

A Deterministic Approach to XML Query Processing with Efficient Support for Pure and Negated Containments

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

ABSTRACT

This article reports the result of the author's recent work on XML query processing/optimization, which is a very important issue in XML data management. In this work, in order to more effectively and efficiently handle XML queries involving pure and/or negated containments, a previously proposed deterministic optimization approach is largely adapted. This approach resorts to heuristic-based deterministic transformations on algebraic query expressions in order to achieve the best possible optimization efficiency. Specialized transformation rules are thus developed, and efficient implementation algorithms for pure and negated containments are presented as well. Experimental study confirms the validity and effectiveness of the presented approach and algorithms in processing of XML queries involving pure and/or negated containments.

Keywords: XML, Database, Data querying, Query processing, Query optimization

INTRODUCTION

XML has become the de facto standard for information and data exchange and representation on the Internet and elsewhere. As a result, more and more data sources have been adopting the XML standard. The rapid accumulation of XML data calls for specialized solutions for managing and querying XML data resources. Among the many challenges related to the XML database management technology, XML query optimization is very interesting because it not only is a critical issue for XML DBMS but also provides a key infrastructure for the future semantic web and applications (especially, the semantic-based web search engines).

Query optimization typically applies the cost-based approach (Selinger, 1979) and aims at obtaining the least expensive – the optimal – evaluation plan for each input query. Heuristic knowledge may be exploited in addition – so that a reduced number of (only highly potential) candidate plans need to be examined and from which the best plan is to be identified.

XML data has the semi-structured nature, and XML queries need to check not only the *contents* but also the *structural patterns* of the source XML data.

Comparing with relational data, XML data has higher complexity (as it has to additionally deal with the structural part), and this complexity trivially translates to an enlarged search space for the “optimal” plan during query optimization (assuming the cost-based approach is adopted). Consequently, a plain application of the cost-based optimization approach does not usually yield the same good efficiency for XML queries as it does for relational queries. On the other hand, apart from adding extra complexity and causing inefficient (cost-based) query optimization, the structural part of XML data implies a rich resource of knowledge that can be used in favor of efficient optimization of XML queries. We are thus motivated to develop a comprehensive optimization framework for XML queries. This framework consists of two separate yet collaborative optimization stages. The first stage performs logical-level optimization. This stage is unique to XML as it explores the specific features (e.g., semantic knowledge) of XML data for query optimization. By its nature, this stage is strongly heuristic-based because it does not rely on any particular knowledge about the underlying storage structure. The second stage – physical optimization – typically applies specialized cost-based optimization techniques. In such an optimization framework, these two stages need to collaborate in a way that the first stage provides a reduced (or pre-screened) set of logical plans to the second stage and the latter shall conduct specialized techniques for cost-based optimization by adequately considering the optimization that has already been exerted at the first stage.

In previous research (Che, 2003, 2005, 2006), we studied the query equivalence issue in the context of XML, which form the basis of our transformation-based optimization approach to XML queries. A comprehensive methodology for fast XML query optimization at the logic level was proposed (Che, 2006) based on exclusive application of deterministic transformations on XML queries represented as PAT algebraic expressions (Salminen 1994; Böhm, 1998). The utmost benefit of this unique approach is the great potential for superb optimization efficiency. More recently, we substantially extended the PAT algebra, which leads to ePAT (extended PAT), endowed with more expressive power. Based on ePAT, we redefined the query equivalences and the deterministic transformation rules, and adapted the prior optimization strategy in order to efficiently support XML queries with pure and/or negated containments. Containment is a core operation in XML queries, and negated containment is as important as the regular containment for XML queries (for example, “find all employees who do *not* have a homepage”, though simple, can be a common (sub-)query pattern). However, little result on efficient support for pure and negated containments has been reported.

In this paper, we make the following important contributions:

- An adapted deterministic optimization approach for XML queries with pure and/or negated containments is presented.
- A group of specialized join algorithms dedicated for pure and negated containment operations are proposed (realizing the known structural join algorithms (Zhang, 2001; Srivastava, 2002) cannot provide efficient support for these special containment operations).
- An experimental study is conducted, and the obtained result confirms the validity and effectiveness of this new approach and the specialized supporting algorithms.

The remaining of this paper is set forth as follows: Section 2 provides necessary

background knowledge and a sketchy overview of the new approach. Section 3 illustrates the new approach by presenting sample deterministic transformation rules that are later on applied to a running example (query). Section 4 presents a number of specialized join algorithms intended for pure and negated containments. Section 5 addresses the experimental study and result. Section 6 reviews related work before the paper is finally concluded at Section 7.

BACKGROUND KNOWLEDGE

The main theme of this work is to carry out deterministic transformations on algebraic query expressions for optimization. This section first review a few important notions, then the ePAT algebra adopted in this work; after that, sample XML query equivalences are provided, and lastly, an overview of the adapted approach is furnished.

Basis Notions

In XML documents/data, elements are identified by means of “tags”. For a given class of documents, the legal markup tags are defined in a DTD/XSD (XML Schema Definition). DTD and XSD serve basically the same kind of function, but the latter is more powerful. The presentation in this paper is based on DTD, but the approach presented shall straightforwardly extend to the situations when XSD is used instead.

A tag (with a distinct name) defined in a DTD corresponds to a type of XML data elements, which may comprise some other tags in its content model. The constructional relationships among the element types are an important source of knowledge that will be utilized by our approach for efficiently pursuing XML query optimization. We introduce the *DTD-graph* mechanism to help envision important structural relationships among the element types. In stead of providing a formal definition for this notion, we use an example to illustrate it. Fig. 1 shows part of the structural relationships existing among the elements (types) about open auctions, which was adopted by the XMark benchmark project (Schmidt, 2002). In Fig. 1, a dashed arrowed line (which is not a necessary part in a DTD-graph) indicates a structure index is available. A *structure index* is formally defined as a map from the extent of one element type to the extent of another element type (Che, 2006). A structure index may be intuitively considered as a “shortcut” between the two related extents (of two different element types). In our test-bed, structure indices are implemented as materialized (or pre-computed) structural join results (Zhang, 2001; Srivastava 2002).

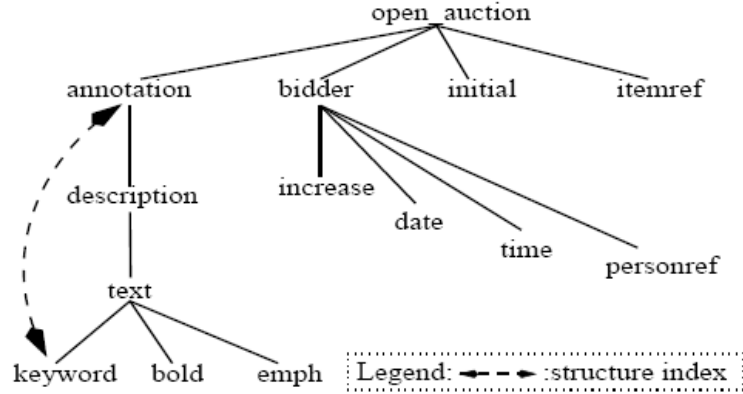


Figure 1: A portion of the XMark benchmark DTD-graph annotated with index info.

