Implementation Issues of a Deterministic Transformation System for Structured Document 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 keeps growing rapidly, XML compliant structured document management becomes an interesting and compelling research area. Query optimization for structured documents stands out as a very challenging issue because of the much enlarged optimization (search) space, which is a consequence of the intrinsic complexity of the underlying data model of structured documents. We therefore propose to apply deterministic transformations on query expressions to most aggressively control 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 implementation issues of our transformation system for structured document query optimization.

1. Introduction

Because of the rapidly growing popularity of XML, the management of structured documents, especially XML compliant structured documents, is now a very interesting and practical research issue. XML data is essentially semistructured and distinct from conventional data, e.g., relational data and object-oriented data, which gained successful management functionality from RDBMS and OODBMS. It has been strongly believed that XML document/data should benefit from the same type of management functionalities as conventional data benefits from RDBMS/OODBMS.

In recent years, almost all kinds of storage schemes have been proposed, e.g., mapping XML data to relational [13, 14, 31, 4] or object-relational models [19, 33], using special-purpose databases such as semistructured databases [23], or even native XML databases. Query processing is yet another, probably more challenging issue because of the much enlarged optimization space. With regard to query processing for structured documents, a lot of work has been done on developing efficient processing algorithms [12, 23, 16, 22, 18, 10, 32] and appropriate indexing schemes [22, 5, 25, 17, 6]. Relatively, only a little work has been reported that focuses on specialized techniques for structured-document query optimization.

In [8], we proposed to use equivalence-based algebraic transformations on XML query optimization. We have identified a large number of equivalences and transformation rules [8, 9] exploiting the DTD constraints, the structural properties of XML documents, and potential indices. One striking feature of our approach is *deterministic* transformation, which in our setting is not just an exciting idea but practically attainable. Our goal 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-hooked applications, which are the major applications now asking for highly efficient document management and querying functionalities. This paper mainly focuses on the implementation issues of our deterministic transformation strategy for query optimization in a structured document database environment.

Related work. With regard to structured document database management, a lot of work has been done focusing on storage modeling, e.g., [1, 29, 28, 20, 34, 11, 7, 26, 27, 15, 13, 14, 31, 4, 19, 33], indexing [22, 5, 25, 17, 6],and algorithms [12, 23, 16, 22, 18, 10, 32] for supporting regular path processing. To the best of our knowledge, there is no reported work that is really closely related with ours, i.e., focuses on algebraic transformation for query optimization by exploiting DTD knowledge, structural properties of documents, and other heuristics. In the following, we briefly review some work that might be conceived as generally related.

Lore [23, 24] is a DBMS originally designed 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 least estimated cost. Lore is well known by its DataGuide path index that together with stored statistics describing the "shape" of the database provides the structural knowledge about the data to help Lore's optimizer prune its search space for a better plan. In this sense, Lore is related to our work, but we capture the structural knowledge of document data mainly from its DTD and apply this knowledge to conduct exclusively deterministic transformations on query expressions.

We are aware of another piece of work reported that deals with query optimization by using the document type definition knowledge. In [11], Consens and Milo replace a query-algebra operator with a cheaper one whenever the DTD allows that. The DTDs considered in that study are simpler than the ones of SGML/XML, and in contrast to our work the authors do not look at the different grammar constructors. The optimization in [11] also makes use of a special cost model and the results are thus not directly transferable to our application scenario. In contrast, the contribution of our work is the independence of a specific data model and of a specific cost model. Our approach is strongly heuristics-based.

In [12] a comparable strategy for exploiting a grammar specification for optimizing queries on semistructured data is studied in a similar setting – where efforts were 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. Other generally related work includes [28, 20, 34, 11, 7, 26, 27].

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 and presents corresponding algorithms adopted in our system. Section 4 shows a little flavor of our deterministic transformation by giving a few transformation examples. Section 5 concludes the discussion of this paper.

