery). This post-processing is very costly, as shown in our experiments in section 5. Secondly, none of them makes use of the rich semantic information resided in the XML document and its schema to further optimize query processing.

Motivated by the above two observations, we propose an innovative semantics-aware query optimization approach. There are three directions to exploit the existing information to optimize query processing: (1) optimizations based on the schema information of XML documents; (2) optimizations based on the query structure; (3) optimizations involving data storage and statistics (e.g. indexing). Our work includes the first two: the first optimization (section 4.2) is based on the ORA-SS model [14] described later; the second optimization (section 4.3) is independent of ORA-SS model. To the best of our knowledge, this is the first work that systematically exploit the known semantics in both XML data and XML query to optimize query processing. As a result, for each distinct query output t, we only find the first twig match in XML document that contains t, and the cost of finding the remaining matches that return the same output as the first match is avoided in most cases. The query processing can stop even earlier if we exploit the semantics of XML document. As a result, both the I/O and CPU cost is significantly reduced.

Another challenge is how to efficiently represent the semantic information in XML documents. Although XML is the standard for publishing and exchanging data on the Web, most business data is managed and will remain to be managed by relational database management system (RDBMS) because of their powerful data management services. There is an increasing need to accurately publish relational data as XML documents for Internet-based applications as well as preserve the semantics captured in RDBMS. In the transformation from relational schema to XML schema, some semantics can be captured by DTD and XMLSchema, such as the *identifier constraint* (denoted by "unique" and "key" constraint in XMLSchema), and *participation constraint* of child element on its parent denoted by signs "$*$", "?" and "$+$". However, richer semantics captured in RDBMS cannot be captured by DTD/XMLSchema, e.g. the *participation constraints* of a parent element on its child, etc. Fortunately, instead of writing down those semantics as comments, notes, or in XML data creator's own mind, we can capture them in ORA-SS schema [14], which is a useful data model and tool for semistructured data. The main advantage of ORA-SS is its ability to distinguish between object classes, relationship types and attributes, and express the degree of an n-ary relationship type. Therefore, in this paper, in order to fully leverage the semantic information in XML data to speedup XML twig query processing, we adopt ORA-SS to model XML data, especially those that are transformed from relational data.

The main contributions of this paper are summarized as follows:

– We exploit important semantics of XML document captured by ORA-SS schema (such as object class identifier, participation constraints, etc.), to derive functional dependencies useful for query optimization. This is the first kind of optimization which depends on the ORA-SS model.

– We propose a *Query Breakup technique* to distinguish the processing of predicate part and output part of a twig query, so for each distinct query result, we only find the first twig match in XML document that contains it, and avoid finding all the remaining redundant matches. This is the second kind of optimization, which is independent of the ORA-SS model.
– We propose *SemanticTwig*, a semantics-aware query processing algorithm which employs the above two kinds of optimizations. Both the I/O and CPU cost reduce significantly compared to existing TPMs. Moreover, our optimization is orthogonal to any existing structural-join based TPMs.
– We perform comprehensive experiments to demonstrate the efficiency and scalability of our approach over existing approaches.

The rest of the paper proceeds as follows. Section 2 reviews the related work. Section 3 gives an overview of ORA-SS. Section 4 propose two query optimization approaches. Section 5 reports experimental results and we conclude in section 6.

2 Related Work

Extensive research efforts have been put into efficient twig query processing with label based structural join. For binary structural join, Zhang et al. [15] proposed a multi-predicate merge join (MPMGJN) algorithm based on region labelling of XML elements. The later work by Al-Khalifa et al. [1] gave a stack-based binary structural join algorithm, called Stack-Tree-Desc/Anc. However, both generate many useless intermediate results for twig query. To solve this problem, Bruno et al. [2] firstly proposed a holistic twig join algorithm called *TwigStack* to solve the problem of useless intermediate results. However, *TwigStack* is only optimal for twig query with only A-D edges in terms of intermediate results. Many subsequent works try to optimize *TwigStack* in terms of I/O. In particular, Lu et al. in [11] introduced a List structure called TwigStack-List for a wider range of optimality. Jiang et al. in [10] proposed an XML Region Tree (XR-tree) index structure and a TSGeneric+ algorithm to effectively skip both ancestors and descendants that do not participate in a join. Chen et al. [4] exploits different data partition methods to boost the holism. Lu et al. [12] used Extended Dewey labeling scheme, and proposed a TJFast algorithm to access labels of leaf nodes only. Recently, Chen et al. in [3] proposed a *Twig*^{2}*Stack* algorithm which uses *hierarchical-stacks* to enumeration the twig matches, but it has to maintain a large amount of intermediate results in memory.

Existing semantic related works[5, 6] only rely on the integrity constraints captured from the real world knowledge, rather than the schema of XML document. It prepares a set of XML rewriting rules with semantic preserving property from the integrity of the database, and a query is transformed into an efficient and optimized form using these rules. It should be highlighted that sine relational-based ones like XPath accelerator [9] and PPFS+ [8] also focus only on output nodes and handle predicates using the exist clause of SQL; however they cannot handle twig query well.

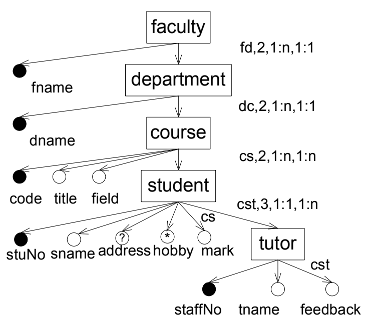
3 Background on ORA-SS Model

The ORA-SS (Object-Relationship-Attribute model for SemiStructured data) data model has three basic concepts: *object class*, object classrelationship type and relationship typeattribute. An *object class* is similar to an entity type in an ER diagram. A *relationship type* describes a relationship among object classes. *Attributes* are properties belonging to an object class or a relationship type. A full description of the data model can be found in [14].