Containment is the dominant structural relationship in XML and is at the core of both the query languages, XPath (Clark, 1999) and XQuery (Boag, 2003). Containments can be differentiated between direct containments, i.e., parent-child relations, and indirect containments, i.e., ancestor-descendent relations. A *path* in a DTD-graph is a sequence of edges. Paths are the primary means used in both XQuery (Boag, 2003) and XPath (Clark, 1999) to capture the structural relationships among the data elements.

In addition to the DTD-graph and the path notions, we adopt three other important notions for capturing particularly interesting structural relationships in XML data. These notions were initially introduced by (Boehm, 1998) and is extensively exploited in (Che, 2006). We review the three notions below.

Definition (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 the document root contains an element of type ET_i .*

Definition (Obligation) *Element type ET_i obligatorily contains element type ET_j if each element of type ET_i in any document complying with the DTD has to contain an element of type ET_j .*

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

The ePAT Algebra for XML Query Optimization

It is typical in a query optimizer to represent queries as algebraic expressions for expediently exploring alternate evaluation plans based on equivalent transformations. ePAT is the algebra adopted in this work, which is extended from PAT (Salminen, 1994; Böhm, 1998).

In ePAT, an expression is generated according to the following grammar:

$$\begin{aligned}
 E ::= & \text{etn} \mid (E) \mid E1 \cup E2 \mid E1 \cap E2 \mid E1 - E2 \\
 & \mid \sigma_r(E) \mid \sigma_{a,r}(E) \mid E1 \subset E2 \mid E1 \supset E2 \\
 & \mid E1 \times_c E2 \mid \pi_{pl}(E) \mid I(E) \mid \neg E
 \end{aligned}$$

“E” (as well “E1” and “E2”) stands for an ePAT expression, and $e \in n$, as the only atomic expression, retrieves the whole extent (i.e., all the instances) of the element type named $e \in n$. The expression (E) produces the same result as E and is needed mainly for composition purpose. \cup , \cap and $-$ are the three standard set operations. ePAT requests type compatibility for its set operations. $\sigma_r(E)$ and $\sigma_{a,r}(E)$ are the two basic selection operators applied to the textual contents and attribute values of elements, respectively. The “r” parameter in the two selection operators introduces a regular expression specifying a matching condition on the textual contents or attribute values of elements, and the “a” in $\sigma_{a,r}(E)$ designates a particular attribute name. \sqsubset returns elements of the first argument that are contained in an element decided by the second argument, and \supset returns elements of the first argument that contain an element of the second argument. \times_c is a “power” join operation that can be tailored to almost any specific form of joins such as structural joins (Zhang, 2001; Srivastava, 2002), *unstructural joins* (i.e., joins not based on any structural relationship), and Cartesian products based on the nature of the join predicate specified by the parameter c. π_{pl} is the projection operator that carries a projection list specified by the parameter pl. I is a generic operation that connotes the application of a relevant index of any type (such as content indices of elements, value indices of attributes, and structure indices). Finally, $-E$ stands for the “negated” form of E and here the E subexpression is limited to only a selection or a containment operation. When it is a selection, the negation is logically applied to the filtering condition of the selection operation; when it is a containment, e.g., $E_1 \supset E_2$, the negation is applied to the containment operation.

It is helpful to point out that in the ePAT algebra, the containment operations, \supset and \sqsubset , are redundant (analogous to the natural join operation in the relational algebra), considering that they can be represented as a general structural join followed by a leftward projection. Yet, there are advantages of retaining this redundancy: considering the dominant role of containment operations in XML data and XML queries, dedicated containment operations greatly facilitate exploiting the structural relationships in XML data for query optimization.

We use the following query as a running example. First, we show how the query is formulated as an ePAT expression. Later on, we show how (logical) optimization can be achieved through consecutive deterministic transformations performed on ePAT expressions.

Example. Find the keyword from the annotation of an `open_auction` that has received a bid from the person (an instance of `bidder`) whose reference is 123.

A partial DTD-graph of the XML data source is shown in Fig. 1. The corresponding XPath expression and ePAT expression are respectively formulated as follows (our query optimization and processing will be solely based on ePAT expressions).

XPath:

```
//open_auction[./bidder[@personref='123']]/annotation//keyword
```

ePAT:

```
(keyword  $\sqsubset$  (description  $\sqsubset$  (open_auction  $\supset$   $\sigma_{a='personref', r='123'}(Bidder))))$ 
```

Sample XML Query Equivalences

Our approach to XML query optimization is based on equivalent algebraic transformations, which are accomplished via deterministic transformation rules derived from equivalences in cooperation with relevant heuristics for query optimization in the context of XML. While these deterministic rules embody our optimization methodology, the equivalences form the basis of this work. In the following, sample equivalences are provided for illustration purpose.

We identify three knowledge sources from which equivalences can be derived: set-theory properties, explicit constraints imposed by DTD/XSD, and hidden constraints representing knowledge about the structure of XML data (often implied in DTD/XSD). Accordingly, our equivalences are classified into three categories.

It is worth noting that we would have obtained a (very) large number of equivalences including less-useful or even useless ones, if we do not limit our choice to only those potential ones from which profitable transformations can be derived. For instance, an equivalence must not imply transformations that merely complicate/expand a query without improving the query's evaluation efficiency (from a heuristic point of view).

In our work, equivalences take the form of “ $E1 \Leftrightarrow E2$ ”, and transformation rules take the form of “ $E1 \Rightarrow E2$ ”. For compactness, we will use “ σ ” to simply represent both σ_r and $\sigma_{a,r}$, and use EL as a short hand notation for “entrance location”.

In the following, we present sample equivalences from each category.

(1) Set-Oriented Equivalences

As ePAT is a set based algebra, a group of equivalences were directly derived from the set-theoretic properties of relevant ePAT operators. A couple of these equivalences are given below (A complete list can be found in (Che, 2006)).

$$\varepsilon 1. (E1 \supset E2) \cap E3 \Leftrightarrow (E1 \cap E3) \supset E2$$

$$\varepsilon 2. (E1 \supset E2) \supset E3 \Leftrightarrow (E1 \supset E3) \supset E2$$

Equivalence $\varepsilon 2$ is called the commutativity law of the containment operations.