2. Preliminaries

This paper addresses the implementation issues of our structured document query optimizer. To make this paper self-contained, we provide this section for introducing background knowledge, including notions about DTD, our optimization algebra - the PAT algebra, an overview of our deterministic optimization strategy, and the six key semantic transformation rules.

2.1. DTD notions

The documents we target at in our system are SGML/XML compliant structured documents. For a given class of documents, the legal markup tags (i.e., entity or element types) are defined in document type definition (DTD). Thus a DTD is the grammar for a class of documents. The content model of each entity describes the composition structure of that entity and is normally defined using the following production:

$c \text{ -> } \text{<etn>} \mid c_1, c_2 \mid c_1|c_2 \mid c_1\&c_2 \mid c_1? \mid c_1^* \mid c_1^+ \mid (c_1)$

etn stands for element-type-name. Using SGML terminology, the comma is 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.

A DTD is usually a graph structure, which plays an import role in our implementation. So, we give a more formal definition of it 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 etn 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 *internal* and *external* terminologies here introduced understandably extend to the instance level of the element types as well.

By means of a *DTD graph*, we can visualize some important relationships induced by a DTD, such as the *contained-in/contains* relationships among document elements. The general containment relationship (either directly or indirectly) between document elements is actually determined by the content models of the corresponding element types in the DTD.

A path in a DTD graph is another important notion used in our system, and is defined as below.

Definition 2.2 (Path in a DTD graph) *A path in a DTD graph G, is a sequence of element types $(ET_i, ..., ET_j)$ s.t. ET_k directly contains ET_{k+1}, $i \leq k < j$. The reverse of this sequence is called a reverse path with regard to the original one.*

In the sequel, when not necessary we may not need to differentiate a reverse path from an ordinary path and refer to either one between two element types E1 and E2 as $path(E1, E2)$.

2.2. The PAT algebra

Our optimization strategy is based on the PAT algebra, which is originally designed as an algebra for searching structured documents [30]. We adopted it as our optimization algebra and extended it according to the features of SGML/XML compliant documents. The PAT algebra is *set* oriented, in the sense that each PAT algebra operator and each PAT expression evaluate to a *set* of document elements. A complete version of the extended PAT algebra has been described in [2]. Herein we focus on only a restricted version, which is sufficient 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 "E1" and "E2") generally stands for a PAT expression, etn introduces a document element type name, "r" is a regular expression representing a matching condition on the textual content of the document elements, and "A" designates an attribute of the document 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 remainder of this paper, we will not intentionally distinguish an element type from an element type name when confusion is not anticipated. Furthermore, we may simply use etn to refer to an element type name for compactness.

More precisely, the semantics of the PAT algebra can be given by using two (partial) functions, $type : \mathcal{P} \rightarrow ETN$ and $ext : \mathcal{P} \rightarrow \mathcal{E}$, where $\mathcal{P}$ is the set of PAT expressions, ETN is the set of element types names (and thus element types) in a document database, and $\mathcal{E}$ is the set of all elements in the document database.

The following corollary holds for our PAT algebra:

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

Consequently, we will refer to the result type of a PAT expression, e.g., E, by $type(E)$.

2.3. Stategy overview

In this subsection we provide a brief overview of our optimization strategy, which is detailed in [8, 9].

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 document servers hooked on the web. Therefore, the *efficiency* of the query optimizer itself is of extreme importance. Furthermore, because the intrinsic data model of structured documents is complex, the underlying optimization algebra contains many operators as compared with the relational algebra. This complexity means that we have to crawl in a much enlarged search space for optimal plans when we apply a transformation-based approach to the query optimization. Traditional heuristics-based and cost-based approaches are inappropriate in this scenario simply because we cannot afford the time needed by these time-consuming approaches when confronted with a very big search space. Therefore, we proposed to pursue the heuristics-based approach at its extreme by applying exclusively *deterministic* transformations. In other words, our optimizer does not produce and evaluate different alternatives, but runs after only convinced improvement stepwise on input query expressions via each transformation performed.

