

# Efficiently Processing XML Queries with Support for Negated Containments

Dunren Che<sup>1</sup>

Southern Illinois University, U.S.A

## Abstract

Efficient processing of XML queries is of utmost importance nowadays not only for XML databases but also for many current and future applications that need to tap information from large volumes of XML data sources. Querying XML data often requires, explicitly or implicitly, examining *negated* structural relationships such as negated containments, in addition to other structural relationships. In this paper, we focus on efficient processing of a class of XML queries involving negated containments, called *negated containment queries*. This issue has not been addressed before by the research community. We show that, in order to process negated containment queries efficiently, specialized structural join algorithms can be used to provide a direct and efficient support on this purpose; furthermore, a collaborating logical query optimizer is very important since it can substantially facilitate the discovery of potential opportunities for beneficially applying these algorithms. In this paper, we first present a logical optimization approach that performs deterministic transformations on XML queries for optimization and for identifying the opportunities of applying these specialized algorithms to a query. We then propose two such specialized structural join algorithms for negated containments. Preliminary experimental results will be presented that further confirm the validity of the proposed approach.

**Keywords:** XML query processing, XML query optimization, Deterministic transformation, negated containment.

## 1. Introduction

Efficient processing of XML queries is of utmost importance nowadays not only for XML databases but also for many current and future applications that need to tap information from large volumes of XML data, for example, web search engines, online digital libraries, e-commerce and digital government, etc. The volumes of XML data generated in various application areas are vastly mounting up. This phenomenon calls upon highly efficient and specialized management and querying solutions for XML. Among the many challenges facing XML database management, XML query optimization is especially interesting because it is not only a conventional key DBMS issue but also a key infrastructure for tomorrow's semantic-based web search engines.

Query optimization typically adopts a cost-based approach

[28] and aims at obtaining the least expensive — the optimal — evaluation plan for each input query. Heuristic knowledge may be developed and applied so that only a reduced number of (highly potential) candidate plans need to be examined and the “optimal” can be quickly determined.

XML data has the semi-structured nature and XML queries need to examine not just the contents but also the structural patterns of the target data. Comparing with relational data, XML data has higher complexity (as it has to deal with both the structural part and the content part) and this raised complexity straightforwardly translates to a much enlarged search space from which the “optimal” plan is to be decided during query optimization. As a result, the cost-based optimization approach very likely will not yield the same good efficiency for XML queries as it can for relational queries. On the other hand, apart from adding extra complexity and causing inefficient query evaluation, the structural part of XML data nonetheless implies a rich source of knowledge that can be favorably applied to the optimization of XML queries. We are thus motivated to develop a comprehensive optimization framework for XML queries. This framework consists of two separate, yet, collaborating stages for XML query optimization. The first stage performs *logical optimization*. This stage is unique to XML queries as it exploits the unique features (e.g., semantic knowledge) of XML data for query optimization. By its nature, the first stage is strongly heuristic-based — it does not rely on any specific knowledge of the underlying data and storage structure. The second stage, called *physical optimization*, applies specialized cost-based optimization techniques. In this optimization framework, the two stages need to collaborate in such a way that the first stage provides a much reduced set of improved (or pre-screened) logical plans to the second stage and the second stage, while conducting cost-based optimization, shall apply highly specialized techniques owing to its “awareness” of the optimization that the first stage has already done.

In this paper, we show that when strong heuristics (which are greatly facilitated by the rich structure-related semantics available in the context of XML data) are applied, logical optimization can be designed to exclusively apply *deterministic* transformations on XML queries. More importantly, potential opportunities of applying specialized structural join algorithms for efficient evaluation of negated structural relationships can be expediently identified via logical optimization. Often, logical optimization alone may render significant improvement on most queries in terms of their evaluation efficiency. So, when the optimal plan is not a rigid goal, a logical optimizer can be conveniently deployed as a standalone optimizer for accelerating XML query execution. An additional advantage of undertaking mere logical optimization is that the resultant op-

---

<sup>1</sup>Department of Computer Science  
Carbondale, IL 62901, USA  
dche@cs.siu.edu

imizer is highly adaptable since it does not depend on any implementation-related and/or platform-specific features.

The main theme of this paper is on logical XML query optimization and this is deliberately tailored for efficient processing of negated containment queries. In our previous work [11], equivalences for XML query optimization had been studied, and a comprehensive methodology for XML query optimization at the logical-level was also proposed there. According to that methodology, heuristic-based *deterministic* transformation rules are developed and applied to XML query expressions represented as PAT algebraic expressions [26, 3]. The utmost benefit of applying deterministic transformations on XML queries is the great potential for superb optimization efficiency. Previously, PAT [26] was used as the optimization algebra in our work [9, 10, 11, 3]. In this paper, we present an extended version of the PAT algebra, called ePAT. Based on that, we develop new equivalences and new deterministic transformation rules for query optimization. To make this paper focused on its theme — efficient processing of negated containment XML queries — we shall present only sample equivalences and sample deterministic transformation rules. We will focus on tapping the potential of direct algorithmic support for negated structural relationships, especially, negated containments. Negated containment queries are common, e.g., “find employees who do not have a homepage”, and thus are very important, but has not been sufficiently addressed by the research community. Our new optimization approach is meant to provide adequate support for negated containment queries in two steps. First, a comprehensive logical optimizer is used to identify potential opportunities that a specialized algorithm for negated containments can be applied. Second, at the execution phase, specialized join algorithms are called on to fast evaluate the negated containment operations involved.

In summary, we make the following major contributions in this paper:

- We present a new deterministic approach for XML query optimization: it identifies potential opportunities so that a specialized algorithm can be called on for negated containment operations.
- We develop specialized algorithms for direct support of negated containment operations.
- Our work substantially enriches the family of structural joins so that efficient algorithmic support can be directly provided also to negated structural relationships. To the best of our knowledge, our work is the first and only one that addresses the issue of efficient optimization and execution of queries involving negated containments.

The remaining of this paper is set forth as follows: Section 2 provides some background knowledge and a sketchy overview of our new optimization approach. Section 3 illustrates our overall approach via presentation of sample equivalences and deterministic transformation rules. Section 4 addresses the structural join algorithms specialized for negated containments. Section 5 presents preliminary experimental results, while related work is discussed in Section 6. The paper is finally concluded in Section 7.