(2) Equivalences Based on Explicit DTD Constraints

There are equivalences based on checking the explicit constraints as given by the DTD/XSD. Two such examples are given below (symbol τ is used to denote the resultant type of a query expression):

$$\varepsilon 3. \sigma_{a,r}(E) \Leftrightarrow \phi \text{ if 'a' isn't an attribute of } \tau(E)$$

$$\varepsilon 4. E1 \subset E2 \Leftrightarrow \phi \text{ if } \tau(E1) \text{ isn't contained in } \tau(E2)$$

(3) Equivalences based on derived constraints about XML document structure

This is a far more interesting class of equivalences as they bear the potential of rendering significant improvement on input query expressions. Following are a few samples of these type of equivalences:

$$\varepsilon 5. (E1 \supset E2) \supset E3 \Leftrightarrow E1 \supset (E2 \supset E3) \text{ if } \tau(E2) \text{ is an EL for } \tau(E1) \text{ and } \tau(E3)$$

- ε6. $(E1 \supset E2) \supset E3 \Leftrightarrow E1 \supset (E3 \supset E2)$ if $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$
 ε7. $E1 \subset E2 \Leftrightarrow E1 \subset (E3 \subset E2)$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$

ε5 refers to as the *associativity* of the containment operations. ε6 embodies both the properties of commutativity and associativity in a single equivalence, thus we coined a new name, *cossoassociativity*, to characterize this specific algebraic law in the context of XML. ε7 directly connotes the meaning of the entrance location concept. Similar equivalences hold when all occurrences of \supset in ε1, ε2, ε5 and ε6 are substituted by \subset . Combined application of these algebraic laws with the concepts exclusivity, obligation, and entrance location introduced at the beginning of this section yields more equivalences, which form the basis of our deterministic optimization approach as detailed in (Che, 2006).

Strategy Overview

We identify a large set of potential equivalences as the basis for XML query optimization; these equivalences are centered around the containment operations. However, equivalences are not directly used in our approach for query optimization (this is in contrast to the traditional way such as in (Graefe, 1987) where equivalences are directly used for plan enumeration). Instead, we recognize the potential of XML document structure in query optimization and the specific feature of XML queries (i.e., always carrying a structure pattern). We are prompted to focus our attention on the logic-level optimization of XML queries as the first step toward thorough optimization. As such, optimized queries may need be further complemented by a succeeding physical optimizer (cost-based) in case highly optimized query plans are expected, e.g., by those repeatedly running and complex queries. Otherwise, the logical optimizer alone can be used.

Therefore, our emphasis is on logical optimization and our approach is designed to extensively explore the rich structure knowledge of XML data to achieve fast XML query optimization.

To obtain the above goal, we derive heuristic-based, deterministic transformation rules based on the identified equivalence in collaboration with relevant heuristics and the availability of structure indices built in an XML database. These transformation rules are applied to each query (expression) step-by-step, and each performed transformation solidly improves the input query (in terms of evaluation efficiently) until no more transformations can be performed.

During this process, transformation rules are arranged in a certain order according to well-established optimization heuristics. When there are multiple rules applicable to an input expression, the precedence is always given to the first applicable rule. The details of our rule control strategy and the transformation algorithms can be found in (Che, 2005) and (Che, 2006).

In this paper, we limit our discussion to the optimization issue of query expressions involving only the main ePAT operators: $e\tau n$, \cap , \cup , $-$, σ , \supset , \subset . Logical optimization in this work consists of three consecutive transformation phases: *normalization*, *semantic optimization*, and *simplification* (clean-up). The normalization phase enforces straightforward DTD/XSD constraints, performs straightforward simplification, and reorders operators in the following order (from bottom up): $e\tau n$, \cap , $-$, σ , \supset , \subset , \cup . It is interesting to note that, after this phase, all \cap and $-$ operations sediment to the bottom on the operation tree and get ultimately

eliminated from the query expressions after applying dedicated transformation rules.

The second phase, semantic optimization, starts with a normalized ePAT expression, explores deep structure knowledge of XML data in order to render an application of a potential structure index into the query. Due to the particular pattern that a query expression may be in, structure indices, although available, may not be applicable to the query. The central goal of the second phase is to reveal such opportunities.

The third phase performs a thorough cleaning-up. For this purpose, an intensive set of simplification rules are needed, including rules specially added for handling the possible redundancy that might be rendered by the second (main) phase.

A key feature of our approach is that deterministic transformations are exclusively applied to all queries at all three phases to achieve quick optimization. Consequently, a typical problem of all rule-based systems – uncontrollable runtime behavior – is expediently circumvented. A second important feature is that after thorough transformation of an input query for optimization, the \cap and $-$ operations eventually get eliminated. This once again brings enhanced performance to the optimization itself and to the optimized queries.

DETERMINISTIC XML QUERY OPTIMIZATION

This section presents selected transformation rules and addresses the heuristics behind these rules.

Normalization

The normalization phase has three main tasks: enforcing simple DTD-constraints, reordering the operators (to facilitate semantic optimization at the second phase), and performing straightforward simplification.

(1) Enforcing explicit DTD-constraints

We do not assign this task to the query parser because we do not generally assume the existence of database schemas in the context of XML due to the semi-structured nature of XML data. This group of rules checks the queries' consistency with the explicit constraints induced by a given DTD/XSD (if any). A couple of example rules are given below:

$$R1. \sigma_{a,r}(E) \Rightarrow \phi \text{ if 'a' is not an attribute of } \tau(E)$$

$$R2. E1 \subset E2 \Rightarrow \phi \text{ if } \tau(E1) \text{ is not contained in } \tau(E2)$$

(2) Operator reordering

The goal of this group of rules is to completely eliminate the \cap and $-$ operations from a query, and re-permute the remaining operators in the following order: $e \in n$ and σ at the bottom, \supset and \subset in the middle, \cup on the top, in order to facilitate the subsequent semantic optimization phase to achieve the primary goal – bringing a beneficial structure index into a query if possible. Several example rules are given below:

$$R3. \sigma(E1 \cup E2) \Rightarrow \sigma(E1) \cup \sigma(E2)$$

$$R4. E1 - (E2 \supset E3) \Rightarrow E1 \cap (E2 \equiv E3)$$

$$R5. (E1 \supset E2) \cap E3 \Rightarrow (E1 \cap E3) \supset E2$$

$$R6. E \cap E \Rightarrow E$$

$$R7. \sigma(E1 \supset E2) \Rightarrow \sigma(E1) \supset E2$$

$$R8. (E1 \supset E2) \supset E2 \Rightarrow E1 \supset E2$$

$$R9. \neg\sigma(E) \Rightarrow \bar{\sigma}(E)$$

$$R10. \neg(E1 \supset E2) \Rightarrow E1 \equiv E2$$

Successful application of R5 makes it possible for R6 to eliminate an occurrence of the \cap operator. The binary ‘ \neg ’ operator is dealt in a similar way. The unary ‘ \neg ’ operation (which negates a selection or a containment) is eliminated in a different way: R9 and R10 absorb the negation into the subsequent selection or containment operation (for convenience presentation, in these rules “ \equiv ” is used to denote the directly negated form of \supset , and “ $\bar{\sigma}$ ” represents the negated form of σ). Ideally, specialized join algorithms for \equiv will be provided for direct and efficient implementation of the operation, while σ is easily handled by simply interpreting the negation as negating the selection operator’s filtering predicate. The above discussion similarly applies to the \equiv operator, i.e., by working with the counterparts of rule R4, R5, R7, R8, and R10 after \equiv is replaced by \equiv in these rules.

