

Accomplishing Deterministic XML Query Optimization*

Dunren Che
Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA
dche@cs.siu.edu

Abstract

As the popularity of XML (eXtensible Markup Language) keeps growing rapidly, the management of XML compliant structured-document databases has become a very interesting and compelling research area. Query optimization for XML structured-documents stands out as one of the most challenging research issues in this area because of the much enlarged optimization (search) space, which is a consequence of the intrinsic complexity of the underlying data model of XML data. We therefore propose to apply deterministic transformations on query expressions to most aggressively prune the search space and fast achieve a sufficiently improved alternative (if not the optimal) for each incoming query expression. This idea is not just exciting but practically attainable. This paper first provides an overview of our optimization strategy, and then focuses on the key implementation issues of our rule-based transformation system for XML query optimization in a database environment. The performance results we obtained from experimentation show that our approach is a valid and effective one.

Keywords. XML query optimization, XML query, Query optimization, XML database, Structured-document database.

1. Introduction

Because of the rapidly growing popularity of XML, the management of XML structured-documents (or XML data in general) is now a very interesting and practical research issue. XML data is essentially semistructured and is distinct from conventional data, such as relational data, which has gained successful management functionality from RDBMS. It is almost a common consensus in the database research community that XML data should benefit

from the same/similar type of management functionalities as relational data gains from RDBMSs.

In recent years, various storage schemes have been proposed for XML data, e.g., mapping to relational or object-relational models [1, 2, 3, 4, 5, 6], or using special-purpose database systems such as semistructured databases [7]. Research for native XML database technology now seems to dominate the area. XML query optimization is a very challenging issue that faces the database research community. The tree based data model of XML is intrinsically more complex than the relational data model. This makes the search space for XML query optimization much larger than that for relational queries. The objective of our research is to develop novel, efficient approaches to XML query optimization. Toward this goal, an embryo of our deterministic optimization approach was first proposed in [8], with emphasis on using equivalence-based algebraic transformations to XML queries. The PAT algebra [9] was adopted and extended [10] for algebraic representation of XML queries and as the carrier of our approach. The PAT algebra can capture most of the constructs of XPath[11], the core sub-language of XQuery[12], although it was designed much earlier than XPath and XQuery. We identified a large number of equivalences and transformation rules [13, 14] used in our approach. The main theme in our approach is to exploit schema-based constraints, structural properties, and potential structure indices of XML data for query optimization. The most important feature of our approach is the *deterministic transformation*, which in our setting is not just an exciting idea but practically attainable. The goal of our approach is to quickly obtain a much improved alternative (if not the optimal) for each input query expression. This idea fits well with the dynamic nature of web-based applications, which usually ask for highly efficient online querying functionalities. In this paper, we put our emphasis on the major implementation issues of our deterministic optimization strategy for XML queries, which are represented as PAT algebraic expressions.

Related work. Lore [7, 15] is a DBMS originally de-

*Final version published by the *J. of Computer Sci. & Technol.*, Jan. 2005, Vol.20, No.1.

signed for semistructured data and later migrated to XML-based data model. Lore has a fully-implemented cost-based query optimizer that transforms a query into a logical query plan, and then explores the (exponential) space of possible physical plans looking for the one with the least estimated cost. In contrast to Lore, we perform purely logical level transformation targeting mainly at exploiting structure knowledge and structure indices for query optimization.

In [16], a strategy for exploiting a grammar specification for optimizing queries on semistructured data is studied; efforts were particularly made on how to make complete use of the available grammar to expand a given query. Our focus is different: we identify transformations that introduce improvements on query expressions in a very goal-oriented manner.

In [17], the minimization issue of XPath queries is discussed, and efforts were mainly given to the re-formulation of queries for avoiding redundant conditions, and a polynomial algorithm was developed for minimizing query patterns involving $/$, $//$, $[]$, and $*$. The mechanism used was to find a homomorphism from query q to p to identify the containment between the two queries (i.e., $p \subseteq q$).

In [18], the authors studied the minimization issue of general tree pattern queries, and both constraint dependent and constraint independent minimizations were addressed. They considered two types of constraints derived from XML schema or DTD (Document Type Definition), namely, required child/descendant (equivalent to what we called obligation, see Section 2.2) and co-occurrence of sibling subelements. The latter is not addressed in our work. However, we exploit two other types of structure knowledge, i.e., exclusivity and entrance location (see Section 2.2), which were not discussed in [18].

As containment is the basic means for deciding the equivalence and the minimization of XPath queries, it was studied in [19] under various DTD implied constraints, such as sibling constraints (same as co-occurrence constraints) and functional constraints (e.g., every a-node has at most one c-node as child). The results (i.e., the decidability and complexity of containment under various conditions) are not directly comparable to our work as containment itself is not XML query optimization.

Schema-based XPath query optimization was also addressed in [20], where several ideas proposed were similar to ours but a different approach was taken. In [20], the path equivalence class concept was put forward and used to (1) eliminate redundant path conditions; (2) shorten necessary paths; (3) identify unsatisfiability of a query at compile-time. Two different types of structural properties (i.e., co-occurrence and implication) were captured in this framework. While the optimality of an XPath query in [20] was with regard to non-redundant and shortest paths, we have to indicate that using a shorter path is good only when traver-

sal is the only means for evaluating the path. In contrast, we put a lot of efforts on identifying potential structure indices and improving the quality of a query by exploiting more extensive structural properties of XML data.

In summary, the contributions of our work consist of a novel algebraic transformation strategy, extensive exploration of potential indices and the structure knowledge of XML data, and other heuristics for XML query optimization.

Paper organization. The remainder of this paper is organized as follows. Section 2 provides preliminaries and sets context for subsequent discussion. Section 3 addresses the major implementation issues of our approach and presents the corresponding algorithms adopted in our implementation. Section 4 presents the transformations applied on two example queries. Section 5 shows the experimental results with regard to the optimization performance of our approach. Section 6 concludes the discussion of this paper.

2. Preliminaries

The main theme of this paper is on the key implementation issues of our approach to XML query optimization. To make the paper self-contained, we provide background knowledge in this section, which includes some DTD related notions, the PAT algebra, and an overview of our deterministic optimization strategy.