We have identified totally 46 generic equivalences, including set-oriented algebraic equivalences and document-structure based semantic equivalences. These equivalences are *generic* because we introduced generic operation symbols for compactness of description, e.g., $\overset{\supset}{\subset}$ stands for either $\subset$ or $\supset$. Equivalences are not directly useful according to our strategy because we do not evaluate different alternatives. However, based upon these equivalences we derived 62 *generic* deterministic transformation rules, which translate into 98 instantiated transformation rules.

To efficiently direct deterministic transformation, the whole optimization process in our system is organized as three transformation phases: *normalization*, *semantic transformation* (mainly to enable index application or shorten navigation paths), and *cleaning-up* (or simplification).

Most of the implementation issues we are going to address in the remainder of this paper are related to either enabling an application of a structure index[1] that is superficially inapplicable or to shorten a navigation path involved in a query expression. Most of this type of rules are based on exploitation of document structure knowledge that is captured in our system through DTD graph analysis. A DTD normally covers a class of documents and is much smaller and more stable than the database itself. The time spent on DTD graph analysis for gathering the structure knowledge is easily paid off through amortization by repeated database querying.

Now we introduce three important notions regarding

[1] Although structure indices in our system are sophisticated, but for understanding of the subsequent discussion, suffice it to know that a structure index between two element types simply provides a short-cut between the elements of the two types so that a (long) path navigation can be avoided.

DTD graph, *exclusivity*, *obligation*, and *entrance locations* between element types, which bear potential for query optimization.

Definition 2.3 (Exclusivity) *Element type ET_j is exclusively contained in element type ET_i if each path $(e_j, \ldots, e_k)$ 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 predicate imposed by the expression $E1 \subset E2$ is exempt for examination, hence $E1 \subset E2$ 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 $E1$ 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.*

As special cases, $type(E1)$ and $type(E2)$ themselves are entrance locations for $type(E1)$ and $type(E2)$. The notion of *entrance location* gives rise to the following equivalences. $E1 \subset E2 \Longleftrightarrow E1 \subset (E3 \subset E2)$ if $type(E3)$ is an entrance location for $type(E1)$ and $type(E2)$.

The primary use of above properties regarding DTD structure is to equivalently transform a query expression to a different form so that a potential structure index can be used, which otherwise is impossible. Another usage of these notions is to help shorten the navigation paths involved in a query expression in case of no beneficial structure index available.

Five different cases concerning exploitation of structure indices for query optimization have been identified as illustrated in Figure 1.

In case (1), the structure index is straightforwardly applied.

Case (2) corresponds to rule $\mathcal{R}55$ 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 E1 and E3 (which is otherwise inapplicable) provided that the newly introduced element type E3 is an entrance location for E1 and E2 and is exclusively contained in E2.

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

Case (4) corresponds to rule $\mathcal{R}57$ 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 E1 and E3 provided that E2 is an entrance location for E1 and the newly introduced element type E3 and is exclusively contained in E3.

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

For case (2) and (3), even if a potential structure index is not available, the transformation is still beneficial because it helps shorten the original navigation path, supposing $path(E1, E3)$ is sufficiently shorter than the original $path(E1, E2)$. The two cases under this situation account for the other two rules, $\mathcal{R}59$ and $\mathcal{R}60$, in our system, respectively.

The above mentioned six semantic rules, i.e., $\mathcal{R}55$ through $\mathcal{R}60$, are furnished to this paper as appendix.

3. Algorithms

Now we are ready to address the interesting implementation issues of our deterministic transformation system. One major goal of query transformation is to identify potential structure indices of XML documents and enable the application of these indices to input query expressions. The majority of these index-related rules come with a complex condition, which is based on the availability of relevant indices and special knowledge about the structure of the source documents such as those indicated by the notion of exclusivity, obligation, and entrance location. The condition of a rule and the pattern of a given query expression collaboratively determine the applicability of the rule to the given input query. In the following, we describe the algorithms that accomplish deterministic transformations and the algorithms that support these transformations in our system – identifying the exclusivity and obligation knowledge, and the entrance locations for element types related by a containment relation.