(3) Straightforward simplification

We observe that most of the simplification carried out at an early stage are beneficial, e.g., $E \subset E \Rightarrow E$, while some less straightforward simplification if hastily done at a too early stage may cause the whole optimization process to be quickly captured at a local optimal. For example, while a leftward application of ε_7 eliminates E3, it takes away the opportunity of applying the structure index between $\tau(E3)$ and $\tau(E2)$ if such a structure index is indeed available, or later on, a reverse transformation has to be called upon to restore the lost opportunity. Therefore, at the normalization phase we only pursue straightforward simplification, and the simplification rules mostly are only related to the set operations. We furnish a couple of such rules for simplification in the following.

$$R11. \sigma(E) \cup E \Rightarrow E$$

$$R12. (E1 \supset E2 \cup E1) \Rightarrow E1$$

A counterpart of R12 analogously works with the \subset operation.

Semantic Optimization

The second phase, semantic optimization, is the core in our optimization approach. The primary goal of this phase is to discover potential opportunities of applying structure indices to a query. Three situations have been identified for this purpose:

- A structure index is available for a \supset or \subset operation in an input expression and the index is then simply applied. This is by far the most favorable situation.
- A structure index is available but is not superficially applicable because of an unfavorable pattern that the query expression is in. However, after desired transformations are performed, the index becomes applicable and used.

- No relevant structure index is available, but a after introducing a third element type as an entrance location into a query, a previously irrelevant index becomes relevant and eventually gets applied.

In summary, the first situation applies a structure index effortlessly, the second situation makes an “inapplicable” index applicable, and the third situation makes an “irrelevant” index relevant and applied. In the following, we present sample rules that exploit the usage of structure indexes corresponding to each of these situations:

(1) Applying readily available structure indices: For an input (sub)query of form “ $E1 \supset E2$ ”, if a structure index between the two involved element types, $\tau(E1)$ and $\tau(E2)$, happens to be available, our strategy firmly grasp this opportunity and applies the index by invoking the following transformation rule (where the parameter $\tau(E1)$, appearing as a subscript of the index operator I , indicates the resultant type of the index operation; a similar rule holds for \subset).

$$R13. (E1 \supset E2) \Rightarrow (I_{\tau(E1)}(E2) \cap E1) \text{ if a structure index is available between } \tau(E1) \text{ and } \tau(E2)$$

(2) Making an “inapplicable” structure index applicable: There are chances that a relevant structure index is available but not applicable because of the particular pattern that a query expression is in. Nevertheless, the applicability of the structure index to the query can enabled by desired transformations rendered by the commutativity and/or associativity laws w.r.t. \supset and \subset in the ePAT algebra. Specifically, the following transformation rules can be used (note the counterparts of R14 and R15 analogously work for \subset):

$$R14. (E1 \supset (E2 \supset E3)) \Rightarrow ((E1 \supset E2) \supset E3) \text{ if there exists a structure index between } \tau(E1) \text{ and } \tau(E2), \text{ and } \tau(E2) \text{ is an EL for } \tau(E1) \text{ and } \tau(E3)$$

$$R15. (E1 \supset (E2 \supset E3)) \Rightarrow ((E1 \supset E3) \supset E2) \text{ if there exists a structure index between } \tau(E1) \text{ and } \tau(E3), \text{ and } \tau(E2) \text{ is an EL for } \tau(E1) \text{ and } \tau(E3)$$

$$R16. (E1 \subset (E2 \supset E3)) \Rightarrow ((E1 \subset E2) \supset E3) \text{ if there exists a structure index between } \tau(E1) \text{ and } \tau(E2), \text{ and } \tau(E1) \text{ is an EL for } \tau(E2) \text{ and } \tau(E3)$$

$$R17. (E1 \supset (E2 \subset E3)) \Rightarrow ((E1 \supset E2) \subset E3) \text{ if there exists a structure index between } \tau(E1) \text{ and } \tau(E2), \text{ and } \tau(E1) \text{ is an EL for } \tau(E2) \text{ and } \tau(E3)$$

These rules are a result of a guided application of the equivalences that relate to the entrance location notion. A comprehensive list of the rules can be found in (Che, 2006).

(3) Making “irrelevant” structure indexes relevant and applicable: For an input query, if there is no structure index found relevant, our approach checks whether it is possible to make a related index relevant by introducing an entrance location (a new element type) into the query (a related index is one that relates to just one of the involved element types, and thus is not quite relevant). To explore this kind of opportunities, we need specialized (and more complex) transformation rules that check specific structural properties, e.g., those connoted by the exclusivity and/or obligation notions, in addition. Our approach examines numerous cases, from most beneficial to least favorable in a heuristic point of view, in order to achieve the above goal. In many of these cases, element type substitution is

performed so that an opportunity of bringing a potential structure index into a query is discovered. After a desired transformation is performed, R13 is then called during the next iteration to secure the application of an identified potential structure index. The transformation rules corresponding to the four most favorable cases were identified in our prior work (Che, 2006) are presented below:

R18. $(E1 \subset E2) \Rightarrow (E1 \subset E3)$ if E3 is an EL for $\tau(E1)$ and $\tau(E2)$ and is exclusively contained in $\tau(E2)$, $free(E2)$, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

R19. $(E1 \supset E2) \Rightarrow (E1 \supset E3)$ if E3 is an EL for $\tau(E1)$ and $\tau(E2)$ and obligatorily contains $\tau(E2)$, $free(E2)$, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

R20. $(E1 \subset E2) \Rightarrow (E1 \subset E3)$ if $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$ and is exclusively contained in $\tau(E3)$, $free(E2)$, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

R21. $(E1 \supset E2) \Rightarrow (E1 \supset E3)$ if $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$ and obligatorily contains $\tau(E3)$, $free(E2)$, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

In the above rules, the *free(E)* condition is used to denote that the evaluation of the expression E is required to return the *full extent* of type $\tau(E)$. *free(E)* trivially holds for every $e \in n$, which is the primary intended use of this predicate. With nontrivial expression E (i.e., a non $e \in n$ expression), database statistics can be used to quickly determine whether the predicate holds without physically accessing the database.

If introducing a structure index into a query is impossible, our approach switches to a less favorite choice – that is, merely reducing the navigation paths required by a query. This kind of rules are relatively less interesting, especially when efficient structural join algorithms (e.g., (Chien, 2002; Srivastava, 2002)) are available for implementing the general containment operations.

It is worth to mention that there may exist alternatives to the new element type E3 interpolated by the above rules into a query (as an entrance location). In order to retain our framework deterministic under all circumstances, we developed an algorithm that searches the optimal entrance location for these rules. This algorithm is discussed in detail in (Che, 2006).

Simplification

The third phase, simplification, re-invokes most of the simplification rules covered by Phase 1. This is necessary because, after major transformations being performed for semantic optimization (the second phase), new redundancies may be introduced, especially when new element types (as entrance locations) are interpolated. The rules that enforce DTD-constraints and reorder operators do not need to be called again as semantic optimization does not introduce new inconsistencies with the DTD-constraints nor causes major structural changes in the queries. Besides, at this point we expect a thorough clean-up on the optimized queries, and for this purpose, we add two new types of rules to the third phase.