2.1. DTD related notions

Our work targets at XML compliant structured-documents. For a given class of documents, the legal markup tags (i.e., entity or element types) are defined in a DTD or an XML schema. DTD and schema play the same/similar function but DTD is simpler. So, for ease of presentation, we set forth our discussion in this paper based on the DTD notion, but our approach shall work equally well when a schema is used instead.

A DTD is the grammar for a class of documents and defines the legal element types with given names. Associated with each element type is a content model that describes the composition structure of the elements of the type. A content model can be defined using the following production:

$$c \rightarrow \langle etn \rangle \mid c_1, c_2 \mid c_1 | c_2 \mid c_1 \& c_2 \mid c_1 ? \mid c_1^* \mid c_1^+ \mid (c_1)$$

etn stands for element-type-name. The comma is called the sequence connector (SEQ) or a SEQ-node if the term is seen as a tree; “|” is the OR-connector (OR) or an OR-node; “?” is the optional occurrence indicator; “*” is the optional-and-repeatable occurrence indicator; and “+” indicates one or more repeatable occurrence of the preceding symbol.

The structural relationships among XML elements implied by a DTD are ideally illustrated by a graph, called

DTD graph. A formal definition for the notion is given below.

Definition 2.1 (DTD graph) *The DTD graph of a specific DTD is a directed graph $G = (V, K)$. Its vertices are the names of the element types from the DTD, and each element-type name occurs only once. An edge (ET_i, ET_j) in K indicates that ET_j occurs in the content model of ET_i . $RT \in V$ is the root element type of the DTD.*

If element type ET_j occurs in the content model of ET_i , we may say that ET_j is *internal* to ET_i and ET_i is *external* to ET_j .

The notion of *DTD graph* is useful in helping us visualize important structural relationships induced by a DTD, such as the *contains/contained-in* relationships among XML elements. A path in a DTD graph is another useful concept in our approach and is formally defined below.

Definition 2.2 (Path in a DTD graph) *A path in a DTD graph G , is a sequence of element types (ET_i, \dots, ET_j) s.t. ET_k directly contains ET_{k+1} , $i \leq k < j$.*

The reverse of a path is called a reverse path with regard to the original one. However, when not necessary, we will not particularly differentiate them but simply call them as paths.

2.2. The PAT algebra

An extended version [10] of the PAT algebra [9] has been adopted as the formalism in our approach. PAT is *set* oriented in the sense that each PAT algebra operator and each PAT expression evaluate to a *set* of XML elements. Herein we only focus on a restricted version of the extended PAT, which suffices to serve the purpose of this paper.

A query expression conforming to the PAT algebra is generated according to the following grammar:

$$E ::= etn \mid E1 \cup E2 \mid E1 \cap E2 \mid E1 - E2 \mid \sigma_r(E) \\ \mid \sigma_{A,r}(E) \mid E1 \subset E2 \mid E1 \supset E2 \mid (E)$$

“E” (as well as “E1” and “E2”) stands for a PAT expression; *etn* introduces an element type name; “r” is a regular expression representing a matching condition on the textual content of the elements determined by its input operand, which is a subexpression; and “A” additionally designates an attribute of the input elements.

\cup , \cap and $-$ are the standard set operators, union, intersection and difference. $\sigma_r(E)$ takes a set of elements and returns those whose content matches the regular expression r , while $\sigma_{A,r}(E)$ takes a set of elements and returns those whose value of attribute A matches the regular expression r . Operator \subset returns all elements of the first argument that are contained in an element of the second argument, while \supset returns all elements of the first argument that contain an element of the second argument.

In the sequel of our discussion, we will not intentionally distinguish element types from element type names when confusion is not anticipated. Furthermore, we may simply use *etn* to refer to an element type name.

The semantics of the PAT algebra can be more precisely characterized by two (partial) functions, **type**: $\mathcal{P} \rightarrow ETN$ and **ext**: $\mathcal{P} \rightarrow \mathcal{E}$, where \mathcal{P} is the set of all PAT expressions, ETN is the set of all element type names used in an XML database, and \mathcal{E} is the set of all XML elements in the database.

The following corollary holds for the PAT algebra:

Corollary 2.1 *Each expression, say E , evaluates to a set of XML elements of a single type, namely $type(E)$.*

Consequently, we will refer to the single result type of every PAT expression, E , by $type(E)$.

2.3. Strategy overview

In this subsection we provide a brief overview of our optimization strategy, which was detailed in [8, 13].

Ever since the beginning of this research, we envisioned a typical scenario of the use of our query optimizer – as a backend support for large XML document servers hooked on the web. Therefore, the *efficiency* of query optimization is of extreme importance. Furthermore, because the intrinsic data model of XML structured documents is complex, the underlying optimization algebra contains much more complicated operators as compared with the relational algebra. This complexity means that we have to crawl in a much enlarged search space for the optimal plans when we apply a transformation-based approach to the XML query optimization. Traditional heuristic-based and cost-based approaches are inappropriate in this scenario simply because we cannot afford the time needed by these time-consuming approaches while we are confronted with a very large search space. Therefore, we propose to pursue the heuristic-based approach at its extreme by applying exclusively *deterministic* transformations on logical query expressions. In other words, our optimizer does not produce nor evaluate different alternatives, but only runs after convinced improvement on input query expressions via each transformation step-by-step.

We have identified 46 *generic* equivalences of XML queries represented as PAT expressions (Imagine how big a search space this could mean!). These equivalences are *generic* because we introduced generic operation symbols for compactness of description, e.g., \supseteq stands for either \subset or \supset . Equivalences are not directly useful according to our optimization strategy because we do not generate and evaluate different alternatives. However, based upon these equivalences we derived 68 *generic* deterministic transformation

rules, which translate into 112 instantiated transformation rules.

To efficiently direct deterministic transformation, the whole optimization process in our system is organized into three transformation phases: *normalization*, *semantic transformation*, and a final *clean-up* phase. The second phase, semantic transformation, is the core in our approach. It accomplishes the most important class of transformations, i.e., introduces or enables applications of potential structure indices¹ or substantially simplifies query expressions by exploiting specific structural knowledge of XML data induced by the DTD.