As Figure 1(b) shows, an ORA-SS *schema* represents an object class as a labeled rectangle, an attribute as a labeled circle. The relationship type between object classes is assumed on any edge between two objects, and described by a label in form of "$name(object\_class\_list)$, n, p, c". Here, *name* denotes the name of relationship type; optional *object_class_list* is the list of participating objects; n indicates the degree of the relationship type; p and c are the participation constraints of parent and child object classes respectively, defined using the min:max notation. All attributes are assumed to be mandatory and single valued, unless the circle contains a "?" indicating it is optional and single valued, "+" indicating it is mandatory and multi-valued, and "*" indicating it is optional and multi-valued. Identifier of an object class is a filled circle. The attribute of a relationship type has the name of the relationship type to which it belongs on its incoming edge, while the attribute of an object class has no edge label.

```
<!ELEMENT faculty (fname, department+)>
<!ElEMENT fname  #PCDATA>
<!ELEMENT department (dname, course+)>
<!ElEMENT dname  #PCDATA>
<!ELEMENT course (code, title, field, student+)>
<!ElEMENT code  #PCDATA>
<!ElEMENT title  #PCDATA>
<!ElEMENT field  #PCDATA>
<!ELEMENT student (stuNo, sname,
                        address?, hobby*, mark,tutor)>
<!ElEMENT sname  #PCDATA>
<!ElEMENT stuNo #PCDATA>
<!ElEMENT address  #PCDATA>
<!ElEMENT hobby  #PCDATA>
<!ElEMENT mark  #PCDATA>
<!ElEMENT tutor (staffNo, tname, feedback)>
```

(a) DTD



(b) ORA-SS Schema Diagram

Fig. 1. Two Schema Representations of XML Document

Figure 1(a) shows the DTD and Figure 1(b) shows an ORA-SS schema diagram for an XML document describing the organization of a university. The rectangles labeled *faculty*, *department*, *course*, *student* and *tutor* are five object classes, and attributes *fname*, *dname*, *code*, *stuNo* and *staffNo* are the identifiers of *faculty*, *department*, *course*, *student* and *tutor* respectively. For each *student*, *address* is an optional single valued attribute, and *hobby* is an optional multi-valued attribute. There are three binary relationship types, namely *fd*, *dc* and *cs. cs* is a relationship type between *course* and *student. dc* represents a one-to-many relationship type, where a *department* can have one or more (1:n) *courses*, and a *course* belongs to exactly one (1:1) *department.* The label *cs* on the edge between *student* and *mark* indicates that *mark* is a single valued attribute of the relationship type *cs*, and a functional dependency (FD) $\{course, student\} \rightarrow$ *mark* holds. A ternary relationship type *cst* involves *course*, *student* and *tutor*. The label *cst* on the edge between *tutor* and *feedback* indicates *feedback* is an attribute of relationship *cst*, and an FD $\{$*course,student,tutor*$\} \rightarrow$ *feedback* holds.

4 Semantic Query Optimization

This section investigates how our semantics-aware query optimization works. Our primary objective is to avoid unnecessary computation on finding matches contributing to the same query result and stop query processing as early as possible. The optimization is carried out in two steps: we first try to exploit the *FD*s explicitly given or derived from the semantics of the *XML document* to answer a query through FD's ability of capturing redundancies(section 4.2); if no FD can be exploited, then we make use of the semantics of the *XML query*, i.e. distinguish the output part and predicate part of a query, in which the predicate is only used for existence checking. Therefore, we only need to find the matches that have distinct output results, rather than the matches of the entire query's twig pattern (section 4.3). In addition, our optimization is orthogonal to most existing structural join based TPMs. The optimization approach in section 4.2 relies on the semantics exclusively captured by ORA-SS schema, while section 4.3's approach works without dependence on ORA-SS. Note that, *FD* could be given explicitly instead of using ORA-SS model here.

4.1 Terminology

In this paper, we discuss three forms of *predicates*: (1) value predicate for point query (in form of *A[m = "value"]//B*), and its *predicate node set* is a set of nodes m involved in the equality condition; (2) the predicate used for existence checking (in form of *A[n]//B*), and its *predicate node set* is a set of nodes n; (3) the mixture of the above two forms, and its *predicate node set* is the union of the *predicate node set* of the above two forms. E.g. given a query *A[//B[C[D="v1" and E="v2"]]]/F*, *predicate node set* = $\{$*D,E,v1,v2*$\}$, because we treat the element values "*v1*" and "*v2*" same as the element node. For query *A[B//C[D and E]]/F*, *predicate node set* = $\{$*B,C,D,E*$\}$. Although a twig query does not specify output and predicate explicitly, it is easy to identify them from its corresponding XQuery query. The *output node set* contains all nodes in its *return* clause.

Moreover, instead of returning the entire twig results, we return only the output part of the query. The result is defined as a set of tuples, and each tuple contains the element labels of output nodes only. If some output node n is a multi-valued attribute and required to return as an ordered set, its matching labels are grouped into a set under their common ancestor before returned. Our approach outperforms the existing TPMs by avoiding the cost on finding as many *redundant matches* as possible, which is defined below.