3.1. Transformation algorithms

As pointed out in Subsection 2.3, our strategy divides the whole transformation process for the optimization of a query as three phases. We use a switch mechanism to control the applicability of rules with regard to a specific

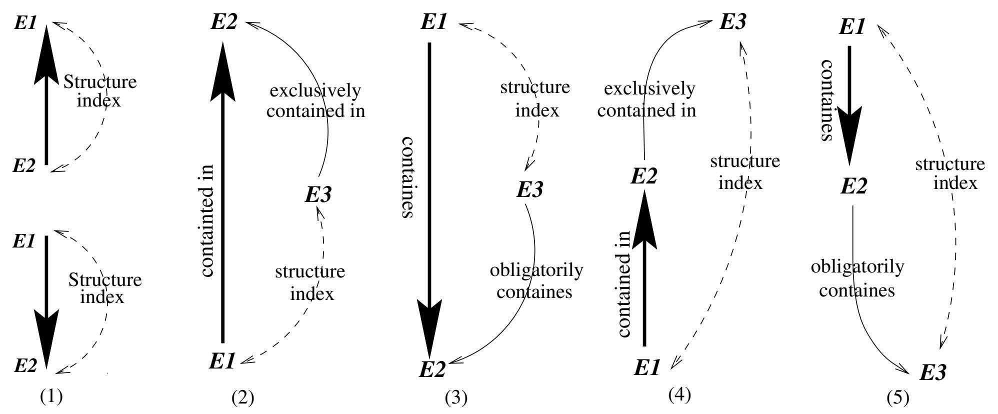

Figure 1. Combining entrance locations and structure Indices for optimal transformation

transformation phase for a given query expression. Another major characteristic of our strategy is the deterministic nature of our transformations, which makes our transformation process simplified and highly efficient. In other words, we don't concern multiple alternatives at each transformation step - we always keep a single optimal alternative. A high level description of our transformation 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 obtained by a separate Boolean array. The *enable* and *disable* are the two operations needed for turning on and off the switch of a rule. At each specific transformation phase, only the rules in "on" state will be considered. This is a simple but effective mechanism for putting the rule system under well control to achieve better performance.

The order of the rules in the rule system is of significance because in case of multiple applicable rules we always choose the first one to pursue our *deterministic transformation*. The first applicable rule in our system is believed to be the most beneficial one of all the applicable rules - this order arrangement is made based on relevant heuristics. The last alternative that our $transform$ algorithm obtained through a stepwise transformation chain is the optimal one. At this point, there are no more applicable "on" rules can be applied to that expression; otherwise the while loop in the `transform` algorithm would continue.

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 documents at the DTD level. These properties imply the corresponding relationships among element instances, but are practically identifiable through a given DTD. This saves significantly physical database accesses during the evaluation phase of queries.

Based on its definition, our algorithm for examining the exclusivity property between two given element types, e.g., whether B is exclusively contained in A, is simply to check the paths between B and the DTD's root type: if none 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 sub-element. Thus, as a first step we eliminate all optional occurrence indicators from the content models.

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 [3]:

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 ($A \supset B$), 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 ***extended 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 *extended 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 [3]:

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 work, content model normalization is important for revealing implicit obligation property, which is otherwise easily not to be identified [3, 9].

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

The essential six, i.e., $\mathcal{R}55$ to $\mathcal{R}60$, of 10 generic semantic rules ($\mathcal{R}51$ to $\mathcal{R}60$) in our system all rely on the *entrance location* notion. In addition, simplification rules $\mathcal{R}31$ through $\mathcal{R}34$ are based on availability of entrance locations. Thus how to find all *relevant* entrance locations to satisfy the conditions of and to enable these rules becomes an important implementation issue at this point.

As is depicted in Figure 1, the notion, "entrance location", as a requisite condition, is mainly used for enabling profitable transformations in two different manners: introduce a new element type $E3$ that is an *entrance location* for $E1$ and $E2$ (corresponding to case (2) and (3) in Figure 1), or $E2$ itself becomes an entrance location for the newly introduced element type $E3$ and the existing type $E1$ (corresponding to case (4) and (5) in Figure 1), and then replace $E2$ with the new type $E3$.