## 2. Preliminaries

Our work addresses XML documents and our approach applies deterministic transformations on algebraic query expressions. In this section, first, we shed lights on several important notions related with XML; then, we present a limited version of the ePAT (extended PAT) algebra, which is adopted in our new approach; later on, we review sample equivalences that serve as the theoretical basis of our entire approach; and finally, we provide a sketchy overview of our approach.

### 2.1 Notions regarding XML document structure

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 can serve basically the same function although the latter is more powerful. In this paper, we base our presentation on the DTD notion. Nevertheless, our approach shall straightforwardly extend to the situation where XSD is used instead.

A tag (with a distinct name) defined in a DTD corresponds to an element type, which may comprise some other tags in its *content model*. The constructional relationships among the element types are a source of knowledge that can be used to pursue semantic query optimization. We introduce a graph, called *DTD-graph*, to help envision the important structural relationships among the element types. As an example, the DTD-graph in Figure 1 illustrates some structural relationships among the data elements (of different types) related to *open\_auctions* as specified in the XMark benchmark [27]. In the figure, an available *structure index* is additionally shown using a dashed line with arrows at both ends (A structure index is formally defined as a map between the *extents* of the two element types[11]. A structure index may be intuitively conceived as a “shortcut” between two the types of elements, and, in our system, it is implemented as a materialized or pre-computed structural join [31, 29]).

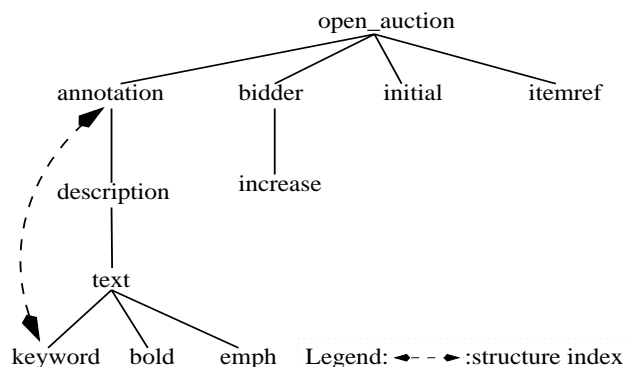


Figure 1. A portion of the XMark benchmark DTD-graph with index info.

*Containment* is the dominant structural relationship in XML and is at the core of both XPath [13] and XQuery [1]. 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 linked one after another. Paths are the primary means used in both XQuery[1] and XPath[13] to capture the structural pattern of XML data.

In addition to the notions of DTD-graph and path, we adopt three other important notions to help capturing particularly interesting structural relationships among XML data. These notions are reviewed below.

**Definition 2.1 (Exclusivity)** *Element type  $ET_j$  is exclusively contained in element type  $ET_i$  if each path  $(e_j, \dots, e_k)$  with  $e_j$  being an element of type  $ET_j$  and  $e_k$  being the document root contains an element of type  $ET_i$ .*

**Definition 2.2 (Obligation)** *For a given DTD, element type  $ET_i$  obligatorily contains element type  $ET_j$  if each element of type  $ET_i$  has to contain an element of type  $ET_j$  in any document complying with the DTD.*

**Definition 2.3 (Entrance location)** *For a given DTD, element type  $EL$  is an entrance location for type  $(E1)$  and type  $(E2)$  if in any document complying with the DTD, all paths from an element  $e1$  of type  $(E1)$  to an element  $e2$  of type  $(E2)$  pass through an element  $el$  of type  $EL$ .*

## 2.2 The ePAT algebra for XML query optimization

It is typical to represent queries as algebraic expressions internally for an optimizer to expediently explore alternate evaluation plans based on equivalent transformations. The algebra adopted by our approach is the so-called ePAT (extended PAT).

An ePAT expression is generated according to the following grammar:

$$E ::= etn \mid (E) \mid E1 \cup E2 \mid E1 \cap E2 \mid E1 - E2 \mid \sigma_r(E) \mid \sigma_{a,r}(E) \mid E1 \subset E2 \mid E1 \supset E2 \mid E1 \not\subset E2 \mid E1 \not\supset E2 \mid E1 \bowtie_c E2 \mid \pi_{pl}(E) \mid I(E) \mid -E$$

“E” (as well “E1” and “E2”) stands for an ePAT expression. *etn*, as the only atomic expression, retrieves the whole extent (i.e., all the instances) of the element type named *etn*. The expression  $(E)$  produces the same result as  $E$ .  $\cup$ ,  $\cap$  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.  $\subset$  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.  $\not\subset$  and  $\not\supset$  are the negated form of  $\subset$  and  $\supset$ , respectively. Accordingly,  $\not\subset$  returns elements of the first argument that are *not* contained in any element decided by the second argument, and  $\not\supset$  returns elements of the first argument that do *not* contain any element of the second argument.  $\bowtie_c$  is a “power” join operation that can be tailored to almost any

specific form of joins such as structural joins[31, 29], unstructural joins<sup>2</sup>, and Cartesian products, based on the nature of the join predicate specified by the join condition  $c$ .  $\pi_{pl}$  is the projection operator that carries a parameter,  $pl$ , which designates a projection list.  $I$  connotes the application of a relevant index of any type (such as content indices of elements, value indices of attributes, and structure indices). Finally,  $-E$  is the “negated” form of  $E$  and here the  $E$  subexpression is limited to only a selection or a containment subexpression. When it is a selection, the negation is logically applied to the filtering predicate of the selection; when it is a containment, e.g.,  $E1 \supset E2$ , the negation is applied to the  $\supset$  operation (meaning “does not contain”). Note that  $\not\supset$  is not semantically equivalent to  $\subset$  by any means. We will show how specialized structural join algorithms can be devised to provide direct and efficient support for “negated containments”.

It is interesting to point out that in the ePAT algebra, the containment operations,  $\supset$  and  $\subset$ , are redundant — i.e., they can be represented as a structural join followed by a leftward projection. Yet, there are advantages of retaining this redundancy (in a similar spirit as retaining natural join in the relational algebra): considering that containments dominate the structural relationships in XML data, having dedicated operations for containments will greatly facilitate the exploitation of the structure knowledge of XML data for query processing and optimization.

We use the following query as a running example. First, we show how the query is formulated as an ePAT expression. Later in Section 3.4, we show how logical optimization is performed through consecutive deterministic transformations on ePAT expressions.