Efficient exploitation of the structure knowledge about XML data is crucial to our approach. Structure knowledge is captured in our system through DTD graph analysis. A DTD typically covers a class of XML documents and is much smaller and more stable than the database itself. The time spent on DTD graph analysis for capturing the structure knowledge is easily paid off through amortization by repeated database querying.

Now we introduce three key notions that characterize three important types of structure knowledge about XML data, namely, *exclusivity*, *obligation*, and *entrance locations*. These three notions bear potential for query optimization.

Definition 2.3 (Exclusivity) *Element type ET_j is exclusively contained in element type ET_i if each path (e_j, \dots, e_k) with e_j being an element of type ET_j and e_k being the document root contains an element of type ET_i . Conversely, element type ET_i exclusively contains ET_j if the condition holds.*

If $type(E1)$ is exclusively contained in $type(E2)$, the containment selection condition imposed by the expression $E1 \subset E2$ is exempt for examination; thus the expression $E1 \subset E2$ can be simply rewritten as $E1$.

Definition 2.4 (Obligation) *Element type ET_i obligatorily contains element type ET_j if each element of type ET_i has to contain, in any document complying with the DTD, an element of type ET_j . Conversely, we say that ET_j is obligatorily contained in ET_i .*

The concept of obligation justifies the rewrite of $E1 \supset E2$ as a single $E1$ if $E1$ obligatorily contains $E2$.

Definition 2.5 (Entrance location) *Element type EL is an entrance location for $type(E1)$ and $type(E2)$ if, in any document complying with the given DTD, all paths from an element $e1$ of $type(E1)$ to an element $e2$ of $type(E2)$ pass through an element el of type EL .*

¹Although structure indices in our system are sophisticated, for understanding this paper, it suffices just to know that a structure index is a mapping between the extents of two element types; structure indices support fast evaluation of containment operations.

As special cases, $type(E1)$ and $type(E2)$ themselves are entrance locations for $type(E1)$ and $type(E2)$. The *entrance location* notion gives rise to a number of interesting equivalences. One example is as follows: $E1 \subset E2 \iff E1 \subset (E3 \subset E2)$ if $type(E3)$ is an entrance location for $type(E1)$ and $type(E2)$.

The primary use of the structural properties regarding XML documents captured by the above notions is to equivalently transform a query expression to a different form so that a potential structure index is used, which otherwise is impossible. Another usage of these notions is to help shorten the navigation paths that are imperative to the evaluation of a query expression in case there is no beneficial structure index available.

Five basic cases regarding the exploitation of a structure index for query optimization are illustrated in Figure 1 (more cases are detailed in [13]).

Case (1) reflects the two situations that a structure index is straightforwardly applicable.

Case (2) corresponds to rule $\mathcal{R}55$ (see Appendix) in our generic rule system, where input expression $E1 \subset E2$ is rewritten as $E1 \subset E3$ for enabling the application of a structure index between $type(E1)$ and $type(E3)$ (which is otherwise inapplicable) provided that the newly introduced element type $E3$ is an entrance location for $type(E1)$ and $type(E2)$ and is exclusively contained in $type(E2)$.

Case (3) corresponds to rule $\mathcal{R}56$ (see Appendix), where input expression $E1 \supset E2$ is rewritten as $E1 \supset E3$ for enabling the application of a structure index between $type(E1)$ and $type(E3)$ provided that $E3$ is an entrance location for $type(E1)$ and $type(E2)$ and obligatorily contains $type(E2)$.

Case (4) corresponds to rule $\mathcal{R}57$ (see Appendix), where input expression $E1 \subset E2$ is rewritten as $E1 \subset E3$ for enabling the application of a structure index between $type(E1)$ and $type(E3)$ provided that $E2$ is an entrance location for $type(E1)$ and the newly introduced element type $E3$, and is exclusively contained in type $E3$.

Case (5) corresponds to rule $\mathcal{R}58$ (see Appendix), where input expression $E1 \supset E2$ is rewritten as $E1 \supset E3$ for enabling the application of a structure index between $type(E1)$ and $type(E3)$ provided that $E2$ is an entrance location for $type(E1)$ and $type(E3)$ and obligatorily contains $type(E3)$.

With case (2) and (3), even if a potential structure index is not available, the performed transformations are still beneficial because the navigation paths involved are getting shorter. These two cases in such a situation render two other rules, $\mathcal{R}59$ and $\mathcal{R}60$ (see Appendix).

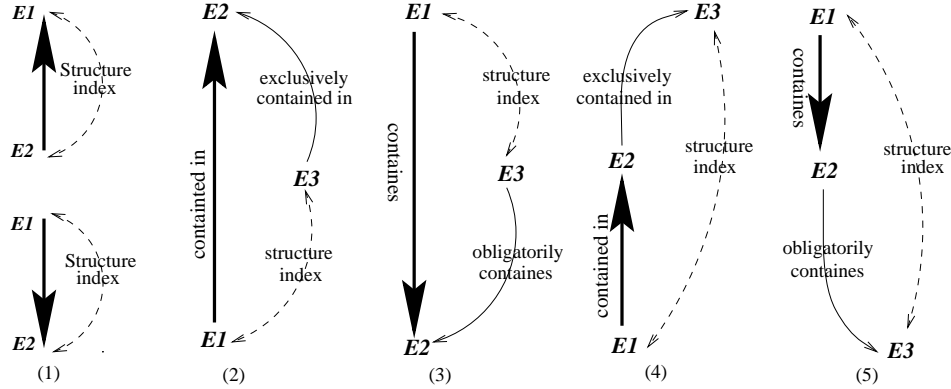


Figure 1. Transformations exploring structure indices

3. Algorithms

Now we are ready to address the interesting implementation issues of our deterministic transformation approach. As mentioned, the major goal of query transformation in our approach is to identify potential structure indices and to enable the application of the indices into query expressions, and this is achieved via controlled application of heuristic transformation rules. The majority of the rules pertinent to achieving the above goal are associated with a complex condition that jointly determines the applicability of a rule to an input expression. This condition is typically with regard to the availability of a relevant index and specific structure knowledge of XML data such as what being captured by the notions, *exclusivity*, *obligation*, and *entrance location*. In the following, we describe the algorithms that accomplish the deterministic transformations in our system and the supporting algorithms, which, for example, are used to identify the expected exclusivity, obligation and/or entrance location knowledge.