Definition 1 {Redundant Match} *For a twig query Q, a match $M1$ is said to be a redundant match, if there exists another match $M2$ found before $M1$, such that $M1$ and $M2$ contribute to the same query output node values.*

4.2 Optimization via Functional Dependency

Functional dependency (*FD*) is used to model real world constraints and capture redundancies. The given semantics in ORA-SS schema such as the identifier of object class, n-ary relationship and the participation constraint can be used to derive useful *FD* for efficient query processing. Given a query Q, if an FD in form of *predicate node set*$\rightarrow$*output node set* can be derived, then Q is answered by finding the first match of Q appearing in XML document. The I/O cost is saved since it avoids loading remaining labels from each query node's stream; the CPU cost is saved since most XML documents contain redundancies. But it is only true when Q involves the *value predicates.* All Lemmas hold based on this assumption.

FD within an object class The identifier of an object o can uniquely determine the value of any single-valued attribute of o, and uniquely determine the whole set of values of any multi-valued attribute of o.

Lemma 1 *Given a query Q with predicate node set P and output node set T.*

Case 1: If $\forall t_i \in T$, t_i is a single-valued attribute of some object class O_i, and $\exists p \in P$, such that p is the identifier of O_i; then an FD $P \rightarrow T$ holds, and the query result is the first match F *of Q over XML data in document order.*

Case 2: If some output node $t'_j \in T$ is a multi-valued attribute of some object class O_j, and other output nodes in T are same as Case 1*; then Q is answered by finding its first match* F *(which finds the first value of each t'_j), then retrieving and grouping the remaining values of each t'_j within* F *into a set.*

In Case 1, the overall processing cost is reduced with the utilization of such FD in two ways. Firstly, an object may have more than one occurrence in document, and it's safe to stop query processing after the first match of Q is found. Secondly, there is no need to check the remaining labels in label streams of each query node, which saves I/O cost. In Case 2, we only need to find one match; however, existing approaches have to enumerate all path matches and merge-join them, and finally eliminate redundant match. One important character that distinguishes our approach from the existing TPMs is that we treat the set/non-set elements(i.e. multi/single-valued attributes) separately. The retrieval and grouping of the remaining occurrences of t' are easy and efficient to implement, since their labels are stored sequentially and compactly.

Example 1 Refer to ORA-SS schema in Figure 1(b)[1], the following XQuery query Q retrieves the name and hobby of a student with *stuNo* equal to "u12".
for \$s in //student[stuNo="u12"], let \$h := \$s/hobby
return <stu>{\$s/sname}{\$h}</stu>

The above query Q's twig pattern is shown in Figure 2. Its *predicate node set* P = {*stuNo, u12*}, and *output node set* T = {*sname*, *hobby*}. *stuNo* is the identifier of *student* and *hobby* is a multi-valued output, so by Case 2 of Lemma 1, it is enough to find the first match F of Q in XML data, then retrieve and

[1] All queries throughout this paper refer to the schema diagram shown in Figure 1(b).

group all labels of remaining *hobbies* within *F*. A tuple containing *sname* and a set of *hobbies* of this student is returned. The advantage of our approach

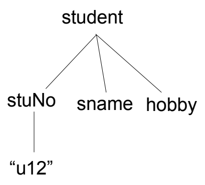


Fig. 2. Example 1

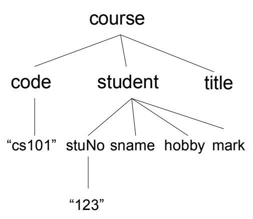


Fig. 3. Example 2

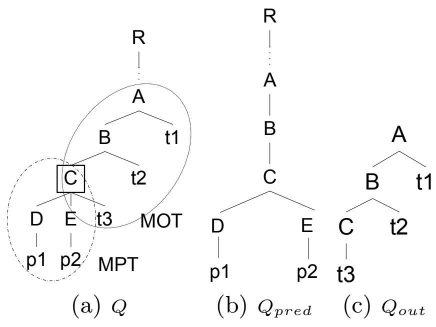


Fig. 4. Break query (Example 4)

is illustrated by an example as below. Assume student *"u12"* has 8 hobbies. Existing TPMs find 8 matches of path *//student/hobby* and join the 8 matches with the match(es) of paths *//stduent/stuNo/ u12* and *//student/sname*, while our approach needs only one structural join. Moreover, after finding 8 matches of the entire twig, TPMs group all hobbies, which is a costly post-processing operation; while we handle the grouping in the middle of twig pattern matching. TPMs even need to do redundancy elimination if this student object has more than one occurrences in XML document.

FD in an n-ary relationship type From ORA-SS schema diagram, the FD in an n-ary relationship is in the form that, identifiers of participating object classes functionally determine the single-valued attributes of the relationship.

Lemma 2 *Given a twig query Q with predicate node set P and output node set T. For each output node t∈T,*

Case 1: t is a single-valued attribute of a n-ary relationship R involving n object classes O_1,...,O_n, and ∃p_i∈P for each i∈[1,n], s.t. p_i is the identifier *of O_i;*

Case 2: t is a single-valued attribute of some object class O_k, ∃p_k∈P s.t. p_k is the identifier of O_k. Then in both cases, FD P→T holds, and Q can be answered by finding the first match F *of Q;*

Case 3: if some t∈T is a multi-valued attribute of R or any participating object class, and each remaining node t′∈T belongs to either Case 1 or Case 2, then we can adopt the approach in Case 2 *of Lemma 1 to efficiently answer Q.*