**Example.** Get 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.

**XPath:** `//open_auction[./bidder[@personref="123"]]/annotation/keyword`

**ePAT:**  $(keyword \subset (description \subset (open\_auction \supset \sigma_{a='personref', r='123'}(Bidder))))$

## 2.3 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 rules embody our optimization methodology, equivalences form the basis of our approach. In the following, we present sample equivalences.

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

<sup>2</sup>Unstructural joins are joins that are not based on checking any structural relationship.

It is worth pointing out that we could have obtained a very large set of equivalences, of which some may not be useful. Therefore, we set up strict criteria to choose only those highly potential equivalences from which *deterministic* transformation rules can be derived. For example, equivalences must *not* imply transformations that merely complicate/expand a query without improving the quality of the query in terms of evaluation efficiency (from a heuristic point of view). More detailed discussions on this can be found in [11].

Equivalences take the form “ $E1 \iff E2$ ”, while transformation rules take the form “ $E1 \implies E2$ ”. For brevity, we introduce the following generic operations: let “ $\cup$ ” stand for both  $\cup$  and  $\cap$ , “ $\sigma$ ” for both  $\sigma_r$  and  $\sigma_{a,r}$ , and “ $\supseteq$ ” for both  $\subset$  and  $\supset$ . When there is more than one occurrence of any of these generic operations, interpretation must be consistent, e.g., all occurrences of  $\cup$  must be all interpreted as  $\cup$  or  $\cap$ , but not mixed.

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

### Set-oriented equivalences

ePAT is a set based algebra. A group of equivalences were derived directly from the set-theoretic properties of relevant ePAT operators. A couple of such equivalences are below.

$$\begin{aligned} \mathcal{E}1. & (E1 \supseteq E2) \cap E3 \iff (E1 \cap E3) \supseteq E2 \\ \mathcal{E}2. & (E1 \supseteq E2) \supseteq E3 \iff (E1 \supseteq E3) \supseteq E2 \end{aligned}$$

$\mathcal{E}2$  is referred to as the **commutativity** law of the containment operations.

### Equivalences based on explicit DTD constraints

There are equivalences based on checking the explicit constraints given by the DTD/XSD. Two such examples are given below (the symbol  $\tau$  denotes the resultant type of an expression).

$$\begin{aligned} \mathcal{E}3. & \sigma_{a,r}(E) \iff \phi \text{ if 'A' isn't an attribute of } \tau(E) \\ \mathcal{E}4. & E1 \subset E2 \iff \phi \text{ if } \tau(E1) \text{ isn't contained in } \tau(E2) \end{aligned}$$

### Equivalences based on deep and implied knowledge about XML document structure

This is a far more interesting class of equivalences as they bear the potential of rendering significant improvement on input queries. We show a few examples below.

$$\begin{aligned} \mathcal{E}5. & (E1 \supseteq E2) \supseteq E3 \iff E1 \supseteq (E2 \supseteq E3) \text{ if } \tau(E2) \text{ is an} \\ & \text{EL for } \tau(E1) \text{ and } \tau(E3) \text{ (associativity)} \\ \mathcal{E}6. & (E1 \supseteq E2) \supseteq E3 \iff E1 \supseteq (E3 \supseteq E2) \text{ if } \tau(E3) \text{ is an} \\ & \text{EL for } \tau(E1) \text{ and } \tau(E2) \text{ (cossociativity)} \\ \mathcal{E}7. & E1 \subset E2 \iff E1 \subset (E3 \subset E2) \text{ if } E3 \text{ is an entrance} \\ & \text{location for } \tau(E1) \text{ and } \tau(E2) \end{aligned}$$

While  $\mathcal{E}5$  refers to the *associativity* of the containment operations,  $\mathcal{E}6$  embodies both commutativity and associativity in a single equivalence and we call it as the *cossociativity* law of the containment operations.  $\mathcal{E}7$  embodies the connotation of the *entrance location* concept. The combined application of this concept with the other two concepts (see Section 2.1) yields more potential equivalences, which play an important role in our approach [11].

## 2.4 Strategy Overview

We identify a large set of potential equivalences as the basis for XML query optimization and these equivalences are centered around the containment operations. However, equivalences are not directly used by our approach for query optimization (this is in contrast to the conventional approach [18] in which equivalences are directly used to enumerate alternative plans). Instead, we recognize the potential of XML document structure in XML query optimization and certain unique features of XML queries (e.g., always carrying a structural pattern), and are thus prompted to focus more on logical XML query optimization. Logically optimized queries can be complemented by a subsequent physical (cost-based) optimizer in case that highly optimized query plans are required, e.g., for complex and repeatedly-executed queries.

Generally, our approach is designed to explore the rich structure-related semantics of XML data for achieving fast XML query optimization. More specifically, we derive heuristic-based, deterministic transformation rules based on the identified equivalences in collaboration with various optimization heuristics which may check the availability of relevant structure indices. The transformation rules are applied to each query expression step-by-step, and each performed transformation is required to solidly improve an input query until no more transformations can be performed.

During this deterministic transformation process for query optimization, rules are arranged in a fixed order based on certain optimization heuristics. When there are multiple rules applicable to an input expression, the precedence is given to the first applicable rule in the list. More details of the rule control strategy and the transformation algorithms are provided in [10] (and also in [11]).

In this paper, we discuss the optimization issue of XML queries consisting of the main ePAT operators: *etn*,  $\cap$ ,  $\cup$ ,  $-$ ,  $\sigma$ ,  $\supseteq$ ,  $\subset$ . Our approach organizes logical optimization in three consecutive phases: normalization, semantic optimization, and simplification. The normalization phase enforces straightforward DTD/XSD constraints, performs straightforward simplification, and reorders operators in the following order (from bottom up): *etn*,  $\cap$ ,  $-$ ,  $\sigma$ ,  $\supseteq$ ,  $\cup$ . Interestingly, after normalization, all  $\cap$  and  $-$  operations sediment to the bottom of the operation tree of a query and ultimately get eliminated from the query.

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 a query. Due to the particular pattern that a query expression is in, structure indices may be available, but not applicable to the query. The central goal of the second phase is to reveal such opportunities.

The final phase, simplification, performs a thorough cleaning. For this purpose, an intensive set of simplification rules are needed (including many of the simplification rules that have been already applied by the normalization phase). Besides, several special rules are needed to handle the new redundancy resulted from semantic optimization.