Firstly, the following rules are added to remove the \cap operations that may be reintroduced together with a structure index operation:

R22. $(I_{\tau(E1)}(E2) \cap E1) \Rightarrow (I_{\tau(E1)}(E2))$ if $free(E1)$

R23. $(I_{\tau(E1)}(E2) \cap \sigma(E1)) \Rightarrow \sigma(I_{\tau(E1)}(E2))$ if $free(E1)$

Secondly, towards thorough clean-up, we add the following rules that carry out deep clean-up by exploiting relevant structure knowledge (as characterized by the three notions: exclusivity, obligation, and entrance location). A few such rules as examples are given below:

R24. $E1 \subset E2 \Rightarrow E1$ if $\tau(E1)$ is exclusively contained in $\tau(E2)$

R25. $(E1 \subset (E3 \subset E2)) \Rightarrow (E1 \subset E2)$ if $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$, and $\text{free}(E3)$

R26. $(E1 \subset (E3 \supset E2)) \Rightarrow (E1 \supset E2)$ if $\tau(E1)$ is an EL for $\tau(E3)$ and $\tau(E2)$ and obligatorily contained in $\tau(E3)$, and $\text{free}(E3)$

An Optimization Example

In the following, we show the transformations applied to the example query introduced in Section 3:

$(\text{keyword} \subset (\text{description} \subset (\text{open_auction} \supset \sigma_{a='personref, r='123'}(\text{bidder}))))$

\Rightarrow (by R14, i.e., \subset associativity)

$((\text{keyword} \subset \text{description}) \subset (\text{open_auction} \supset \sigma_{a='personref, r='123'}(\text{bidder})))$

\Rightarrow (by R15, i.e., \subset commutativity)

$((\text{keyword} \subset (\text{open_auction} \supset \sigma_{a='personref, r='123'}(\text{bidder}))) \subset \text{description})$

\Rightarrow (by R13, i.e., index introduction)

$(I_{\text{keyword}}(\text{open_auction} \supset \sigma_{a='personref, r='123'}(\text{bidder})) \cap \text{keyword}) \subset \text{description}$

\Rightarrow (by R22: \cap deletion)

$(I_{\text{keyword}}(\text{open_auction} \supset \sigma_{a='personref, r='123'}(\text{bidder}))) \subset \text{description}$

\Rightarrow (by R23: simplification based on exclusivity)

$(I_{\text{keyword}}(\text{open_auction} \supset \sigma_{a='personref, r='123'}(\text{bidder})))$

The performed transformations eventually makes the structure index $I_{\text{keyword}}(\text{open_auction})$ interpolated and applied, and have the element type `description` completely removed from the query because of the identified exclusive containment property between `keyword` and `description`. As a result, the query expression (a logical plan) is considerably improved because the original form involves three containment operations, while the optimized one has only one containment operation plus a structure index. Without first performing the accomplished transformations, the structure index between `open_auction` and `keyword`, although available and relevant, would not be eventually applied to this query.

SUPPORT FOR PURE AND NEGATED CONTAINMENTS

While structural joins (Zhang, 2001; Srivastava, 2002) have been widely recognized as important primitives for evaluating the containment operations that are at the core of XML queries, there are aspects of containment operations that necessitate specialized algorithms. This important issue however does not seem to receive sufficient attentions from other researchers. Pure and negated containments represent two special but important cases of containment operations.

Structural joins are proper for the situation when both operands are required for output or for serving a subsequent operation during query evaluation. Nevertheless,

this does not cover all the cases that the structural relationships (typically via containment operations) need to be involved in an XML query.

A rather common situation is when a query (or a sub-query) simply asks for qualified elements from one operand, for example, “find all the `items` listed in `Australia`”, assuming the XMark dataset (Schmidt, 2002) is used. This query can be formatted as the following ePAT expression: `item ⊆ Australia`. This query involves a pure containment operation that needs to return only the elements from the first operand, while the second operand in this query is used only to help screen the elements from the first operand. This query, as consisting of a pure containment operation, is thus referred to as a *pure containment query*. If, instead, a general structural join as defined in (Zhang, 2001; Srivastava, 2002) is used for answering this query, the query plan shall consist of the following three nontrivial steps: performing the structural join, projecting the result to the left operand, and eliminating duplicates from result set.

Alternatively, if the query engine calls for a specialized algorithm that directly computes the pure containment semantics of \subset as defined in ePAT (see the last section), the evaluation plan of this query then comprises just a single evaluation step: i.e., the invocation of this specialized containment algorithm (which may also be called a *pure containment join* or *containment join* for short). Now, an arising question is: will this single algorithm be more efficient than the previous three-step evaluation plan? The answer is an absolute yes – in fact, this pure containment join is much cheaper than a corresponding structural join alone, not to mention the other two extra steps required by the first plan that uses a general structural join.

Containment joins come in two different forms. In our work, the one that computes \subset is called ICJoin (the “Is-Contained” Join), and the one that computes \supset is called CJoin (the “Contains” Join). The pseudo code of the two algorithms are respectively given in Fig. 2 and 3. In these algorithms, we adopt the same numbering scheme for elements as in (Srivastava, 2002), i.e., (`DocId`, `StartPos` : `EndPos`, `LevelNum`), which uniquely encodes the doc-id, the start and end position of an element, and the nesting level of the element in the DOM tree of the source document. With both algorithms, the AList (a list of potential ancestors) and DList (a list of potential descendants) are scanned just once, and there is no need to use any stacks or queues (this is in contrast to the tree-merge and stack-tree algorithms presented in (Srivastava, 2002)).

Algorithm ICJoin (DList, AList)

/ the algorithm computes the result of “ $D \subset A$ ” */*

input: DList of type D, and AList of type A; both in sorted order of `startPos` and all elements having the same `docId`

output: OList holding a sub list of DList that satisfy “ $D \subset A$ ”

begin

```
1. a = AList->firstNode; OList = NULL;
2. for (d = DList -> firstNode; d != NULL; d = d -> nextNode) {
3.   while (a.end < d.end && a->hasNextNode()) a = a->nextNode;
4.   /* Now find the first possible descendant d of node a: */
5.   if (a.begin < d.begin && a.end > d.end) add d to OList
6. } /* end of for loop */
```

end

FigureE2: The ICJoin algorithm calculating “ $D \subset A$ ”

Algorithm CJoin (AList, DList)

/* the algorithm computes the result of “ $A \supset D$ ” */

input: AList of type A, and DList of type D; both in sorted order of startPos and all elements having the same docId

output: OList holding a sub list of AList that satisfy “ $A \supset D$ ”

begin

1. $d = \text{DList} \rightarrow \text{firstNode}$; $\text{OList} = \text{NULL}$;
2. **for** ($a = \text{AList} \rightarrow \text{firstNode}$; $a \neq \text{NULL}$; $a = a \rightarrow \text{nextNode}$) {
3. **while** ($d.\text{begin} < a.\text{begin}$ && $d \rightarrow \text{hasNextNode}()$) $d = d \rightarrow \text{nextNode}$;
4. /* find the first possible descendant d of node a */
5. **if** ($a.\text{end} > d.\text{end}$ && $a.\text{begin} < d.\text{begin}$) add a to OList
6. } /* end of for loop */

end

Figure 3: The CJoin algorithm calculating “ $A \supset D$ ”