Example 2 Find the title of course with code "cs101", and the mark, name and hobby of student taking it, whose stuNo is "123".
for $c in //course, for $s in $c[code="cs101"]/student[stuNo="123"]
let $h := $s/hobby
return <stu> {$c/title} {$s/sname} {$s/mark} {$s/hobby} </stu>

Figure 3 shows the twig pattern of the above query *Q*. The predicate set *P* = {*code*, *stuNo*}, the output node set T = {*title*, *sname*, *mark*,*hobby*}, in which *mark* is a single-valued attribute of relationship type *cs*. From Figure 1(b), it is easy to identify the FD $\{code, stuNo\} \rightarrow mark$, which means a student has exactly one mark for each course taken. *stuNo*→*sname* and *code*→*title* also hold. So we can infer $\{code,stuNo\} \rightarrow \{sname,title,mark\}$. So by Lemma 2 Case 3, Q can be answered by finding its first twig match F in XML document, followed by retrieving and grouping the values of remaining *hobbies*.

Participation constraint in n-ary relationship The 1:1 participation constraint intuitively infers an FD between the participating object classes. A typical example is the *one-to-many relationship type dc* shown in Figure 1(b), in which the 1:1 participation of *course* on relationship type *dc* specifies a course is offered by exactly one department, i.e. *course*→*department*. The 1:1 participation between some object classes can simplify the functional dependency of an n-ary relationship R. E.g. as Figure 1(b) shows, single-valued attribute *feedback* of ternary relationship *cst* is determined by the 3 participating object classes, i.e. $\{course,student,tutor\} \rightarrow feedback$. The 1:1 participation of *tutor* on *cst* specifies $\{course,student\} \rightarrow tutor$, which means a student has exactly one tutor for each course he takes. Therefore, a new FD $\{course,student\} \rightarrow feedback$ is derived.

Example 3 Find the name of faculty and department offering course "cs101".
for \$f in //faculty, \$d in \$f/department, \$c in \$d/course[code="cs101"]
return <fac>\$f/fname<dept>\$d/dname</dept></fac>

In this query, two binary relationships *fd* and *dc* exist, and the participation of parent node on child node are both 1:1, so two FDs *course*→*department* and *department*→*faculty* hold. Then a new FD *course*→ $\{faculty,department\}$ is inferred. Since the predicate node *code* is the identifier of *course*, the query has a unique answer and we can stop query processing after the first match is found.

4.3 Optimization via Query Breakup

Motivation According to the semantics of an XPath/XQuery query, only the output results with no duplicates are expected to return. However, existing approaches are not aware of this distinction of output and predicate nodes in a query, and assume all nodes in a query tree need to be output. They answer a query by applying pattern matching on its entire twig, followed by a costly post-processing which includes the tasks of redundant matches elimination and results grouping. In this section, we aim to fulfill the post-processing task during the pattern matching procedure with low extra cost. In fact we are able to avoid finding as many redundant matches as possible by distinguishing the output nodes and predicate nodes of a query.

Another motivation to propose the query breakup technique is: for each distinct output result of the query Q, there are many redundant matches of the predicate part in XML document. If we can break Q into two sub-twigs corresponding to Q's predicate and output part respectively, and avoid finding those redundant matches, then both I/O and CPU cost are reduced. Because we skip reading the elements of redundant matches into memory, and need less number of structural joins, and process a twig query of smaller size.

Find the breakpoint The choice of breakpoint is not unique, but it depends on the definition of *predicate node* of a query (defined in section 4.1), and its choice determines the breakup method used. Therefore, the *predicate node*, breakpoint, the breakup method and query optimization based on breakup are defined in a consistent and cooperative way to meet three properties: (1) guarantee the completeness and correctness of query result; (2) read as few elements as possible into memory; (3) less structural join cost. Before introducing the definition of breakpoint, we have the following two definitions.

Definition 2 {*Minimal Predicate Tree (MPT)*} *The Minimal Predicate Tree of a query Q is the minimal sub-tree of Q's twig tree that covers all nodes in the predicate node set. If there is only one predicate node p, the MPT is a path connecting p and its parent. The root node of MPT is called R_{MPT}.*

Definition 3 {*Minimal Output Tree (MOT)*} *The Minimal Output Tree of a query Q is the minimal sub-tree of Q's twig tree that covers all nodes in the output node set. If there is only one output node t, the MOT is a path connecting p and its parent. The root node of MOT is called R_{MOT}.*

Definition 4 {Breakpoint} *The breakpoint* BP *of the query Q with the predicate node set P and the output node set T is:*

*Case 1: the lowest common ancestor **LCA** of R_{MPT} and R_{MOT} in Q, if there is no A-D relationship between R_{MPT} and R_{MOT}; (line 7 of Algorithm 1)*

Case 2: the node $t \in MOT$, such that t is the ancestor of an output node of Q, and t has the lowest hierarchy along the path downward from R_{MOT} to R_{MPT}, if R_{MOT} is the (self) ancestor of R_{MPT}; (line 3-4 of Algorithm 1)

Case 3: the node $p \in MPT$, such that p is the ancestor of a predicate node of Q, and p has the lowest hierarchy along the path downward from R_{MPT} to R_{MOT}, if R_{MPT} is the (self) ancestor of R_{MOT}. (line 5-6 of Algorithm 1)