For terminological uniformity, we generalize our *entrance location* concept so that, when the term is generally used, it covers both the situations 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 general notion "entrance location" may refer to either an *inset entrance location*, applying to case (2) and (3) of Figure 1, or an *outside entrance location*, applying 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 element type graph. For the sake of efficiency, we suppose (precompute) that each element type ET maintains all the paths that diverge from the element type down to the leaves of the DTD-graph considered and call it $down\_paths(ET)$. The algorithm identifying inset entrance locations is described as follows:

```
identify_ELs(ET1, ET2) := {
  return type: set of <ent>;
  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;
}
```

This algorithm identifies all inset entrance locations for element type E1 and E2. It is evident from the algorithm that an entrance lactation EL is *relevant* for type E1 and E2 if it falls on the shortest path between the type $type(E1)$ and $type(E2)$.

With regard to identifying the outside entrance locations for element type E1 and E2, our algorithm is relatively simple because we have a much focused area to investigate for this case – only the types that are related to either E1 or E2 via a structure index need to be examined. For example, if EL is related to E1 by a structure index, we would just examine whether E2 is an inset entrance location for E1 and EL. In other words, we simply check every path from E1 to EL to see if it contains a solid occurrence of E2 on the path. We use the *outside entrance location* notion solely to enable the application of a superficially unrelated structure index (outside entrance locations do not contribute to the shortening of any navigation path).

3.4. Determine optimal entrance locations

For a given expression of form $E1 \overset{\supset}{\subset} E2$, algorithm $identify\_ELs(ET1, ET2)$ identifies relevant inset entrance locations for E1 and E2. The purpose for us to identify these entrance locations is to apply one of the semantic rules $\mathcal{R}55$ to $\mathcal{R}60$ to an (sub-)expression of form $E1 \overset{\supset}{\subset} E2$. But if the additional conditions of the rules regarding structure indices and exclusivity or obligation of a containment relation do not hold for the identified entrance locations, these entrance locations are not interesting. So we need to further identify those interesting entrances from all identified relevant ones. After that, if multiple interesting entrance locations exist, we still need to choose the most profitable one, i.e., the optimal one, to activate a corresponding rule of the set $\mathcal{R}55$ through $\mathcal{R}60$ because our deterministic transformation strategy asks us to always apply the most beneficial rule to a query expression whenever multiple are applicable. Therefore, we need certain criteria to determine which one is the optimal entrance location for a given (sub-)expression of form $E1 \overset{\supset}{\subset} E2$.

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 $\mathcal{R}60$ to an expression of form $E1 \overset{\supset}{\subset} E2$, 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 extension[2].

(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.

Now we give the steps that are needed by our algorithm to determine the optimal entrance location for a given query (sub-)expression of form $E1 \overset{\supset}{\subset} E2$.

```
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 the optimal entrance
          location and return it;
}
```

[2]The extension of an element type is the set of all the element instances conforming to that type.

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 $identify\_ELs$-$(E1, E2)$; the result is annotated as $rELs(E1, E2)$, which is a set of element type names; all relevant outside entrance locations can be decided accordingly 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 an available structure index.

For identifying interesting entrance locations, the following two cases are 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 yes, 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 (in other words, 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 differentiate the output of last step as either "structure index defined" (thus will be applied) or "no structure index defined" (will be used only for shortening the navigation paths):

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

(Case 2) For the entrance locations output by Step 2 and not marked as "structure index defined", select the one that produces the shortest navigation path for reaching $type(E1)$ in the element type graph.

The last step of the above algorithm output the optimal entrance location that actually helps decide which of the six essential semantic rules is the most profitable one and thus should be chosen next for conducting transformation on the incoming (sub-)expression of form $E1 \overset{\supset}{\subset} E2$.