A distinguishing feature of our approach is that it applies exclusively deterministic transformations on queries to achieve

fast optimization. Consequently, the typical problem that bothers many rule-based systems — i.e., the uncontrollable runtime behavior and runtime complexity — is expediently circumvented. Another important feature is that the two useful set operations,  $\cap$  and  $-$ , ultimately get eliminated from optimized queries. The third important feature of our approach is the direct support provided to negated containments for fast processing.

### 3. Transformation-Based XML Query Optimization

In this section, we present selected transformation rules and explain the heuristics behind the arrangement of our system.

#### 3.1 Normalization

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

##### 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 a query's consistency with the constraints explicitly induced by a given DTD/XSD. A couple of such rules are given below.

- R1.**  $\sigma_{a,r}(E) \implies \phi$  if 'A' is not an attribute of  $\tau(E)$   
**R2.**  $E1 \subset E2 \implies \phi$  if  $\tau(E1)$  is not contained in  $\tau(E2)$

##### Operator reordering

The objectives of operator reordering include: completely eliminate the  $\cap$  and  $-$  operations from a query; re-permute other operators in the following order:  $etn$  and  $\sigma$  at the bottom,  $\supseteq$  in the middle,  $\cup$  on the top. The reordered query will facilitate the subsequent semantic optimization achieving the primary goal — bringing a beneficial structure index into a query. Several example rules are given below.

- R3.**  $\sigma(E1 \cup E2) \implies \sigma(E1) \cup \sigma(E2)$   
**R4.**  $E1 - (E2 \supseteq E3) \implies E1 \cap (E2 \not\supseteq E3)$   
**R5.**  $(E1 \supseteq E2) \cap E3 \implies (E1 \cap E3) \supseteq E2$   
**R6.**  $E \cap E \implies E$   
**R7.**  $\sigma(E1 \supseteq E2) \implies \sigma(E1) \supseteq E2$   
**R8.**  $(E1 \supseteq E2) \supseteq E2 \implies E1 \supseteq E2$   
**R9.**  $-\sigma(E) \implies \phi(E)$   
**R10.**  $-(E1 \supseteq E2) \implies E1 \not\supseteq E2$

A successful application of **R5** makes it possible for a subsequent application of **R6** to eliminate an occurrence of the  $\cap$  operator. The binary ' $-$ ' operator is dealt with in a similar way (the rules are omitted). The unary ' $-$ ' operation (that negates a selection or a containment operation) is eliminated in a different way: **R9** and **R10** absorb the negation into the subsequent selection or containment operation (for brevity and convenience, here we use  $\not\supseteq$  as a short hand notation denoting both the negated containment operations, i.e.,  $\not\supseteq$  and  $\not\supset$ , and  $\phi$  denoting a negated selection operation,  $\sigma$ , which may be  $\sigma_r$  or

$\sigma_{a,r}$ ;  $\phi$  shall be interpreted as negating the filtering predicate of the selection operation). The negated containments,  $\not\supseteq$ , are eventually implemented by a specialized structural join algorithm, which turns to be highly efficient. We will address the specialized structural join algorithms with much detail in Section 4. The negated selection operation,  $\phi$ , is simply handled by pushing the negation down to the filtering predicate of the selection operation. It is important to point out that direct support for negated containments and the elimination of both  $\cap$  and ' $-$ ' (unary and binary) from an optimized XML query plan are to the greatest advantage of the evaluation efficiency of negated containment queries.

##### Straightforward simplification

We observe that most simplifications carried out at an early stage are beneficial, e.g.,  $E \subset E \implies E$ , while some less straightforward simplifications if hastily done at a too early stage may cause the whole optimization process being quickly captured at a local optimal result. For example, while a leftward application of **E7** eliminates **E3**, it takes away the opportunity of using 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 on for restoring this lost opportunity. Therefore, at the normalization phase we only pursue straightforward simplification, and the performed simplification rules mostly are only related to the set operations. We furnish a couple of such simplification rules below.

- R11.**  $\sigma(E) \cup E \implies E$   
**R12.**  $(E1 \supseteq E2 \cup E1) \implies E1$

#### 3.2 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 for applying structure indices to a query. Three situations have been identified for this purpose:

- A structure index is available for a  $\supseteq$  operation in an input expression and the index is 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 input (sub-)expression is in. However, after some desired transformations, the index becomes applicable and used.
- No relevant structure index is available, but after introducing a new element type (as an entrance location), a previously irrelevant index becomes applicable and 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 applies it. In the following, we present sample rules that exploit the usage of structure indexes corresponding to each of the above situations:

**Applying a readily applicable structure index.** For an input (sub)query of form  $E1 \supseteq E2$ , if a structure index between the two involved element types,  $\tau(E1)$  and  $\tau(E2)$ , happens to be available, we grasp this opportunity and apply the index by invoking the following transformation rule (In the rule, the parameter “ $\tau(E1)$ ” as a subscript of the index operator  $I$  indicates the resultant type of  $I$ ).

**R13.**  $(E1 \supseteq E2) \implies (I_{\tau(E1)}(E2) \cap E1)$  **if** a structure index is available between  $\tau(E1)$  and  $\tau(E2)$

### 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 be enabled by the transformations rendered by the  $\supseteq$  commutativity and/or associativity laws of the ePAT algebra. Specifically, the following transformation rules can be used:

**R14.**  $(E1 \supseteq (E2 \supseteq E3)) \implies ((E1 \supseteq E2) \supseteq E3)$  **if** there exists a structure index between  $\tau(E1)$  and  $\tau(E2)$ , and  $\tau(E2)$  is entrance location for  $\tau(E1)$  and  $\tau(E3)$ .

**R15.**  $(E1 \supseteq (E2 \supseteq E3)) \implies ((E1 \supseteq E3) \supseteq E2)$  **if** there exists a structure index between  $\tau(E1)$  and  $\tau(E3)$ , and  $\tau(E2)$  is entrance location for  $\tau(E1)$  and  $\tau(E3)$ .

**R16.**  $(E1 \subset (E2 \supset E3)) \implies ((E1 \subset E2) \supset E3)$  **if** there exists a structure index between  $\tau(E1)$  and  $\tau(E2)$ , and  $\tau(E1)$  is entrance location for  $\tau(E2)$  and  $\tau(E3)$ .

**R17.**  $(E1 \supset (E2 \subset E3)) \implies ((E1 \supset E2) \subset E3)$  **if** there exists a structure index between  $\tau(E1)$  and  $\tau(E2)$ , and  $\tau(E1)$  is entrance location for  $\tau(E2)$  and  $\tau(E3)$ .