Algorithm 1: findBreakPoint(Q, MPT, MOT)

```
1 $r1$ = root($MOT$); $r2$ = root($MPT$)
2 $P_1$ = path($r1$,$r2$); $P_2$ = path($r1$,$r2$)
3 if ($r1$.isAncestorof($r2$)) then
4     brkpoint = t | t∈MOT ∩ t∈$P_1$ ∩ $\forall t' \in T\ t' \neq t \Rightarrow$ level(t)>level(t')
5 else if (r2.isAncestor(r1)) then
6     brkpoint = t | t∈$MPT$ ∩ t∈$P_2$ ∩ ∀t'∈P $t' \neq t \Rightarrow$ level(t)>level(t')
7 else brkpoint = findLowestComAncestor($r1$,$r2$,$Q$)
8 return brkpoint
```

Break up the query Based on the breakpoint BP found, a query Q is broken into two sub-twigs namely Q_{pred} and Q_{out}. Besides the three properties introduced in last subsection, the breakup method should guarantee the union of Q_{pred} and Q_{out} entirely constitute Q. Since some nodes in Q are not covered by either MOT or MPT, the main problem is how to distribute them into Q_{pred} and Q_{out} to meet the above four properties. Since the definition of *Breakpoint* has three cases, the query breakup is presented to handle the query in each case. In Algorithm 2, lines 1-3 handle the query in Case 1 of Algorithm 1; lines 4-10 handle the query in Case 2; optimization is impossible in Case 3, because no redundant match exists (line 11).

Example 4 We show how *findBreakPoint* and *breakup* work for the query type in *Case 2* of Definition 4. In Figure 4, the query Q has two predicate nodes $p1$ and $p2$ and three output nodes $t1$, $t2$ and $t3$. The Minimal Predicate Tree (MPT) and Minimal Output Tree (MOT) of Q are highlighted by dotted circles in Figure 4(a). A and C are the root of MPT and MOT respectively, and A has a higher hierarchy than C. C is chosen as the breakpoint because it satisfies the three conditions in line 4 of *Algorithm 1*: C is a node in both the path P_1 and MOT, and among the three nodes A, B and C that have an output node as descendant, C has the lowest hierarchy. Next, we follow *Algorithm 2* to break Q. We first find the minimal tree Q_{temp} covering both MPT and C (line 5). Figure 4(b) shows Q_{pred}, which is the minimal tree covering both Q_{temp} and Q's root R(line 6). Q_{rem} resulted from removing Q_{pred} from Q except breakpoint C. Since Q_{rem} is a sub-structure of MOT, Figure 4(c) shows the Q_{out} is MOT(line 7-9).

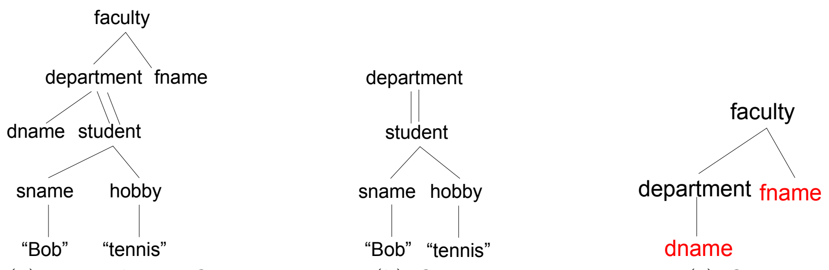
Algorithm 2: breakup(Q, MPT, MOT, BP)

```
1  if (!isAD($R_{MPT}$,$R_{MOT}$) ∩ !isAD($R_{MOT}$,$R_{MPT}$)) then
2      $Q_{pred}$ = findMinTree($MPT$, root($Q$))
3      $Q_{out}$ = ($Q$ - $Q_{pred}$) ∪ $BP$
4  if (isAD($R_{MOT}$,$R_{MPT}$)) then
5      $Q_{temp}$ = findMinTree($MPT$, $BP$)
6      $Q_{pred}$ = findMinTree($Q_{temp}$, root($Q$))
7      $Q_{rem}$ = ($Q$ - $Q_{pred}$) ∪ $BP$
8      if ($Q_{rem}$.isSubTree($MOT$)) then
9          $Q_{out}$ = $MOT$
10     else $Q_{out}$ = $Q_{rem}$
11 else no optimization is possible
```

Example 5 The query Q below finds the name of faculty and department, in which a student descendant is called “Bob” and one of his hobby is “tennis”.
for \$f in //faculty, \$d in \$f/department, \$s in \$d//student
where \$s/sname=“Bob” and \$s/hobby=“tennis”
return <fac>\$f/fname<dept>\$d/dname</dept></fac>

The *predicate node set* P={*sname*,*hobby*,*Bob*,*tennis*}, and the *output node set* T={*fname*,*dname*}. Thus, R_{MPT} is node *student*, and R_{MOT} is node *faculty*. Since *department* is the lowest hierarchy node along the path from *faculty* to *student* and is the ancestor of output node *dname*, the breakpoint is *department* by case 2 of Definition 4. So Q is broken into two sub-twigs Q_{pred} and Q_{out} shown in figure 5(b) and 5(c). If *dname* is removed from the *return* clause of the above query, then the breakpoint is node *faculty* rather than *department*.

Optimization based on query breakup After Q is broken into two sub-queries Q_{pred} and Q_{out}, we build a 4-step query plan to achieve optimization.



(a) original twig Q (b) Q_{pred} (c) Q_{out}

Fig. 5. Breakup a twig query

Step 1: Initialize the set S_{BP} to empty, which stores the labels of breakpoint BP, each of which contributes to a distinct query output.
Step 2: Once a match F of sub-twig Q_{pred} is found, add the label A of current BP-element into S_{BP}.
Step 3: Keep moving the cursor of BP, i.e. C_{BP} forward, until the current BP-element pointed by C_{BP} is not the descendant of A found in F. Go back to *Step 2* to locate the next match of Q_{pred} until C_{BP} reaches the end.
Step 4: Replace the label stream of BP by S_{BP}, and evaluate the sub-twig query Q_{out}. Note that no join between Q_{pred} and Q_{out} is needed.