4. Transformation examples

In the following we give three transformation examples using our deterministic transformation rules. Our transformations explore the following indices: content index $I_{\sigma_r}(Surname)$ and structure indexes $I_{Article}(Name)$, $I_{Paragraph}(Article)$, and $I_{Paragraph}(ShortPaper)$. For structure indices, the subscript of the index operator indicates the result type of the index operation. Since we did not introduce all the transformation rules used in the fowllowing examples beforehand due to the space limitation and the purpose of this paper, we do not expect a sharp understanding of the details of the examples from the readers but just to give them a tast of the flavors of our deterministic transformation system for query optimization.

Example 1. Find the paragraphs of the "Introduction" section of each article that has the words "Data Warehousing" in its title.

$(\ Paragraph \subset (\sigma_{A=Title, r=\text{`}Introduction'}(Section)$
$\subset (Article \supset \sigma_{r=\text{`}Datawarehousing'}(Title))) \ )$
$\Longrightarrow$ (by 1st step of $\mathcal{R}54$, i.e., associativity)
$(\ (Paragraph \subset \sigma_{A=Title, r=\text{`}Introduction'}(Section))$
$\subset (Article \supset \sigma_{r=\text{``}Datawarehousing''}(Title)) \ )$
$\Longrightarrow$ (by 2nd step of $\mathcal{R}54$, i.e., commutativity)
$(\ (Paragraph \subset (Article \supset$
$\sigma_{r=\text{`}Datawarehousing'}(Title))) \subset$
$\sigma_{A=Title, r=\text{`}Introduction'}(Section) \ )$
$\Longrightarrow$ (by 3rd step of $\mathcal{R}54$, i.e., index introduction)
$(\ (I_{Paragraph}(Article \supset$
$\sigma_{r=\text{`}Datawarehousing'}(Title)) \cap Paragraph)$
$\subset \sigma_{A=Title, r=\text{`}Introduction'}(Section) \ )$
$\Longrightarrow$ (by $\mathcal{R}61$: $\cap$ deletion)
$(\ (I_{Paragraph}(Article$
$\supset \sigma_{r=\text{`}Datawarehousing'}(Title)))$
$\subset \sigma_{A=Title, r=\text{`}Introduction'}(Section) \ )$

Example 2. Find all articles in which the surname of an author contains the value "Aberer".

$(\ Article \supset \sigma_{r=\text{`}Aberer'}(Surname) \ )$
$\Longrightarrow$ (by $\mathcal{R}50$, supose index $I_{\sigma_r}(Surname)$ is ailable)
$(\ Article \supset I_{\sigma_{r='Aberer'}}(Surname) \ )$

Notice that $\mathcal{R}57$ is unfortunately not applicable for using the index between *Name* and *Article* since the *free expression* condition does not hold for $\sigma_{r=`Aberer'}(Surname)$.

Example 3. Find all "Summary" paragraphs of all sections (if any).

$(\sigma_{A=Title,r=`Summary'}(Paragraph) \subset Section)$
$\Longrightarrow$ (by $\mathcal{R}57$)
$(\sigma_{A=Title,r=`Summary'}(Paragraph) \subset Article)$
$\Longrightarrow$ (by $\mathcal{R}53$)
$(I_{Paragraph}(Article) \cap \sigma_{A=Title,r=`Summary'}(Paragraph))$
$\Longrightarrow$ (by $\mathcal{R}62$)
$(\sigma_{A=Title,r=`Summary'}(I_{Paragraph}(Article)))$

5. Summary

Query optimization is a critical and challenging issue for structured document management, especially for efficient query processing. Being driven by the high efficiency request by the web-hooked applications for structured document query processing, our approach applies exclusively deterministic transformations on structured document queries to achieve the best possible optimization efficiency. Our strategy is based on heuristics about DTD knowledge and structural properties of XML documents, with the main purpose for fast exploring potential structure indices to speed up query processing in a database environment. In this paper, we addressed mainly the implementation issues of our deterministic query transformation strategy. Our work is original as we did not notice really closely related work being done so far to the best of our knowledge.