Both algorithms have the time complexity of $\Theta(|\text{AList}| + |\text{DList}|)$, while all the structural join algorithms proposed as in the papers (Zhang, 2001; Srivastava, 2002) have the complexity of $O(|\text{AList}| + |\text{DList}| + |\text{OutputList}|)$, where $|\text{OutputList}|$ alone can be the scale of $|\text{AList}| * |\text{DList}|$, which is far larger than $|\text{AList}| + |\text{DList}|$. The performance advantage of the pure containment joins over general structural joins thus becomes obvious. This superiority gets further amplified when the extra operations (which may be required if a general structural join is used instead) are taken into account, as illustrated by the example given at the beginning of this section.

From our specialized containment join algorithms (see Fig. 2 and 3), we further derive the negated counterparts for the operations \Subset and \supseteq , respectively. These two algorithms are shown in Fig. 4 and 5, correspondingly.

As can be seen, NICJoin and NCJoin both reverse the logic of their non-negated counterparts. That is, for example, whenever a qualified descendant is identified by ICJoin, it is excluded from output by NICJoin. NICJoin and NCJoin share the same time complexity as their counterparts. Applying negated containment joins to negated containment queries (i.e., queries involving negated containments like the example that is to be addressed at the end of this section) is yet another persuasive case demonstrating the advantage of resorting to specialized containment joins.

Algorithm NICJoin (DList, AList)

/* the algorithm computes the result of “ $D \Subset A$ ” */

input: DList of type D, and AList of type A; both in sorted order of startPos and all elements having the same docId

output: OList holding a sub list of DList that satisfy “ $D \Subset A$ ”

begin

0. $\text{OList} = \text{DList}$;
1. $a = \text{AList} \rightarrow \text{firstNode}$; $\text{OList} = \text{NULL}$;
2. **for** ($d = \text{DList} \rightarrow \text{firstNode}$; $d \neq \text{NULL}$; $d = d \rightarrow \text{nextNode}$) {

```

3.   while (a.end < d.end && a->hasNextNode()) a = a->nextNode;
4.   /* Now find the first possible descendant d of node a: */
5.   if (a.begin < d.begin && a.end > d.end) remove d from OList
6. } /* end of for loop */
end

```

Figure 4: The NICJoin algorithm calculating “ $D \Subset A$ ”

Algorithm NCJoin (AList, DList)

/* the algorithm computes the result of “ $A \Subset D$ ” */

input: AList of type A, and DList of type D; both in sorted order of startPos and all elements having the same docId

output: OList holding a sub list of AList that satisfy “ $A \Subset D$ ”

begin

```

0. OList = AList;
1. d = DList->firstNode; OList = NULL;
2. for (a = AList -> firstNode; a != NULL; a = a -> nextNode) {
3.   while (d.begin < a.begin && d->hasNextNode()) d = d->nextNode;
4.   /* find the first possible descendant d of node a */
5.   If (a.end > d.end && a.begin < d.begin) remove a from OList
6. } /* end of for loop */

```

end

Figure 5: The NCJoin algorithm calculating “ $A \Subset D$ ”

In (Che, 2005), two alternative algorithms (i.e., NMPMJIN and NSTD) for computing negated containments were presented that are based on direct modification on the two well-known structural join algorithms, MPMJIN in (Zhang, 2001) and Stack-Tree-Desc in (Srivastava, 2002), respectively. NMPMJIN and NSTD were aligned to the same style and numbering scheme adopted by Stack-Tree-Desc. In the following, we present an alternated version of NMPMJIN (Che, 2005), which is straightforwardly revised from MPMJIN for ease of comparison. We refer to this algorithm as ANMPMJIN.

ANMPMJIN is the algorithm that we implemented in our test-bed and for the experimental study to be reported in Section 6 (and MPMJIN is the counterpart structural join algorithm adopted by our test-bed). ANMPMJIN follows exactly the same logic as MPMGIN (Zhang, 2001). It accepts two sorted lists of items as input and outputs a list of items (usually a sub list of the first input list) that do not have a containment relationship with any item from the second input list. ANMPMJIN has two levels of nested loops (while MPMGIN has three). The worst-case time complexity of ANMPMJIN is $O(n^2)$ (comparing with $O(n^3)$ of the original MPMGIN).

Finally, we look at an example query that has a negated containment operation – a so-called negated containment query. Assume this query asks for `open_auctions` that have not received a bid, which is formatted as “`open_auction - (open_auction \supset bidder)`”. A reasonable plan (an ePAT expression) for this query is “`open_auction - $\pi_{open_auction}$ (open_auction \times_{\supset} bidder)`”, which means a structural join algorithm, e.g., MPMGIN, is to be called first for computing the operation \times_{\supset} ; then a leftward

projection is invoked for computing $\pi_{\text{open_auction}}$; after that, duplicates are eliminated, and finally the difference is computed. This plan consists of four steps. A much better alternative plan is “open_auction \equiv bidder”, assuming dedicated algorithm like ANMPMJIN for computing the “ \equiv ” operation is implemented. The superiority of the second plan is obvious if we consider that \equiv is the only operation needs and its implementation (e.g., ANMPMJIN) is more efficient than MPMGIN alone ($O(n^2)$ vs. $O(n^3)$), not to mention the other three extra operations remained in the first evaluation plan, which all come with a nontrivial cost.

Algorithm ANMPMJIN (list1, list2)

begin

1. copy list1 to list3
 2. set cursor1 at beginning of list1
 3. set cursor2 at beginning of list2
 4. set cursor3 at beginning of list3
 5. **while** (cursor1 \neq end of list1 and
 6. cursor2 \neq end of list2) **do**
 7. **if** (cursor1.docno < cursor2.docno) **then**
 8. cursor1++; cursor3++
 9. **else if** (cursor2.docno < cursor1.docno) **then**
 10. cursor2++
 11. **else**
 12. mark = cursor2
 13. **while** (cursor2.position < cursor1.position and
 14. cursor2 \neq end of list2) **do**
 15. cursor2++
 16. **if** (cursor2 == end of list2) **then**
 17. cursor1++; cursor3++
 18. cursor2 = mark
 19. **else if** (cursor1.val (directly)contains cursor2.val) **then**
 20. remove cursor0.val
 21. cursor1++
 22. **endif**
 23. **endwhile**
 24. **endif**
 25. **endwhile**
 26. output list3
- end**

Figure 6: ANMPMJIN: a straightforward counterpart of MPMGIN for negated containments

EXPERIMENTS

In order to evaluate our new optimization approach that has integrated support for pure and negated containments, we conducted an experimental study. As our new approach is adapted from the deterministic optimization approach we presented previously in (Che, 2006), and the effectiveness, efficiency, and scalability of the general approach has been inherited. Herein, we only show the performance advantage of this newly adapted approach as compared with the prior version of the

approach (Che, 2006).