– Case 1: If every output node $t \in Q$ is a single-valued attribute, then Q is answered by finding all matches of Q_{out} over the XML document.
– Case 2: If some output node $t' \in Q$ is a multi-valued attribute and declared to return as an ordered set (i.e. t' is declared in the *let* clause of XQuery), then Q_{out} is replaced by $Q_{out'}$ by removing all those t' nodes; and once each match FM' of $Q_{out'}$ is located, all labels of such t' nodes within FM' are retrieved one by one and grouped together.

Example 6 This example illustrates the superiority of our approach. The query is same as *Example 5* except that the output node is *fname*. Suppose the university has 8 faculties, 5 departments per faculty, 100 students per department; in each department *dept*, 3 students are called "Bob" and have a hobby "tennis". *TwigStack* [2] is used as a representative of existing TPMs.

TwigStack decomposes Q into 3 root-to-leaf paths, i.e. *//faculty/fname, //faculty/department//student/sname/Bob and //faculty/department//student /hobby/tennis*. The number of matches for each path is 8, 8*5*3 and 8*5*3 respectively. Then it joins these path matches to get the entire twig match of Q, which is 8*5*6*0.5 = 120 matches. Finally, it eliminates redundant matches. In the optimal case, total number of labels scanned is 8+8+8*5*3+8*5*3*4 = 716.

In contrast, our approach finds the breakpoint *department*, and breaks Q into Q_{pred} and Q_{out} (Figure 5). Number of matches of Q_{pred} and Q_{out} are both 5*8, so in total 80 matches are found. In the optimal case, 8+8+8*5 = 56 labels are scanned. 8 structural joins are needed in processing Q_{pred}; no structural join is needed in processing Q_{out}. Compared to *TwigStack*, our approach scans smaller number of labels, needs less structural joins and avoid many redundant matches.

4.4 SemanticTwig Optimization Algorithm

Algorithm 3 presents the two kinds of optimizations introduced in section 4.2 and 4.3. If an *FD* in form of {*predicate part→output part*} can be derived from the semantic information in ORA-SS schema, then the query is answered by only finding the first occurrence of its twig pattern in XML document (lines 1-3). Otherwise, we execute the second optimization, i.e. *query breakup*. Firstly, the *Minimal Predicate Tree* and *Minimal Output Tree* are found (line 4-5). The *breakpoint* n is identified by calling Algorithm 1(lines 6), and used to break Q into Q_{pred} and Q_{out}(line 7). Secondly, the labels of n are collected into S_n, each label contributes to a distinct query output(lines 8-13). The cursor C_n of node n keeps moving forward until the current n is not the descendant of the n-element found in last match of Q_{pred}(lines 12-13). Thirdly, based on the shortened label stream of node n(line 14), all matches of sub-twig Q_{out} are located(lines 15-21). *TwigStack* algorithm is the backbone of $findMatch()$ and $find1stMatch()$.

Theorem 1 *Given a twig query Q with specified predicate node set P and output node set T, and an XML database D. Algorithm SemanticTwig correctly returns all the answers for Q on D.*

PROOF:[Sketch] The main difference of *SemanticTwig* and *TwigStack* is the movement of cursors. In *SemanticTwig*, the cursor of breakpoint n skips all elements that are descendants of the n-element A in previous match of Q_{pred}, but the output results R' associated with these descendants are not skipped. Because each time the first match of Q_{pred} is found, all the matches *MS* of Q_{out} associated with A are located, and R' is in fact a subset of the query results in *MS*. Therefore, it is safe for C_n to directly jump to the first element which is not the descendant of the A in previous match of Q_{pred}.□

Time and Space Complexity

Theorem 2 *Consider an XML database D, a query twig pattern Q consisting of m nodes and ancestor-descendant(A-D) edges only, with specified predicate node set P and output node set T. Algorithm SemanticTwig has the worst-case I/O and CPU complexities linear in the sum of sizes of the m input lists and the output list of Q's sub-twig Q_{out} (which is retrieved via* query breakup*). The upper bound is O($m * |R| + |X|$), where $|X|$ is the size of the m input lists, and $|R|$ is the number of matches of Q_{out}.*

PROOF: *SemanticTwig* first finds the matches of sub-twig Q_{pred} where each match contributes a distinct output result, and it costs O(m_1*$|R|$); then it locates all the matches of sub-twig Q_{out} based on the shrinked label stream S_n of node n, and it costs O(m_2*$|R|$). Since m_1 and m_2 is number of query nodes in Q_{pred} and Q_{out} respectively, we have m_1+m_2= $m + 1$. The cost of reading input streams of each query node is $|X|$. Thus, the total cost is O($m * |R| + |X|$).□

Since *SemanticTwig* calls *twigstack* to find matches of the sub-twigs of Q, and the worst-case size of any stack in TwigStack is proportional to the maximal length of a root-to-leaf path XML database, we have the following results about the space complexity of SemanticTwig.