3.1. Transformation algorithms

As pointed out in Subsection 2.3, our strategy organizes the whole optimization process of a query as three consecutive transformation phases. A switch mechanism is used to turn on/off each rule depending on whether the rule makes sense to the current transformation phase. An important feature of our approach is the determinism of query transformations, which makes the implementation of query transformations simple and efficient. Because of the same feature, we do not concern the enumeration and screening of multiple alternatives. Rather, we always keep a single optimal alternative, which is the output of the transformation performed at each step. A high level description of the deterministic transformation strategy is given below.

```
optimize(input) := {
  return type: PAT_expression;
```

```
  enable all normalization rules
  (but disable all other rules);
  normalized = transform(input);
  enable all DTD semantic rules
  (but disable all other rules);
  improved = transform(normalized);
  enable all simplification rules
  (but disable all other rules);
  optimized = transform(improved);
  return optimized
}
```

```
transform(expr) := {
  return type: PAT_expression;
  for each argument arg of expr
    arg = transform(arg);
  while there are applicable rules in
  the currently enabled rule set
  {
    let r be the first of such rules;
    generate a new expression new_e
    by applying r to expr;
    result = transform(new_e);
  }
  /* if no more applicable rule */
  return result;
}
```

In our implementation, the switching mechanism of rules is implemented by a Boolean array. *Enable* and *disable* are the two operations needed for turning on/off the switch of a rule. In each specific transformation phase, only the rules in “on” state are considered. This is a simple but effective mechanism for putting the rule system under well control to achieve better transformation efficiency.

Obviously, the order of the rules in the rule system is significant. In case there are multiple applicable rules, our approach always chooses the first one as the unique one at that step to pursue a transformation. So the transformation performed is *deterministic*. The first rule applicable at each

transformation step is believed the most beneficial to invoke. The order of rules in our system is made based on relevant heuristics. When no more applicable rule can be found, our optimization process for that input query terminates. This last alternative expression reached during the linear transformation process is regarded the optimal one. As long as there are still applicable rules, the while loop in the transform algorithm does not stop.

3.2. Identifying exclusivity and obligation properties

In Subsection 2.3, we gave the definitions of the three important notions about the structural properties of XML data. These properties are practically identifiable from the DTD graph.

According to the definition of exclusivity, to examine, e.g., whether B is exclusively contained in A, our algorithm simply checks the paths between B and the root type in the DTD graph: if no path is found without the occurrence of A then the property holds.

In order to identify all cases of obligation, a deeper look at the content model of element types is indispensable. Otherwise, we cannot distinguish whether an element requires or just optionally contains a specific subelement. Thus, as the first step we eliminate all optional occurrence indicators from the content model of an element type. To do so, we introduce the following notion [21].

Definition 3.1 (Reduced version of a DTD) *Let D be a DTD. By taking all content models and removing all subexpressions whose root has an optional occurrence indicator (?) or an optional-and-repeatable occurrence indicator (*), we obtain a reduced DTD D' .*

The following proposition holds [21]:

Lemma 3.1 *The obligation properties within a DTD and its reduced version are identical.*

For examining the obligation property between element type A and B, we next flatten the content model of A so that the obligatory occurrence of B in A's content model eventually become obvious, and the result is called A's **flattened content model** for B, which is achieved through the following algorithm:

```

extend_content_model(A,B) := {
  return type: content model;
  let  $c_A$  be the content model of A;
  while ( $c_A$  contains non-terminal
    element types different from B)
  {
    let o be the occurrence of such an
    element type and let C be the

```

```

    element type;
    replace o in  $c_A$  with the content
    model of C;
  }
  return  $c_A$ ;
}

```

Base on the *flattened content model* notion, we produce a **normalized form** for the content model of element type A by performing the following steps, which were specified in detail in [21] by the former colleague of the author (the *flattened* model was called *extended* model there):

1. Recursively inline all child SEQ-nodes into their parent SEQ-nodes;
2. Recursively inline all child OR-nodes into their parent OR-nodes;
3. If any OR-node is not root, transform the content model so that the OR-node is relocated to the root;

These steps do not alter the content represented by a content tree but leads to a normal form that possesses the following properties:

- the root is an OR-node,
- the children of the root are SEQ-nodes,
- the children of SEQ-nodes are leaves, i.e., element types.

In the context of our approach, content model normalization is essential for revealing implicit obligation property, which is otherwise easily not to be identified [21, 13].

Finally, based on the normalized content model of A for B, we can conveniently identify the existence of the obligation property between A and B by examining obvious occurrences of B in A's content model.

As a summary, our algorithm for examining the obligation property between element type A and B is specified at a high level as below:

```

obligatorily_contains(A,B) := {
  return type: Boolean;
  step 1. Produce a reduced version
    of the DTD;
  step 2. Extend the content model of
    A for B;
  step 3. Generate normalized form
    for the content model of
    A;
  step 4. Search for occurrence of B
    in each SEQ-node of A's
    content model;
  Return true if found,
  otherwise false;
}

```

3.3. Identifying relevant entrance locations

Many important semantic transformation rules, e.g., $\mathcal{R}55$ to $\mathcal{R}60$ (see Appendix), rely on the *entrance location* notion. Therefore, finding all *relevant* entrance locations with regard to an input (sub-)expression is an essential implementation issue.

As depicted in Figure 1, the notion, “entrance location”, representing a requisite condition for the corresponding rules, is used to render profitable transformations in two different manners: introducing a new element type $E3$ (which must be an *entrance location* for $type(E1)$ and $type(E2)$) into a containment operation (i.e., case (2) and (3) in Figure 1), or making $E2$ itself an entrance location for $type(E1)$ and a newly introduced element type $E3$ (i.e., case (4) and (5) in Figure 1) and then replacing $E2$ with the new type $E3$.