Our prototype system is implemented in the Java programming language and the Oracle RDBMS is used as a fast platform for storage management (however, our optimization approach is platform independent per se). Query processing in our test-bed consists of the following steps: translation from XQuery to ePAT expressions, logical optimization performed on ePAT expressions, translation from optimized ePAT expressions to Oracle SQL, and SQL query execution by Oracle query engine.

Our test-bed is set in a typical 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 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.

For this test, the benchmark databases and queries are the same as in (Che, 2006). Here we present the result of our test with only Database 1, which is a synthesized one based on a rather practical DTD (Che, 2006) for conference proceedings. The characteristics of this database are shown in Table 1 (The database was populated at different scales; Table 1 shows the statistics at the representative scales, 1, 5, and 10). The benchmark queries and their selectivities are shown in Table 2.

Scale	Size (MB)	#Docs	#Elements	Max degree	Min degree	Depth	Max cardn.	Min cardn.
1	28.93	100	1,206,224	20	1	6	421,681	100
5	138.80	500	5,763,581	20	1	6	2,125,564	500
10	252.40	1000	11,546,873	20	1	6	3,695,219	1000

Table 1: Characteristics of Database 1 at scale 1, 5, and 10 (*cardn.* stands for cardinality)

The performance data (obtained when the database was populated at scale 10) are illustrated in Fig. 7. From Fig. 7, one can see that the new approach with dedicated support for pure and negated containments outperform the old approach by several times. This is true for both optimized and non-optimized queries. This observation further means that the performance gain of the new approach is mainly from the contribution of the specialized containment join algorithms integrated within the new approach; these specialized algorithms (as additional primitives) can be broadly and separately applied to other XML querying systems.

	Queries (formatted in XPath)	Select	Optimize
Q1	//Article[./Title ftcontains "Data Warehousing" or ./Keywords ftcontains "Data Warehousing"]	2.0%	0.192
Q2	//Article[./Title ftcontains "Data Warehouse"]/Abstract	1.1%	0.153
Q3	//Articles[./Title ftcontains "Data Warehousing"]/Sections/Section[@title="Introduction"]/Paragraph	6.4%	0.769
Q4	//Article[./Surname ftcontains "Aberer"]	2.3%	0.737
Q5	//Section/Paragraph [@title = "Summary"]	1.8%	0.841
Q6	/(article ShortPaper) // Paragraph[.ftcontains "Multidimension" or .ftcontain "OLAP"]	1.1%	2.221

Table 2: Database 1 benchmark queries and properties (the *select* column indicates the selectivity of queries and the *optimize* column records the optimization time in seconds)

Finally, before concluding this section, we would like to point out: analytically, we believe ICJoin and CJoin (and their negated counterparts, NICJoin and CJoin) shall bring more improvement on the query performance because they all have a linear time complexity as shown in the last section (at this time our experiment is based on ANMPMJIN and NMPMJIN only).

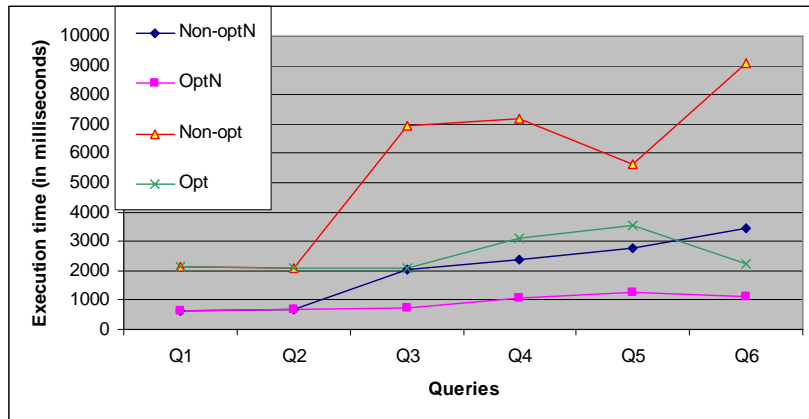


Figure 7: Performance comparison: new approach with special containment join support vs. previous approach without special support (Opt, Non-opt, OptN, and Non-optN stand for optimized and non-optimized queries in regard to our previous and new adapted approach, respectively)

RELATED WORK

Now we briefly review some related work.

Lore (McHugh, 1997; McHugh, 1999) is a DBMS originally designed for semi-structured data and later migrated to XML-based data model. The Lore optimizer is cost-based and does not perform logic-level optimization. The work in (Consens, 1994) is one of the early works reported on using the DTD knowledge on query transformations. The DTDs considered there are much simpler than that of XML. In (Fernandez, 1998), a comparable strategy for exploiting a grammar specification for query optimization over semi-structured data is studied. In that study, efforts were

made on how to make a complete use of the available grammar to expand a query. Our focus is different – we identify transformations that introduce “guaranteed” improvements on query expressions from a heuristic point of view. In (Chung, 2002), a fresh idea about using the DTD of XML documents to guide the query processor’s behavior was proposed, where, the same type of elements (corresponding to the same node in the DTD-graph) is further classified according to the diverging paths from the type node. Query processing is then guided to a more specific class of the elements of this type to prune the search space. This classification information can effectively guide the query engine to narrow the search space and speed up query evaluation. But with relatively complex DTD, too many classes may be produced. In (Wang, 2003), based on the notion of rooted paths, two types of optimization were proposed: path complementing (i.e., if the path in a path query has a complement and its evaluation is cheaper, then the complement substitutes for the original path) and path shortening (i.e., if the head segment of a path is the unique one that reaches the ending point of the segment, then this segment is removed). The path shortening idea is similar to ours (Che, 2005; Che, 2006). However, our algorithm is more general and may identify more opportunities of reducing a path (which in our approach does not need to be rooted). In (Buneman, 1999), path constraints were used to convey the semantics of semi-structured data and applied to query optimization. It is worth noticing that there are two types of semantics with regard to XML data: *data semantics* (i.e., the meanings of the data) and *structure semantics* (i.e., the knowledge about the general structure or structural relationships among the data elements). The path constraints investigated in (Buneman, 1999) is restricted to only data semantics. Our approach exploits structural semantics. Path and tree pattern matching build the core of XML query processing. A bundle of approaches (Bremer, 2003; Srivastava, 2002; Zhang, 2001; Li, 2001; Grust, 2002) have been proposed for supporting path and tree pattern matching, commonly known as structural joins. These approaches focus on composing the path/tree query patterns node-by-node through pair-wise matching of ancestor and descendant or parent and child nodes. These approaches have performance advantages over the simple path navigation method and have been integrated with our deterministic optimization approach in our prototype implementation.

Distinctively, we provide direct support for pure and negated containments (which are common in XML queries), and developed a family of specialized supporting algorithms, which showed performance potential.