These rules are a result of a guided application of the equivalences that relate to the *entrance location* notion. We choose R15 as an example to illustrate how the rules are derived.

### Proof.

$$\begin{aligned} & (E1 \supseteq (E2 \supseteq E3)) \\ \implies & ((E1 \supseteq E2) \supseteq E3) \quad (\supseteq \text{ associativity, i.e., } \mathcal{E5}) \quad (1) \\ \implies & ((E1 \supseteq E3) \supseteq E2) \quad (\supseteq \text{ commutativity, i.e., } \mathcal{E2}) \quad (2) \quad \blacksquare \end{aligned}$$

### Making “irrelevant” structure indexes relevant and applicable.

For an input query, if there is no relevant structure index found, we check whether it is possible to make a related index relevant by introducing a new element type (as entrance location) into the query (a related index is one that relates to just one of the two element types involved in an containment operation, and thus is not fully relevant but related). To explore this kind of opportunities, we need specialized rules that check specific structural properties, e.g., those connoted by the exclusivity and/or obligation notions (see Section 2.1). In order to achieve the above goal, our approach examines numerous cases. Due to space consideration, herein we show just a couple of rules of this type (a full presentation can be found in [11]) while our emphasis herein is put on the support for negated containments.

**R18.**  $(E1 \subset E2) \implies (E1 \subset E3)$  **if**  $E3$  is an entrance location for  $\tau(E1)$  and  $\tau(E2)$  and is exclusively contained in  $\tau(E2)$ ,  $\text{free}(E2)$ , and a structure index between  $\tau(E3)$  and  $\tau(E1)$  is available.

**R19.**  $(E1 \subset E2) \implies (E1 \subset E3)$  **if**  $\tau(E2)$  is an entrance location for  $\tau(E1)$  and  $\tau(E3)$  and is exclusively contained in  $\tau(E3)$ ,  $\text{free}(E2)$ , and a structure index between  $\tau(E3)$  and  $\tau(E1)$  is available.

The above rules refer to the  $\text{free}(E)$  condition, which states that the evaluation of the subexpression  $E$  returns the *full extent* of type  $\tau(E)$ , i.e., free of restriction. This condition trivially holds for every *etn*. For nontrivial  $E$  (i.e., non *etn*), database statistics can be used to quickly determine whether the predicate holds or not.

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., [12] and [29]) are furnished. We skip over this kind of rules.

It is worthy to mention that there may exist alternatives to the new element type  $E3$  interpolated by the above rules into a query (as entrance location). In order to retain our approach deterministic under all circumstances, we developed dedicated algorithms [9, 10] that can search for the optimal entrance location to serve the above purpose.

### 3.3 Simplification

The third phase, simplification, re-invokes most of the simplification rules covered by Phase 1. This is necessary as, after major transformations are performed by semantic optimization, (new) redundancies may be reintroduced, 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 on again as semantic optimization does not introduce new inconsistencies with the DTD-constraints nor causes major structural changes in the queries. Besides, as at this point we expect a thorough clean-up on the optimized queries, we add two new types of rules to the simplification phase.

The following rules are added for removing the  $\cap$  operations that may be reintroduced with an index operation:

**R20.**  $(I_{\tau(E1)}(E2) \cap E1) \implies (I_{\tau(E1)}(E2))$  **if**  $\text{free}(E1)$

**R21.**  $(I_{\tau(E1)}(E2) \cap \sigma_R(E1)) \implies \sigma_R(I_{\tau(E1)}(E2))$  **if**  $\text{free}(E1)$

We also add the following rules that carry out deep clean-up by exploiting relevant structure knowledge. A few such rules are given below as examples:

**R22.**  $E1 \subset E2 \implies E1$  **if**  $\tau(E1)$  is exclusively contained in  $\tau(E2)$

**R23.**  $(E1 \subset (E3 \subset E2)) \implies (E1 \subset E2)$  **if**  $\tau(E3)$  is entrance location for  $\tau(E1)$  and  $\tau(E2)$ , and  $\text{free}(E3)$

**R24.**  $(E1 \subset (E3 \supset E2)) \implies (E1 \supset E2)$  **if**  $\tau(E1)$  is an entrance location for  $\tau(E3)$  and  $\tau(E2)$  and obligatorily contained in  $\tau(E3)$ , and  $\text{free}(E3)$ .

### 3.4 An Example

In the following, we show the transformations that were applied to the example query introduced in Section 2 by our system. These transformations eventually made the structure index  $I_{keyword}(open\_auction)$  applied, and have the element type description completely removed from the query because of the identified exclusive containment property between keyword and description. As such, the query expression (as a logical plan) is considerably improved (the original form involves three containment operations, and the optimized form has only one containment operation plus a structure index). Without first performing the desired transformations, the structure index between `open_auction` and `keyword`, although available and relevant, would not be applied to this query.

$$\begin{aligned}
 & (keyword \subset (description \subset (open\_auction \supset \\
 & \sigma_{a='personref',r='123'}(bidder)))) \\
 \Rightarrow & \text{(by } \mathcal{R}14, \text{ i.e., } \subset \text{ associativity)} \\
 & ((keyword \subset description) \subset (open\_auction \supset \\
 & \sigma_{a='personref',r='123'}(bidder))) \\
 \Rightarrow & \text{(by } \mathcal{R}15, \text{ i.e., } \subset \text{ commutativity)} \\
 & ((keyword \subset (open\_auction \supset \\
 & \sigma_{a='personref',r='123'}(bidder))) \subset description) \\
 \Rightarrow & \text{(by } \mathcal{R}13, \text{ i.e., index introduction)} \\
 & (I_{keyword}(open\_auction \supset \\
 & \sigma_{a='personref',r='123'}(bidder)) \cap keyword) \subset \\
 & description) \\
 \Rightarrow & \text{(by } \mathcal{R}20: \cap \text{ deletion)} \\
 & (I_{keyword}(open\_auction \supset \\
 & \sigma_{a='personref',r='123'}(bidder))) \subset description) \\
 \Rightarrow & \text{(by } \mathcal{R}22: \text{ simplification based on exclusivity)} \\
 & (I_{keyword}(open\_auction \supset \\
 & \sigma_{a='personref',r='123'}(bidder)))
 \end{aligned}$$

## 4. Structural Join Algorithms for Negated Containments

It has been shown [31, 29] that effective structural join algorithms can outperform navigation and standard RDBMS join algorithms by several orders of magnitude for the tree pattern matching of XML queries. As negated containments are almost as common as regular containments used in XML queries, generalizing the family of structural join algorithms to provide direct support for negated containments can further advance the performance of XML queries involving negated containments. In this section, as examples, we propose two specialized join algorithms providing direct support for negated containments. The first one is based on the MPMGIN structural join algorithm proposed by Zhang *et al* in [31], and thus is called NMPMGIN (Negated counterpart of MPMGIN). The second one is based on the Stack-Tree-Desc algorithm proposed by Srivastava *et al* in [29], and thus is called NSTD (Negated counterpart of Stack-Tree-Desc).