For terminological uniformity, we generalize concept of *entrance location* so that, when the term is generally referred to, it covers both situations pointed out above.

Definition 3.2 (Entrance location generalization) *If $E3$ is an entrance location for $E1$ and $E2$ as defined in Definition 2.5, we call $E3$ an inset entrance location for $E1$ and $E2$, and call $E1$ (or $E2$) an outside entrance location for $E3$ and $E2$ (or $E1$).*

Therefore, in the remainder of this paper, the notion “entrance location” may refer to either an *inset entrance location*, applicable to case (2) and (3) of Figure 1, or an *outside entrance location*, applicable to case (4) and (5) of Figure 1.

The algorithm identifying inset entrance locations between a pair of element types heavily relies on exploring the DTD graph. For the sake of efficiency, we let each element type ET maintains (via pre-computation) all the paths that diverge from the type down to the leaves of the DTD graph and denote the set of such paths as $down_paths(ET)$. The following algorithm identifies all inset entrance locations for two given element types.

```

identify_Els(ET1, ET2) := {
  input: element type ET1 and ET2;
  return type: set of <etn>;
  return identify_Els(ET2, ET1) if
    ET2 is external to ET1;
  for each path maintained in
    down_paths(ET1),
    mark up those containing vertex ET2;
  let s_path be the shortest path among
    the marked-up ones w.r.t. the
    length from vertex "ET1" to "ET2";
  for each intermediate vertex ET
    contained in s_path,
    if it is also contained in all other

```

```

    paths marked up,
    collect it into the entrance
    location set el_set;
  return el_set;
}

```

It is evident from the algorithm that an entrance location EL is *relevant* for element type $ET1$ and $ET2$ if it falls on the shortest path between $ET1$ and $ET2$.

Identifying the outside entrance locations for element type $ET1$ and $ET2$ is relatively simple: we only need to concern the types that are related to either $ET1$ or $ET2$ via a structure index. For example, if EL is related to $ET1$ by a structure index, we only need to examine whether $ET2$ is an inset entrance location for $ET1$ and EL . In other words, we simply check every path from $ET1$ to EL to see if it contains a solid occurrence of $ET2$ on the path. We use the *outside entrance location* notion solely to bring a superficially unrelated structure index into a containment operation (outside entrance locations can not be used to shorten imperative navigation paths).

3.4. Determine optimal entrance locations

For a given expression of form $E1 \succeq E2$, algorithm $identify_ELs(ET1, ET2)$ identifies relevant inset entrance locations for $E1$ and $E2$. These entrance locations are identified for the purpose of invoking one of the semantic rules such as $\mathcal{R}55$ to $\mathcal{R}60$ with a (sub-)expression of form $E1 \succeq E2$. But if the additional conditions of these rules do not hold with respect to the identified entrance locations, the rule can not be applied. The identified entrance locations, although relevant, are no longer interesting. So we need to further identify those really interesting ones from all identified relevant entrance locations. After that, if multiple interesting entrance locations exist, we proceed further to choose the most profitable one, i.e., the optimal entrance location, to successfully activate a corresponding semantic rule because our deterministic transformation strategy asks to always apply the most beneficial rule to a query expression whenever there are multiple rules applicable. Therefore, we need certain criteria to determine which one is the optimal entrance location for a given (sub-)expression of form $E1 \succeq E2$.

Definition 3.3 (Optimal entrance location) *If multiple relevant entrance locations, of which each may cause a separate application of one of the rules $\mathcal{R}55$ through $\mathcal{R}60$ (see Appendix) to an expression of form $E1 \succeq E2$, exist, then the optimal entrance location is determined according to the following criteria:*

- (1). *When a structure index can be exploited (this case corresponding to $\mathcal{R}55$ through $\mathcal{R}58$), the optimal entrance location is the one that has the smallest cardinality of its extent.*

- (2). When using a structure index is not possible (this case corresponds to $\mathcal{R}59$ and $\mathcal{R}60$), the optimal is the one that results in the shortest navigation path, i.e., the path from $E1$ to the entrance location in the DTD graph is the shortest.

Now we give the steps of another algorithm that determines the optimal entrance location for a given query subexpression of form $E1 \supseteq E2$.

```

identify_optimal_EL(E1,E2) := {
  return type: element type name;
  Step 1. Find relevant inset entrance
    locations;
  Step 2. Collect all interesting
    entrance locations;
  Step 3. Determine and return the
    optimal entrance location;
}

```

The details of each of the steps are addressed below:

Step 1: Find entrance locations

All relevant inset entrance locations for element type $E1$ and $E2$ are found (if exist) by calling function $identify_ELs(E1, E2)$; the result is a set of element type names, denoted as $rELs(E1, E2)$. All relevant outside entrance locations can be found in the way as discussed in the last paragraph of Subsection 3.3.

Step 2: Collect interesting entrance locations

Assume, beside the diverging paths, each element type t maintains two transitive closures, say, $excl_C^*(t)$ and $obli_C^*(t)$, which are the transitive closure of the relationship “*exclusively contained in*” and the relationship “*obligatorily contains*” on this type t , respectively. The closures can be computed by the methods given earlier in this section. In addition, each element type t maintains a set of interesting *etn*’s, annotated as $indices(t)$, which relate to the type t through a structure index.

For identifying interesting entrance locations, the following two cases need to be treated separately:

(Case 1: for $\mathcal{R}55, \mathcal{R}56, \mathcal{R}59, \mathcal{R}60$) check each inset entrance location for $E1$ and $E2$ obtained from last step to see if it, say $E3$, is contained in $excl_C^*(E2)$ or in $obli_C^*(E2)$. Next, check whether $E3$ is a member of $indices(E1)$. If so, mark it as “structure index defined”. For instance, the output of this step for Rule $\mathcal{R}55$ is $rELs(E1, E2) \cap excl_C^*(E2) \cap indices(E1)$, for Rule $\mathcal{R}56$ is $rELs(E1, E2) \cap obli_C^*(E2) \cap indices(E1)$, for Rule $\mathcal{R}59$ is $rELs(E1, E2) \cap excl_C^*(E2)$, and for Rule $\mathcal{R}60$ is $rELs(E1, E3) \cap obli_C^*(E2)$.