Acknowledgement. The author would like to show great appreciation to his former colleagues at Fraunhofer-IPSI (formerly known as GMD-IPSI), Germany. This continuing research at the author's current affiliation was initiated at GMD-IPSI in close collaboration with Prof. Karl Aberer, Dr. Klemens BÖehm, and Prof. M. Tamer Özsu (during his visit to GMD-IPSI on his sabbatical leave).

References

[1] Serge Abiteboul, Sophie Cluet, Vassilis Christophides, et al. *Querying Documents in Object Databases*. Digital Libraries. No. 1, pp5-19, 1997.

[2] Klemens Böhm, Karl Aberer, Erich J. Neuhold, and Xiaoya Yang. *Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM*. The VLDB Journal, Vol. 6, No. 4, pp296-311, November 1997.

[3] Klemens Böhm, Karl Aberer, M. Tamer Özsu, and Kathrin Gayer. *Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition*. Proc. of IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL'98), pp196-205, Santa Barbara, California, April 22-24, 1998.

[4] P. Bohannon, J. Freire, P. Roy, J. Simon. From XML Schema To Relations: A Cost-Based Approach to XML Storage. Proc. of the 18th International Conference on Data Engineering (ICDE'02).

[5] C.Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions", Proc. of Intl' Conference on Data Engineering, pp235-244, San Jose, California, February 2002.

[6] Chee Yong Chan, Minos N. Garofalakis, Rajeev Rastogi: RE-Tree: An Efficient Index Structure for Regular Expressions. Proc. of VLDB 2002.

[7] Surajit Chaudhuri and Luis Gravano. *Optimizing Queries over Multimedia Repositories*. Proc. of SIGMOD'96, pp91-102, Montreal, Canada, June 1996.

[8] Dunren Che and and Karl Aberer. *A Heuristics-Based Approach to Query Optimization in Structured Document Databases*. Proc. of 1999 International Database Engineering & Application Symposium, pp24-33, Montreal, Canada, August 2-4, 1999, IEEE Computer Society.

[9] Dunren Che, Karl Aberer, M. Tamer Özsu, and Klemens Böhm. *Query Processing and Optimization in Structured Document Database Systems*. Manuscript in preparation for publication on The VLDB Journal.

[10] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, Carlo Zaniolo. *Efficient Structural Joins on Indexed XML Documents*. Proc. of VLDB 2002, Hong Kong.

[11] Mariano Consens and Tova Milo. *Optimizing Queries on Files*. Proc. of the 1994 ACM SIGMOD International Conference on Management of Data, pp301-312, Vol. 23, ACM Press, May 1994, Minneapolis, Minnesota.

[12] Mary F. Fernandez and Dan Suciu. *Optimizing Regular Path Expressions Using Graph Schemas*. Proc. of the Fourteenth International Conference on Data Engineering, pp14-23, February 23-27, 1998, Orlando, Florida, USA, 1998.

[13] M. Fernandez, W. Tan, D. Suciu. SilkRoute: Trading between Relations and XML. 9th Int. World Wide Web Conf. (WWW), Amsterdam, May, 2000

[14] D. Florescu, and D. Kossmann. Storing and Querying XML Data Using an RDMBS. IEEE Data Engineering Bulletin 22(3), pp. 27-34, 1999.

[15] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. *Information Retrieval–Data Structures and Algorithms*. New Indices for Text: PAT trees and PAT arrays, Prentice hall, 1992.

[16] Georg Gottlob, Christoph Koch, Reinhard Pichler. *Efficient Algorithms for Processing XPath Queries*. Proc. of VLDB 2002.

[17] Torsten Grust: Accelerating XPath location steps. Proc. of SIGMOD Conference 2002: 109-120.

[18] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava and Ting Yu. *Approximate XML joins*. Proc. of the ACM SIGMOD Conference on Management of Data, 2002

[19] 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

[20] Kyuchul Lee, Yong Kyu Lee and P. Bruce Berra. *Management of Multi-Structured Hypermedia Documents: A Data Model, Query Language, and Indexing Scheme*. Multimedia Tools and Applications, Vol. 4, No. 2, pp199-224, march 1997.