In our algorithms, we adopt the same scheme as in [29]: (DocId, StartPos : EndPos, LevelNum) for encoding the positions of XML elements and string values into the IDs of the data items.

### Algorithm NMPMGIN (AList, DList)

/\* the algorithm computes the result of  $A \not\supset B$  \*/

**input:** AList of type A, and DList of type B; both sorted on DocId and position

**output:** OList holding a sub list of AList that satisfy  $A \not\supset B$

```

1. begin
2.   copy AList to OList;
3.   set a_cursor at beginning of AList;
4.   set d_cursor at beginning of DList;
5.   set o_cursor at beginning of OList;
6.   while (a_cursor != end of AList and
7.         d_cursor != end of DList) do {
8.     if (a_cursor.DocId < d_cursor.DocId)
9.       then { a_cursor++; o_cursor++; }
10.    else if (d_cursor.DocId <
11.            a_cursor.DocId)
12.      then { d_cursor++; }
13.    else {
14.      mark = d_cursor;
15.      while (d_cursor.StartPos
16.            < a_cursor.StartPos and
17.            d_cursor != end of DList) do {
18.        d_cursor++;
19.        if (d_cursor == end of DList)
20.          then { a_cursor++; o_cursor++;
21.                d_cursor = mark; }
22.        else if (a_cursor.EndPos >
23.                d_cursor.EndPos) then {
24.          remove o_cursor.val from OList;
25.          a_cursor++; /* no need to advance
26.                    o_cursor as ``remove`` has done*/
27.        }
28.      }
29.    }
30.  }
31.  output OList;
32. end

```

**Figure 2. The NMPMGIN algorithm for negated containments based on MPMGIN**

Our first algorithm for negated containments, NMPMGIN, is shown in Figure 2. The basic idea of the algorithm is similar to that of the original MPMGIN and the standard DBMS merge join algorithm. Like MPMGIN, it accepts two sorted list of items<sup>3</sup> 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. NMPMGIN initializes the output list, OList, using the first input list, AList. The original MPMGIN outputs a list of items pairs that have a specified containment relationship with items from the second input list, while our NMPMGIN returns a sub list of items extracted from list 1. This is in accordance to the semantics of the operation defined in our ePAT algebra. However, NMPMGIN can be easily adapted to produce a list of pairs of items in the same style

<sup>3</sup>Here, we use *items* instead of *elements* to extend the connotation of the concept of *containment* to cover also the containment relationship between elements and their textual values.

as the original MPMGIN. Yet, another noticeable difference between NMPMGIN and MPMGIN is that the third level loop in MPMGIN no longer exists in NMPMGIN because the new algorithm does not need to step through the matching partition in the second input list for adding the matching pairs to the output list. As a result, the time taken by NMPMGIN shall be less than the original MPMGIN which is typically bound by  $O(n^2)$ .

The advantage of providing direct support for negated containments is obvious. Let us look at the following example: “find `open_auctions` that have not received any bid”. This query can be represented as “`open_auction`—(`open_auction`  $\supset$  `bidder`)” using our ePAT algebra. A reasonable logical plan for this query is “`open_auction`— $\pi_{open\_auction}(open\_auction \bowtie_{\supset} bidder)$ ”, and the MPMGIN algorithm can be called to implement the structural join operation  $\bowtie_{\supset}$ . To execute this query, a query engine first retrieves all `open_auction` elements and all `bidder` elements, then performs a structural join using MPMGIN (which alone takes more time than our NMPMGIN), projects the join results to `open_auction`, and calculates the difference between all `open_auction` elements and the result of last step. An alternative plan is “`open_auction`  $\not\supset$  `bidder`”, if direct support for negated containments is provided. Instead of three operations, it uses only a single operation,  $\not\supset$ . The superiority of the alternative plan in terms performance is obvious, which is facilitated in our approach by the direct support from the NMPMGIN algorithm.

Our second algorithm, NSTD, is provided in Figure 3. It is different from the original Stack-Tree-Desc algorithm [29] at two places: first, at the place where the original algorithm pop the top element in the stack, we add an output statement (see line 12 in Figure 3) producing output for NSTD; second, the inner loop used by Stack-Tree-Desc for producing output is no longer needed. Analogous to NMPMGIN, the NSTD algorithm also has improved performance due to the removed inner loop.

## 5. Experiments

Previously, we proposed a novel approach [11] for XML query optimization based on deterministic transformations and various optimization heuristics, and conducted extensive experimental evaluation. The result shows that the approach is highly efficient and effective (query optimization usually takes less than a quarter second, and the performance improvement factor is usually 3 or more).

In this paper, we mainly addressed the issue of efficient support for negated containment queries. We generalized the family of structural join algorithms to provide direct, algorithmic support for the evaluation of negated structural relationships. In order to effectively apply these specialized structural join algorithms to query evaluation, our approach is revised for additionally identifying the opportunity of applying a dedicated structural join algorithm to negated containment queries.