(Case 2: for $\mathcal{R}57$ and $\mathcal{R}58$) if $indices(E1)$ is not empty, for each $E3 \in indices(E1)$ check whether $E3$ is

an outside entrance location for $E1$ and $E2$ (i.e., $E2$ is an inset entrance location for $E1$ and $E3$), and whether $E2$ belongs to $excl_C^*(E3)$ or $E2$ belongs to $obli_C^*(E3)$. Mark the element types that satisfy these conditions as “structure index defined”, then output.

Step 3: Determine the optimal entrance location

This step differentiates the output of last step as either “structure index defined” (thus the index will be applied) or “no structure index defined” (thus the identified entrance locations will be used solely to shorten navigation paths involved):

(Case 1) For the entrance locations identified by Step 2 and marked as “structure index defined”, output the one that holds the smallest cardinality. If none is found, continue with (Case 2).

(Case 2) For the entrance locations identified by Step 2 and marked as “no structure index defined”, select the one that renders the shortest navigation path to reach $type(E1)$ in the DTD graph.

The last step of the above algorithm output the optimal entrance location that actually helps decide which of the six basic semantic rules is the most profitable one and thus should be exploited next to conduct a beneficial transformation on the incoming (sub-)expression of form $E1 \supseteq E2$.

4. Examples

In the following we use two example queries to illustrate the way our approach applies deterministic transformations to query expressions to achieve logical optimization. The operation $I_{Paragraph}(Article)$ that will be applied in the following transformations stands for the structure index defined between type *Article* and *Paragraph*, returning elements of type *Paragraph*. Since we did not introduce all the transformation rules used in these examples due to the theme of this paper and the space limitation, we do not expect readers’ sharp understanding of the details of the examples. Instead, our purpose here is just to show a little flavor of our deterministic transformation approach.

Query 1. Find the paragraphs of the “Introduction” section of each article that has the words “Data warehousing” in its title.

This query is formulated as a PAT expression and goes through the following transformations stepwise:

$$\begin{aligned}
 & (Paragraph \subset (\sigma_{A=Title, r='Introduction'}(Section) \\
 & \quad \subset (Article \supset \sigma_{r='Data\ warehousing'}(Title)))) \\
 \implies & (\subset \text{associativity}) \\
 & ((Paragraph \subset \sigma_{A=Title, r='Introduction'}(Section))
 \end{aligned}$$

$$\begin{aligned}
& \subset (Article \supset \sigma_{r='Data\ warehousing'(Title)}) \\
\Rightarrow & (\subset \text{ commutativity}) \\
& ((Paragraph \subset (Article \supset \\
& \sigma_{r='Data\ warehousing'(Title)}) \subset \\
& \sigma_{A=Title,r='Introduction'(Section)}) \\
\Rightarrow & (\text{Index application}) \\
& ((I_{Paragraph}(Article \supset \\
& \sigma_{r='Data\ warehousing'(Title)}) \cap Paragraph) \\
& \subset \sigma_{A=Title,r='Introduction'(Section)}) \\
\Rightarrow & (\cap \text{ deletion}) \\
& ((I_{Paragraph}(Article \\
& \supset \sigma_{r='Data\ warehousing'(Title)}) \\
& \subset \sigma_{A=Title,r='Introduction'(Section)})
\end{aligned}$$

In the optimized format, subexpression $(Article \supset \sigma_{r='Data\ warehousing'(Title)})$ and $\sigma_{A=Title,r='Introduction'(Section)}$ remain unchanged. The second \subset operation is replaced by the new index operation $I_{Paragraph}$, which is supposedly more efficient. The first \subset operation in the original format is still retained but now the extent of its new left operand, $I_{Paragraph}(Article \supset \sigma_{r='Data\ warehousing'(Title)})$, is much smaller than the original operand $Paragraph$; although, now the right operand's extent may be a little bigger, considering in practice the extent of type $Section$ is usually much smaller than the type $Paragraph$, the cost saving with this operation is still significant. The total effect of the optimization achieved from the above transformations is evident.

Query 2. Find all ‘‘Summary’’ paragraphs of all sections (if any).

This query is reformulated according to the PAT algebra and the following transformations are applied (We assume there is no structure index defined between $Paragraph$ and $Section$ but there is one between $Paragraph$ and $Article$):

$$\begin{aligned}
& (\sigma_{A=Title,r='Summary'(Paragraph)} \subset Section) \\
\Rightarrow & (\text{by } \mathcal{R}57: \text{ index enabling}) \\
& (\sigma_{A=Title,r='Summary'(Paragraph)} \subset Article) \\
\Rightarrow & (\text{Index application}) \\
& (I_{Paragraph}(Article) \cap \\
& \sigma_{A=Title,r='Summary'(Paragraph)}) \\
\Rightarrow & (\cap \text{ deletion}) \\
& (\sigma_{A=Title,r='Summary'(I_{Paragraph}(Article))})
\end{aligned}$$

With a structure index, $I_{Paragraph}(Article)$, being introduced, which replaces the \subset in the original query expression, the evaluation of the query is supposed to be much faster.

5 Performance Result

This section presents the preliminary result of our experiments with the prototype optimizer implemented based

on the approach and techniques presented in this paper. As our interest is focused on the performance of our approach, which does not pre-assume any particular storage model, we simply chose the Oracle RDBMS as platform for storage management to quickly build a test-bed for our experiments.

The test-bed is implemented in Java. The statistic data we gained from our experiments shows that our approach is indeed a valid one – in most cases it produces much better (logical) query plans that run 7 times faster by average than the non-optimized counterparts.

The experiments were carried out in a client/server database environment. The client side runs Oracle SQL*Plus (Release 8.1.6.0.0) on Window XP Pro., and the CPU is a Celeron processor of 500MHZ with a 192MB RAM. The server side runs an Oracle8i Enterprise Edition Release 8.1.6.0.0 database server, installed on Sun Ultra 5 with an UltraSparc-II CPU of 333MHZ and a RAM of 512MB. In our test-bed, a user query (internally represented as a PAT expression) is optimized on the client machine and the result is translated into an equivalent SQL query, which is then sent to the Oracle database server for execution.