Algorithm 3: semanticTwig(Q, *predSet*, *outSet*)

```
1  Identify FDs $F_1$, $F_2$,..., $F_m$ from ORA-SS schema of XML data
2  if FD Predicate→output is derived then
3      resultSet += find1stMatch(Q)
4  $MPT$ = findMinPredicateTree($Q$, predSet)
5  $MOT$ = findMinOutputTree($Q$, predSet)
6  Node $n$ = $findBreakPoint$($Q$, $MPT$, $MOT$)
7  {$Q_{pred}$, $Q_{out}$} = breakup($Q$, $n$, $MPT$, $MOT$)
8  let $C_n$ be the cursor of node $n$, let $S_n$ be a label set of $n$
9  while (!end(n)) do
10     $predMatch$ = $find1stMatch$($Q_{pred}$)
11     $S_n$ += $C_n$; prevC = $C_n$; $C_n$ = $C_n$.advance()
12     while ($C_n$.isDescendantof(prevC)) do
13         $C_n$ = $C_n$.advance()
14 $T'$ = $t'$| $t'$ is multi-valued output ∩ $t'$ in let clause of $Q$
15 Replace the label stream of node $n$ by $S_n$
16 while (!end(root($Q_{out}$))) do
17     fm = findMatch($Q_{out}$)
18     Let t1,t2,... be labels of each single-valued node of $Q_{out}$
19     foreach $t' \in T'$ do
20         $Set_{t'}$ += retrieveLabels($t'$, fm)
21     ResultSet += <t1,t2,...,$Set_{t'}$,...>
22 return resultSet
```

Theorem 3 *Consider a query twig pattern Q with m nodes and an XML database D. The worst case space complexity of Algorithm SemanticTwig is the minimum of (i) the sum of sizes of the m input lists, and (ii) m times the maximum length of a root-to-leaf path in D.*

Theorem 4 *The two optimization methods in SemanticTwig are both orthogonal to all existing structural join based TPMs. The only difference is the implementation of method find1stMatch() and findMatch().*□

Moreover, *SemanticTwig* is optimal for twig queries with A-D edges only, but sub-optimal for queries with P-C edges. But this sub-optimality is due to the sub-optimality of *TwigStack* on which our optimization method applies.

5 Experimental Study

We implement *TwigStack* and *SemanticTwig* in JDK 1.4, and run experiments on a 3.0 GHz Pentium 4 processor PC with 1GB RAM running on windows XP system. We compare them in terms of intermediate path solutions, I/O cost (i.e. number of elements read into memory) and total execution time.

In order to evaluate the performance of a particular operation exactly, we choose DBLP as the real dataset; and generate the synthetic dataset based on the university's schema diagram shown in Figure 1(b), by manually specifying semantic constraints such as the uniqueness of certain values, the frequency of some node value in document etc. Three synthetic datasets are used, details summarized in Table 1. *Doc*1 is a non-recursive document corresponding to the schema in Figure 1(b); *Doc*2 is adapted from *Doc*1, in which *course* becomes a recursive element, s.t. a course is the prerequisite of other courses. *Doc*3 is adapted from *Doc*2 by increasing the frequency of some leaf nodes' values. DBLP's summarization is shown in the last row of Table 1.

Table 1. XML Data Sets

Data	Size(MB)	Nodes	Depth
Doc1	10.4	882854	7/4.1
Doc2	11.7	889453	13/5.2
Doc3	18.3	1522218	7/4.5
DBLP	130	3736406	6/2.9

Table 2. Queries over data sets

Q1	//course[code="cs101"]/student[stuNo="u1"]/mark
Q2	//department[.//course/field="www"]/dname
Q3	//student[hobby]/sname
Q4	//faculty[.//student/hobby="tennis"]/fname
Q5	//dblp/article[author]//year
Q6	let $t:=//inproceedings[.//title]/author return {$t}

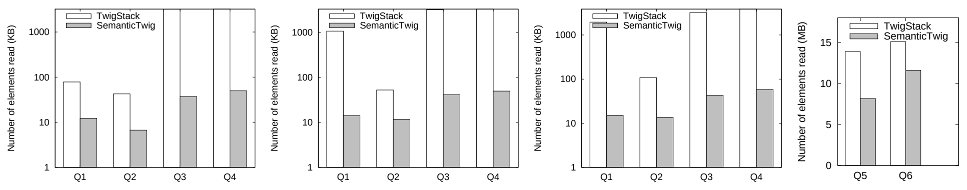
5.1 SemanticTwig VS TwigStack

Queries Q1-Q4 are chosen for synthetic datasets Doc1-Doc3, and Q5-Q6 are selected for DBLP dataset, as shown in Table 2. Q1 is used to test the effect of the first optimization method exploiting the *FD* {*course,student*}→*mark*, and Q2-Q6 are used to test the effect of the pure second optimization or the deployment of both optimizations. In particular, Q3 can exploit the role of multi-valued attribute *hobby* as existence check; Q2 and Q4-Q6 test the effects of query breakup technique.

SemanticTwig outperforms *TwigStack* on both synthetic and real datasets, as shown in Figure 6 and 7. We also compared it to *TJFast*, and find *SemanticTwig* outperforms *TJFast* in a similar fashion, due to its orthogonality to all structural-join based TPMs. The comparison is further analyzed in terms of the *cost of disk access*, *size of intermediate results* and *query running time.*

Cost of disk access As shown in Figure 6, *SemanticTwig* reads much fewer elements than *TwigStack* (at least two orders of magnitude, the y-axis's statistics is log-scaled). This huge gap results from the fact that *TwigStack* scans elements for all the query nodes, while *SemanticTwig* scans only the elements of matches contributing to distinct query results, so the elements of redundant matches are skipped. As Figure 6(b) and 6(c) show, *TwigStack* performs even worse for recursive document, while *SemanticTwig*'s I/O cost has small change. E.g. in evaluating Q2, within a certain *department*, once the first *course*-element *C* whose field is *"www"* is found, we can skip all descendants of *C* which have the same node type as *C*, i.e. all *courses* which are the pre-requisite of *C* can be skipped, since they contribute to the same output value as *C*; however *TwigStack* still loads them into memory. Figure 6(d) shows our approach reads less elements than *TwigStack* for DBLP.