For the purpose of experimental evaluation, we conducted three tests. Test 1 is designed to study the performance of our new approach, which is compared with plain query evaluation (no optimization) and with our previous approach (having no special support for negated containments). Test 2 is for the scaling effect, and Test 3 is for the effect of selectivity of queries on

### Algorithm NSTD (AList, DList)

/\* The algorithm computes the result of  $A \not\supset B$  \*/

/\* Assume that all nodes in AList and DList have the same DocId \*/

**input:** AList of type A, and DList of type B; both sorted on StartPos

**output:** Items from AList that do not contain any items from DList as descendents

```

1. begin
2.   a = AList->firstNode;
3.   d = DList->firstNode;
4.   OutputList = NULL;
5.   while (the input lists are not empty or
6.         the stack is not empty) do {
7.     if ((a.StartPos > stack->top.EndPos) and
8.         (d.StartPos > stack->top.EndPos)) then {
9.       /* time to pop the top element in the
10.        stack and produce output*/
11.       tuple = stack->pop();
12.       output tuple; }
13.     else if (a.StartPos < d.StartPos) then {
14.       stack->push(a)
15.       a = a->nextNode }
16.     else { d = d->nextNode }
17.   }
18. end

```

**Figure 3. The NSTD algorithm for negated containments based on Stack-Tree-Desc**

execution performance. For carrying out Test 1, we adopted the well-known XMark [27] benchmark database (at the standard scaling factor of 1.0); we refer to this database as **Database 1**. For doing Test 2 and 3, we used a synthesized database, a proceedings database (the same as we used before [11]), referred to as **Database 2**; using a synthesized database gives us much convenience for populating the database at various scales and adjusting the selectivity of the tested queries.

Our prototype system is implemented in Java. The Oracle RDBMS is used as a fast platform for the 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, SQL query execution by Oracle query engine, and SQL result synthesizing. The last step invokes the structural join algorithms which in system are implemented outside Oracles query engine due to its closed-ness. We implemented a variant of MPMGIN [31] and our NMPMGIN (see Section 4) in the test-bed.

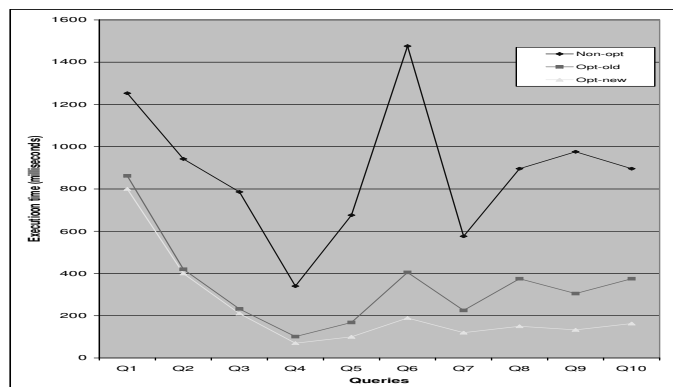
Our test-bed is set up in a typical client/server database environment. The client side runs Oracle SQL\*Plus (Release 8.1.6.0.0) on Window XP Pro., and the CPU is a Celeron processor of 500MHZ with 192MB RAM. The server side runs an Oracle8i Enterprise Edition (Release 8.1.6.0.0) database server, installed on Sun Ultra 5 with an UltraSparc-II CPU of 333MHZ and a RAM of 512MB.

**Test 1: Effectiveness of proposed approach**

The XMark benchmark database (scaling factor 1.0) is used for this test. XMark comes with 20 predefined benchmark queries, of which only one (Query 17) involves negated containment operations. As our objective with this test is to study the performance potential of our new approach to negated containment queries, we adopted just four (including Query 17) of the 20 XMark benchmark queries, and devised 6 new (negated containment) queries over the same database. The 10 queries are listed below (of which, Q1, Q2, Q3, and Q4 are selected/adapted from the original XMark queries, Query 2, 6, 15, and 17, respectively):

- Q1. Return the increases of all open auctions.
- Q2. Return names of items in any region.
- Q3. Return keywords in emphasis in annotations of closed auctions.
- Q4. Which persons do not have a homepage?
- Q5. Find names of persons who do not have a homepage.
- Q6. Find names of persons who do not have a homepage and do not have an income.
- Q7. Find name of items that do not have a reserve.
- Q8. Find all names of item that have "gold" in their description, but no name.
- Q9. Find all bidders of open-auctions that do not have an increase of "\$5".
- Q10. find all bidders of open-auctions that have increase but no increase of "\$5".

The selectivity and optimization time of the queries are provided in Table 1; the performance result is shown in Figure 4



**Figure 4. Performance results of Database 1 benchmark queries**

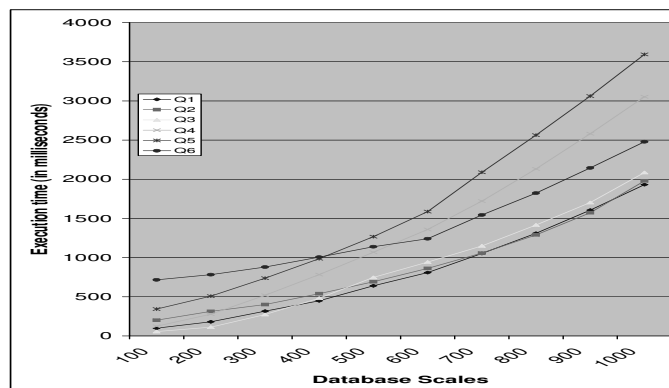
Figure 4 compares the performance of the queries without optimization, queries optimized by our old approach (but without special support for negated containments), and queries optimized by our new approach (proposed in this paper, with special

support for negated containments). For queries without negated containments, our new approach does not improve the execution performance any further (comparing with our old optimization approach [11]). For negated containment queries, our new approach outperforms the old approach almost as much as our old approach surpasses non-optimized queries; the improvement is roughly of a factor of 3 or 4. We expect a much bigger improvement factor if we can switch to a rather open host system that allows deployment of built-in structural join algorithms (for both containments and negated containments).

**Test 2: Scaling effect**

In order to study the scaling effect of our new approach, we repopulated the proceedings database at 10 different scales, corresponding to 100, 200, . . . , 1000 root documents, respectively. The characteristics of the database are shown in Table 2 (We show only the characteristics at the two end scales, i.e., scale 1 and 10, for example). The benchmark queries and their selectivity and optimization time are shown in Table 3.

Both optimized and non-optimized queries show similar — weak linear — scalability as shown in Figure 5 (in the figure, we only show the scalability of optimized queries).



**Figure 5. Scaling effect of optimized benchmark queries with Database 2**

**Test 3: Effect of selectivity on query performance**

Our test for studying the effect of selectivity on query performance is based on the same proceedings database (at scale 10, i.e., with 1000 root documents) and the same set of (six) test queries. We observe consistent sub-linear effect of query selectivity on execution performance for optimized queries (similar for non-optimized queries), as shown in Figure 6.