As the underlying storage model adopted by our test-bed is a relational one, mapping of XML data to relational schema is imperative with this setting. We used a straightforward mapping scheme that maps each element type in a DTD to a separate relation. Within a relation, a unique ID is assigned to each element, which also holds a foreign key to its parent relation. The mapping information and relevant statistics are stored in a system catalog table, which provides meta-information needed for the optimization of PAT query expressions and subsequent translation to SQL format.

Two different synthetically-generated databases have been used for the experiments. The first database, a proceedings database (called Database 1), is generated based on the proceedings DTD [13], which is a fairly practical one and can be used for real proceedings. There are totally 178000 data elements populated in the database. The second database, called Database 2, is created according to a DTD that was intentionally designed to have relatively long paths. Database 2 was populated with a total of 189002 elements.

Performance Result with Database 1. This test includes the six queries used in [13] as running examples, including the two queries introduced in the last section. For each of these queries, we repeatedly run 10 times both the original (non-optimized) query and the optimized one and record the average elapsed time (in seconds). The result is shown in Table 1, and the comparison is charted in Figure 2.

Performance Result with Database 2. With database 2, we tested 10 groups of queries involving various lengths of

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6
Non-opt	0.722	0.796	2.601	3.414	1.285	4.590
Opt	0.722	0.796	1.553	0.366	0.797	0.584
Ratio	1.000	1.000	1.675	9.378	1.578	7.860

Table 1. Performance data with database 1

	G 1	G 2	G 3	G 4	G 5	G 6	G 7	G 8	G 9	G 10
Non-opt	2.521	5.396	12.517	2.405	4.386	2.231	2.486	3.583	12.569	6.555
Opt	0.278	0.468	2.518	0.325	0.500	0.278	0.307	0.515	2.445	0.528
Ratio	9.069	11.53	4.970	7.400	8.772	8.025	8.098	6.957	5.141	12.415

Table 2. Performance data with database 2

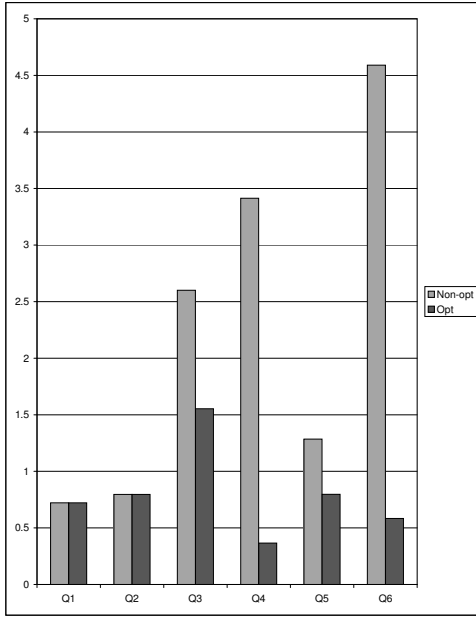


Figure 2. Performance result with database 1

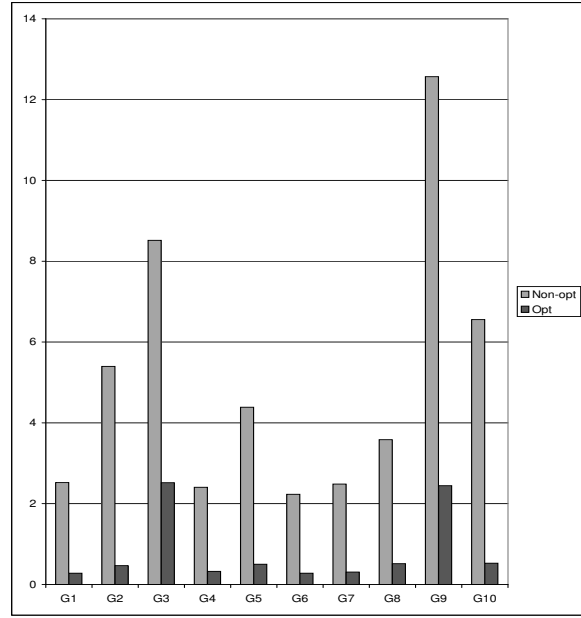


Figure 3. Performance result with database 2

paths. The longest path tested contains 9 element types and the shortest path covers just 3. We run each query 10 times and record the average elapsed time of each group with respect to the optimized queries and their non-optimized counterparts. The performance result is shown in Table 2, and the comparison is charted in Figure 3.

In our prototype implementation, paths are implemented as consecutive relational joins; and structure indexes are simulated by binary relations, which is obviously not an efficient way because each application of a structure index causes two extra relational join operations. If a more efficient implementation, e.g., a hashing scheme, is used for the structure indices, we shall expect further better performance result. In our current implementation, most queries can be optimized in a 2 or 3 seconds. We are satisfied with both the efficiency and the effectiveness of the implemented pro-

totype optimizer based on our approach and methods presented in this paper.

6. Summary

Query optimization has been always an exciting and challenging research issue in the database area. It is even more challenging in the context of XML databases because of the intrinsic complexity of XML data model, which causes a much enlarged search space for the optimization of XML queries. Driven by the request of modern web-based applications for highly efficient query processing, we proposed a novel optimization approach, called *deterministic optimization*. The basic idea of the approach is to apply exclusively deterministic transformations on XML queries to achieve the best possible optimization efficiency. To this

end, extensive heuristics and particular knowledge about the structure of XML data are explored to (1) significantly simplify XML query expressions, (2) introduce beneficial applications of potential structure indices into the queries, (3) or shorten the imperative paths involved in the queries. In this paper, our emphasis was given to the major implementation issues of our approach. We presented numerous algorithms which are crucial to our approach. We also presented preliminary experimental results, which additionally show the validity of our approach. As future work, we plan to do further in-depth experiments, and formally study the complexity of our approach. Lastly, it is interesting to study how the presented approach may be integrated into the context where a cost-based approach is applied to get further better optimization results.