To the best of our knowledge, our work as reported in (Che, 2006) and in this paper is the only one that uses algebraic transformations extensively to exploit structural properties and other important optimization heuristics for XML query optimization, and our work is the only one that integrates special containment join algorithms with a deterministic optimization approach.

SUMMARY

In this paper, we presented an innovative, logic-level optimization approach which is particularly adapted for dealing with XML queries that may contain special containments such as pure and negated containments. Regular structural join algorithms cannot be efficiently applied to this kind of queries. The uniqueness of

our work lies in, first, we applied a deterministic approach for XML query optimization, which shows great potential for improved optimization performance; second, we proposed dedicated algorithms for special containment operations (i.e., pure and negated containments) in the context of XML query processing. Our experimental study confirmed the validness and performance advantage of the presented new approach and the proposed special containment join algorithms.

REFERENCES

- Böhm, K., Aberer, K., Özsu, T. M., & Gayer, K. (1998). 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, 196-205.
- Böhm, K., Aberer, K., Neuhold, E. J., & Yang, X. (1997). Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. The VLDB Journal. 6 (4), 296-311.
- Boag, S., Chamberlin, D., Fernandez, M. F., Florescu, D., Robie, J., & Simeon, J. (2006). XQuery 1.0: An XML Query Language (<http://www.w3.org/TR/xquery/>).
- Bremer, J.-M. & Gertz, M. (2003). An Efficient XML Node Identification and Indexing Scheme. University of California at Davis. Technical Report, 2003 (http://www.db.cs.ucdavis.edu/papers/TR_CSE-2003-04_BremerGertz.pdf).
- Buneman, P., Fan, W. & Weinstein, S. (1999). Query Optimization for Semistructured Data Using Path Constraints in a Deterministic Data Model. Proc. of DBPL Conf., 1999, 208-223.
- Chan, C. Y., Felber, P., Garofalakis, M. & Rastogi, R. (2002). Efficient Filtering of XML Documents with XPath Expressions. Proc. of Int. Conf. on Data Engineering, San Jose, California, February 2002, 235-244.
- Chan, C. Y., Garofalakis, M. N. & Rastogi, R. (2002). RE-Tree: An Efficient Index Structure for Regular Expressions. Proc. of VLDB Conf., Hong Kong, China, 2002, 263-274.
- Che, D. Accomplishing Deterministic XML Query Optimization. Journal of Computer Science and Technology. 20(3), 357-366.
- Che, D. Efficiently Processing XML Queries with Support for Negated Containments. Int. Journal of Computer & Information Science, 6(2), 119-120.
- Che, D. Implementation Issues of a Deterministic Transformation System for Structured Document Query Optimization. Proc. of the Seventh Int. Database Engineering and Applications Symposium, Hong Kong, July 16 -18, 2003, 268-277.
- Che, D., Aberer, K. & Özsu, M. T. (2006). Query Optimization in XML Structured-Document Databases. The VLDB Journal, 15(3): 263-289.
- Chien, S. Y., Vagena, Z., Zhang, D., Tsostras, V. J. & Zaniolo, C. (2002) Efficient Structural Joins on Indexed XML Documents. Proc. of VLDB Conf., Hong Kong, August 20-23, 2002, 263-274.

- Chung, T.-S. & Kim, H.-J. (2002) XML query processing using document type definitions. *Journal of Systems and Software* 64(3): 195-205.
- Clark, J. & DeRose, S. (1999). XML Path Language (XPath) Version 1.0 (<http://www.w3.org/TR/1999/REC-xpath-19991116>).
- Consens, M. & Milo, T. (1994). Optimizing Queries on Files. *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, Minnesota, May 1994, 301-312.
- Fernandez, M. F. & Suciu, D. (1998). Optimizing Regular Path Expressions Using Graph Schemas. *Proc. of the Fourteenth Int. Conf. on Data Engineering*, Orlando, Florida, USA, February 23-27, 1998, 14-23.
- Frasincar, F., Houben, G. J. & Pau, C. (2002) XAL: an Algebra for XML Query Optimization. *Proc. of the 13th Australasian Database Conf., (ADC2002)*, 49-56.
- Gottlob, G., Koch, C. & Pichler, R. (2002). Efficient Algorithms for Processing XPath Queries. *Proc. of VLDB Conf., Hong Kong, China, 2002*, 95-106.
- Graefe, G. & DeWitt, D. (1987). The EXODUS optimizer generator. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, May 1987, 160-172.
- Grust, T. (2002). Accelerating XPath location steps. *Proc. of ACM SIGMOD Conf., 2002*, 109-120.
- Guha, S., Jagadish, H. V., Koudas, N., Srivastava, D. & Yu, T. (2002). Approximate XML joins. *Proc. of the ACM SIGMOD Conf., 2002*, 287-298
- Jagadish, H.V., Lakshmanan, L.V.S., Srivastava, D. & Thompson, K. (2001). TAX: A Tree Algebra for XML. *Proc. of DBPL Conf., Rome, Italy, 2001*, 149-164.
- Li, Q. & Moon, B. (2001). Indexing and Querying XML Data for Regular Path Expressions. *Proc. of VLDB Conf., Rome, Italy, September 2001*, 361-370.
- McHugh, J. & Widom, J. (1999). Query Optimization for XML. *Proc. of VLDB Conf., Edinburgh, Scotland, September 1999*, 315-326.
- McHugh, J., Abiteboul, S., Goldman, R., Quass, D. & Widom, J. (1997). Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3), 54-66.
- Milo, T. & Suciu, D. (1999). Index Structures for Path Expressions. *Proc. of ICDT, 1999*: 277-295.
- Salminen, A. & Tompa, F. W. (1994). PAT Expressions: an Algebra for Text Search. *Acta Linguistica Hungarica*, 41(1), 277-306.
- Schmidt, A. R., Waas, F., Kersten, M. L., Carey, M. J., Manolescu, I. & Busse, R. (2002). XMark: A Benchmark for XML Data Management. *Proc. of VLDB, Hong Kong, China, August 2002*, 974-985.
- Selinger, P. G., Astrahan, M. M., Chamberlin, D., Lorie, R. & Price, T. (1979). Access Path Selection in a Relational Database Management System. *Proc. of ACM SIGMOD Conf., 1979*, 23-34.
- Srivastava, D., Al-Khalifa, S., Jagadish, H. V., Koudas, N., Patel, J. M. & Wu, Y. (2002). Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *Proc. of ICDE Conf., San Jose, CA, 26 February - 1 March 2002*, 141-152.
- Wang, G. & Liu, M. (2003). Query Processing and Optimization for Regular Path Expressions. *Proc. of CAiSE Conf., Klagenfurt, Austria, June 16-18, 2003*, 30-45.

Zhang, C., Naughton, J. F., DeWitt, D., J., Luo, Q. & Lohman, G. M. (2001). On Supporting Containment Queries in Relational Database Management Systems. Proc. of ACM SIGMOD Conf., Santa Barbara, CA, USA, 2001, 425-436.