[21] Wen-Syan Li, Junho Shim, K. Selcuk Candan and Yoshinori Hara. *WebDB: A Web Query System and its Modeling, Language, and Implementation*. Proc. of IEEE International Forum on Research and Technology Advances in Digital Libraries (ADL'98), pp216-225, Santa Barbara, California, Aprill 22-24, 1998.

[22] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions, Proc. of the 27th International Conference on Very Large Databases (VLDB'2001), pp361-370, Rome, Italy, September 2001.

[23] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. *Lore: A Database Management System for Semistructured Data*. SIGMOD Record, 26(3), pp54-66, September 1997.

[24] J. McHugh and J. Widom. *Query Optimization for XML*. Proc. of the Twenty-Fifth International Conference on Very Large Data Bases, pp315-326, Edinburgh, Scotland, September 1999.

[25] Tova Milo, Dan Suciu: Index Structures for Path Expressions. Proc. of ICDT 1999: 277-295.

[26] Atsuyuki Morishima and Hiroyuki Kitagawa. *A Data Modeling and Query Processing Scheme for Integration of Structured Document Repositories and Relational Databases*. Proc. of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1-4, 1997.

[27] Atsuyuki Morishima and Hiroyuki Kitagawa. *A Data Modeling Approach to the Seamless Information Exchange among Structured Documents and Databases*. Proc. of 1997 ACM System on Applied Computing, San Jose, Feb. 1997.

[28] Gonzalo Navarro and Ricarrdo Baeza-Yates. *Proximal Nodes: A Model to Query Document Databases by Content and Structure*. ACM Transaction on Information Systems, Vol. 15, No. 4, pp400-435, October 1997.

[29] M. T. Özsu, P. Iglinski, D. Szafron, S. El-Medani, M. Junghanns. *An Object-Oriented SGML/HiTime Compliant Multimedia Database Management System*. Proc. of Fifth ACM International Multimedia Conference (ACM Multimedia'97), pp239-249, Seattle, WA, November 1997.

[30] A. Salminen and F. W. Tompa. *PAT Expressions: an Algebra for Text Search* Acta Linguistica Hungarica 41 (1994), no.1, pp277-306.

[31] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. Proc. of VLSB, pp. 302-314, 1999.

[32] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. Proc. of ICDE02.

[33] 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), Hongkong, June 2000.

[34] Tak W. Yan and Jurgen Annevelink. *Integrating a Structured-Text Retrieval System with an Object-Oriented Database System*. Proc. of the 20th VLDB Conference, pp740-749, Santiago, Chile, 1994.

Appendix

$\mathcal{R}$55. $(E1 \subset E2) \Longrightarrow (E1 \subset E3)$ ***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.

$\mathcal{R}$56. $(E1 \supset E2) \Longrightarrow (E1 \supset E3)$ ***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.

$\mathcal{R}$57. $(E1 \subset E2) \Longrightarrow (E1 \subset E3)$ ***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.

$\mathcal{R}$58. $(E1 \supset E2) \Longrightarrow (E1 \supset E3)$ ***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.

$\mathcal{R}$59. $(E1 \subset E2) \Longrightarrow (E1 \subset E3)$ ***if*** $type(E3)$ is an entrance location for $type(E1)$ and $type(E2)$, and is exclusively contained in $type(E2)$, and $free(E2)$ holds.

$\mathcal{R}$60. $(E1 \supset E2) \Longrightarrow (E1 \supset E3)$ ***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 $free(E_i)$ requests that the evaluation of the E_i returns the full extent of the element type of E_i, i.e., all elements of the type in the database. One typical example of such an E_i is an expression comprising just an etn, meaning all elements of the type 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 less ID#) and is tried first during query optimization for enabling a potential structure index. Analogously for $\mathcal{R}56$ and $\mathcal{R}60$.