Acknowledgement. The author would like to show great appreciation to his former colleagues, Karl Aberer, M. Tamer Özsu, and Klemens Böhm, etc., at Fraunhofer-IPSI (formerly known as GMD-IPSI), Germany. The work reported in this paper is the result of a continued effort on the research for structured-document database management, which was initiated by the author with the former colleagues at GMD-IPSI.

References

- [1] M. F. Fernandez, W. C. Tan, D. Suciu. SilkRoute: trading between relations and XML. *Computer Networks* 33(1-6), 2000, pp.723-745.
- [2] D. Florescu, and D. Kossmann. Storing and Querying XML Data Using an RDMBS. *IEEE Data Engineering Bulletin* 22(3), 1999, pp.27-34.
- [3] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proc. of VLDB*, 1999, pp.302-314.
- [4] P. Bohannon, J. Freire, P. Roy, J. Simeon. From XML Schema To Relations: A Cost-Based Approach to XML Storage. *Proc. of the 18th Int. Conf. on Data Engineering (ICDE)*, 2002, pp.64-75.
- [5] M. Klettke, H. Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. *Proc. of Int. Workshop on the Web and Databases (WebDB)*, Dallas, May 2000, pp.63-68.
- [6] B. Surjanto, N. Ritter, H. Loeser. XML Content Management based on Object-Relational Database Technology. *Proc. Of the 1st Int. Conf. On Web Information Systems Engineering (WISE)*, Hong Kong, June 2000, pp.70-79.
- [7] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record*, 26(3), Sep. 1997, pp.54-66.
- [8] D. Che and K. Aberer. A Heuristics-Based Approach to Query Optimization in Structured Document Databases. *Proc. of 1999 Int. Database Engineering & Application Symposium*, Montreal, Canada, Aug. 2-4, 1999, pp.24-33.
- [9] A. Salminen and F. W. Tompa. PAT Expressions: an Algebra for Text Search. *Acta Linguistica Hungarica*, 41(1994), no.1, pp.277-306.
- [10] K. Böhm, K. Aberer, E. J. Neuhold, and X. Yang. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. *The VLDB Journal*, Vol. 6, No. 4, Nov. 1997, pp.296-311.
- [11] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0 (<http://www.w3.org/TR/1999/REC-xpath-19991116>).
- [12] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language (<http://www.w3.org/TR/2004/WD-xquery-20040723/>).
- [13] D. Che, K. Aberer, and M. T. Özsu. Query Optimization in XML Structured-Document Database Systems (Manuscript in preparation for The VLDB Journal).
- [14] D. Che. Implementation Issues of a Deterministic Transformation System for Structured Document Query Optimization. *Proc. of 2003 Int. Database Engineering & Application Symposium*, Hong Kong, July 16-18-4, 2003, pp.268-277.
- [15] J. McHugh and J. Widom. Query Optimization for XML. *Proc. of VLDB*, Edinburgh, Scotland, Sep. 1999, pp.315-326.
- [16] M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. *Proc. of the 14th Int. Conf. on Data Engineering*, Orlando, Florida, USA, Feb. 23-27, 1998, pp.14-23.
- [17] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of Xpath queries. *Proc. of VLDB*, 2003, pp.153-164.
- [18] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. *Proc. of ACM Conf. on Management of Data (SIGMOD)*, 2001, pp.497-508.

- [19] P. T. Wood. Containment for XPath fragments under DTD constraints. *Proc. of 9th Int. Conf. of Database Theory*, Jan. 2003, pp.300-314.
- [20] A. Kwong and M. Gertz. Schema-based optimization of XPath expressions. Technical report, Univ. of California, dept. of Computer Science, 2001.
- [21] K. Böhm, K. Aberer, M. T. Özsu, and K. Gayer. Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition. *Proc. of IEEE Int. Forum on Research and Technology Advances in Digital Libraries (ADL'98)*, Santa Barbara, California, April 22-24, 1998, pp.196-205.

Appendix

R55. $(E1 \subset E2) \implies (E1 \subset E3)$ **if** $type(E3)$ is an entrance location for $type(E1)$ and $type(E2)$ and is exclusively contained in $type(E2)$, and $free(E2)$ holds, in addition, a structure index between $type(E3)$ and $type(E1)$ is available.

R56. $(E1 \supset E2) \implies (E1 \supset E3)$ **if** $type(E3)$ is an entrance location for $type(E1)$ and $type(E2)$ and obligatorily contains $type(E2)$, and $free(E2)$ holds, in addition, a structure index between $type(E3)$ and $type(E1)$ is available.

R57. $(E1 \subset E2) \implies (E1 \subset E3)$ **if** $type(E2)$ is an entrance location for $type(E1)$ and $type(E3)$ and is exclusively contained in $type(E3)$, and $free(E2)$ holds, in addition, a structure index between $type(E3)$ and $type(E1)$ is available.

R58. $(E1 \supset E2) \implies (E1 \supset E3)$ **if** $type(E2)$ is an entrance location for $type(E1)$ and $type(E3)$ and obligatorily contains $type(E3)$, and $free(E2)$ holds, in addition, a structure index between $type(E3)$ and $type(E1)$ is available.

R59. $(E1 \subset E2) \implies (E1 \subset E3)$ **if** $type(E3)$ is an entrance location for $type(E1)$ and $type(E2)$ and is exclusively contained in $type(E2)$, and $free(E2)$ holds.

R60. $(E1 \supset E2) \implies (E1 \supset E3)$ **if** $type(E3)$ is an entrance location for $type(E1)$ and $type(E2)$ and obligatorily contains $type(E2)$, and $free(E2)$ holds.

Note. In the above rules, the condition $free(Ei)$ requests that the evaluation of the expression Ei returns the full extent of $type(Ei)$, i.e., all elements of the type in the database. One typical example of free expressions is a plain element type name, etn , meaning that all elements of type etn are to be returned.

$\mathcal{R}59$ has almost the same condition as $\mathcal{R}55$ except for an available structure index. $\mathcal{R}55$ is assigned a higher priority (with a smaller ID#) and is tried first during query optimization for enabling an application of a potential structure index. Analogously for $\mathcal{R}56$ and $\mathcal{R}60$.