Size of intermediate results Table 3(a) shows the number of intermediate path solutions generated by *TwigStack* and *SemanticTwig* for non-recursive



(a) Doc1 (b) Doc2 (c) Doc3 (d) DBLP

Fig. 6. TwigStack vs SemanticTwig on Number of Elements Read (log-scale)

*Doc*1. The 4*th* column is the minimal number of merge-joinable path contributing to distinct query answers. *TwigStack* outputs more path solutions than *SemanticTwig* for all queries except Q1 which has the same number of path solutions, because in Q1 a student has only one mark for each course he takes. TwigStack even generates much more partial solutions for a query on recursive XML document, shown in Table 3(b). The reduction of intermediate path solutions on Doc3 is similar to the result on Doc1, we do not show the table due to space limitation. The reduction is 32.8% for $Q5$ and 13.7% for $Q6$ on DBLP. Since each article often has no more than three authors, the reduction is not significant. This is because *SemanticTwig* only generates matches with distinct output result here, while multiple matches contributing to the same result are generated by *TwigStack*.

Table 3. Number of intermediate path solutions

(a) Doc1

Q	Twig	SemTwig	Useful	Reduction
Q1	3	3	3	0%
Q2	18	8	8	55.6%
Q3	213008	53252	53252	75%
Q4	76828	16	16	99.9%

(b) Doc2

Q	Twig	SemTwig	Useful	Reduction
Q1	66	3	3	95.5%
Q2	30	8	8	73.3%
Q3	213008	53252	53252	75.2%
Q4	76828	16	16	99.9%

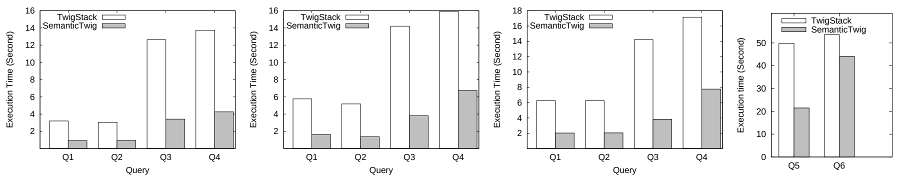
(c) DBLP

Q	Twig	SemTwig	Useful	Reduction
Q5	331997	223218	223218	32.8%
Q6	703819	607680	697680	13.7%

Query running time Figure 7 reports the total execution time on both synthetic and real datasets. In order to be fair, we do not include the time spent on post-eliminating redundant matches into the *TwigStack*'s total execution time. *SemanticTwig* is at least 4 times faster than *TwigStack*. The improvement is not significant on DBLP, due to the fact that DBLP is a shallow and wide document, and less redundant matches exist for $Q5$ and $Q6$. The improvement is much more significant if post-processing time is counted.

6 Conclusion and Future Work

In this paper, we aim to avoid finding redundant matches that return the same results by making use of the semantic information resided in XML document and the issued query. On one hand we utilize the semantics resided in



(a) Doc1 (b) Doc2 (c) Doc3 (d) DBLP

Fig. 7. TwigStack vs SemanticTwig on Execution Time

XML document to derive functional dependencies; on the other hand we explore the semantics of an XML query, distinguish its predicate and output nodes and initiate a query breakup technique. These two techniques can be deployed simultaneously in the same query. As a result we propose *SemanticTwig*, which is a novel semantics-aware query optimization algorithm. As part of future work, we want to investigate more semantics useful for efficient XML query processing.

References

1. S. Al-Khalifa, H. V. Jagadish, J. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–152, 2002.
2. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.
3. S. Chen, H. Li, J. Tatemura, W. Hsiung, D. Agrawal, and K. Selçuk Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over xml documents. In *VLDB*, pages 283–294, 2006.
4. 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.
5. P. Chippimolchai, V. Wuwongse, and C. Anutariya. Semantic query formulation and evaluation for xml databases. In *WISE*, pages 205–214, 2002.
6. P. Chippimolchai, V. Wuwongse, and C. Anutariya. Towards semantic query optimization for xml databases. In *ICDE Workshops*, 2005.
7. J. Clark and S. DeRose. Xml path language xpath version 1.0, 1999.
8. Haris Georgiadis and Vasilis Vassalos. Xpath on steroids: exploiting relational engines for xpath performance. In *SIGMOD*, pages 317–328, 2007.
9. Torsten Grust, Maurice Van Keulen, and Jens Teubner. Accelerating xpath evaluation in any rdbms. *ACM Trans. Database Syst.*, 29(1):91–131, 2004.
10. H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed xml documents. In *VLDB*, pages 273–284, 2003.
11. J. Lu, T. Chen, and T. Ling. Efficient processing of xml twig patterns with parent child edges: A look-ahead approach, 2004.
12. J. Lu, T. Ling, C. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of twig pattern matching. In *VLDB*, pages 193–204, 2005.
13. D. Florescu S. Boag, D. Chamberlin and J. Robie. Xquery 1.0: An xml query language, 2007.
14. Xiaoying Wu, Tok Wang Ling, Mong-Li Lee, and Gillian Dobbie. Designing semistructured databases using ora-ss model. In *WISE*, pages 171–180, 2001.
15. C. Zhang, J. Naughton, J. DeWitt, and Q. Luo andM. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436, 2001.