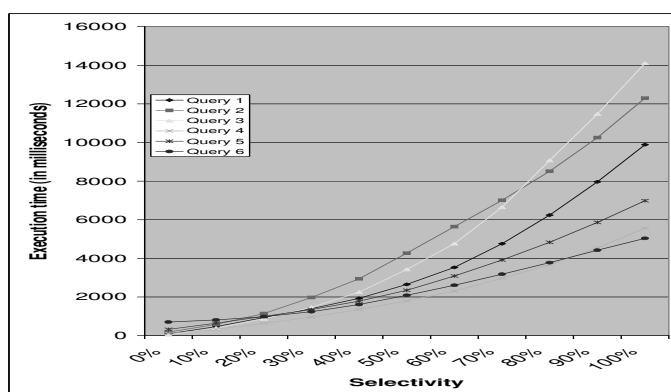
Our new experiment results regarding the scaling effect and effect of query selectivity on execution performance are very similar to what we obtained from our previous study with our old approach reported in [11], except for the noticeable improvement on the total performance of all optimized queries. This is understood because the six tested queries on the proceedings database all do not involve negated containments, and thus the new mechanisms introduced by our new approach for dealing with negated containments do not have direct impact on these queries. Yet, the performance of these queries gets gener-

Description	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Selectivity(%)	60	30	12	4	4	3	6	7	12	14
Optimization(sec)	.333	.254	.410	.096	.041	.202	.254	.172	.110	.124

**Table 1. Properties of Database 1 benchmark queries**

Scale	Size(MB)	#Docs	#Elements	Max degree	Min degree	Depth	Max cardinality	Min cardinality
1	28.95	100	1,206,424	20	1	6	421,690	100
10	252.68	1000	11,546,896	20	1	6	3,695,237	1000

**Table 2. Characteristics of Database 2 (at scale 1 and 10)**



**Figure 6. Effect of selectivity on the optimized benchmark queries with Database 2**

ally improved because our new implementation tuned the entire system' performance.

**6. Related Work**

A direct experimental comparison with other related systems (like Timber [32]) is appealing, but is very hard because it is almost impossible for us to implement a related (huge) system on our experimental platform. Instead, in this section, we briefly review some related work and analytically compare with our approach.

Lore [23, 24] 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 logical level optimization. The work in [14] 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 [15], a comparable strategy for exploiting a grammar specification for optimizing queries on semi-structured data is studied. In that study, efforts were made on how to make complete use of the available grammar to *expand* a given query. In [6], 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 node. Query processing is then guided to a more specific class of the elements of this type for pruning 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 [30], based on the notion of rooted paths, two types of optimization were proposed: path complementing (i.e., if the path in a 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 [10, 11]. However, our algorithm is more general and may identify more opportunities for reducing a path (not necessarily a rooted one). In [5], 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). The path constraints investigated in [5] is restricted to only data semantics. Our approach explores structural semantics. Path and tree pattern matching build the core of XML query processing. A bundle of approaches [4, 29, 31, 22, 19] have been proposed for supporting path and tree pattern matching, commonly known as *structural or containment 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 within the lists of 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. More importantly, we introduced the idea of providing direct support for negated containments in a query. Finally, with regard to algebras for XML, TAX [21] and XAL [16] are worth of mentioning. TAX is a tree-based algebra, and is proper as a "representation algebra" for XML queries. In contrast, XAL is a node-based algebra, and is more proper for query optimization. In our opinion, there are two kinds of trees that are equally important for XML queries: pattern trees (as advocated by TAX) and operation trees (which are naturally and well supported by

	Queries (formatted in XPath)	Selectivity	Optimization(sec)
Q1	//Article[./Title ftcontains "Data Warehousing" or ./Keywords ftcontains "Data Warehousing"]	2.0%	0.192
Q2	//Article[./Title ftcontains "Data Warehouse"]/Abstract	5.1%	0.143
Q3	//Articles[./Title ftcontains "Data Warehousing"]/Sections/Section[@title="Introduction"]/Paragraph	6.4%	0.666
Q4	//Article[./Surname ftcontains "Aberer"]	2.3%	0.537
Q5	//Section/Paragraph [@title = "Summary"]	4.8%	0.741
Q6	//(article   ShortPaper) // Paragraph[.ftcontains "Multidimension" or .ftcontain "OLAP"]	3.1%	1.921

**Table 3. Database 2 benchmark queries and properties**

node-based algebras). Pattern trees vest great expressive power to an algebra for representing XML queries, while operation trees form the basis of manipulation for query optimization. We are now in a process of extending our ePAT to dual modes: *deep mode* works with pattern trees and *shallow mode* works with individual nodes.

To the best of our knowledge, our work as reported in this paper is the only one that combines deterministic algebraic transformation with direct algorithmic support for negated containments. Our work is the only one that addresses the issue of efficient support for negated containments.

## 7. Summary

In this paper, we presented an innovative, logical-level optimization approach for XML queries. Our approach is accomplished by a rule system that carries out deterministic transformations on XML queries (represented internally as ePAT algebraic expressions) to achieve optimization. Logical-level optimization is achieved through three consecutive transformations phases: normalization, semantic optimization, and simplification. Experimental study confirms the validity of the proposed new approach: it is efficient, effective, and scalable.

An important point we have made is that direct support for negated containments has great potential for improved execution efficiency of XML queries. This issue is very important but overlooked by the research community.

Our approach is generally a logical-level one and is independent of the specifics of any particular host system. Our implemented optimizer is good enough as a "performance enhancer" for any XML data management system if it does not yet have such a functional module. In case the targeted application domain has a high demand for the quality of optimized queries (for example, there are many relatively complex and repeatedly executed queries), our logical-level optimization approach can be gracefully complemented by additionally using a subsequent physical-level optimizer, which typically performs cost-based optimization and should be aware of the logical optimization already done at the logical level. Effective collaboration between such a logical optimizer and a physical optimizer expects further investigation.

At present, we are extending the ePAT algebra to make it more expressive and proper for XML query representation and

optimization. In the meanwhile, we are developing more variants of specialized structural join algorithms for supporting negated containments. Experimental study of these algorithms is the authors near future work.

## References

- [1] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon, "XQuery 1.0: An XML Query Language", 2003 (<http://www.w3.org/TR/xquery/>).
- [2] K. Böhm, K. Aberer, E. J. Neuhold, and X. Yang, "Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM", *The VLDB Journal*, vol. 6, no. 4, pp. 296-311, Nov. 1997.
- [3] K. Böhm, K. Aberer, M. T. Özsu, and K. Gayer, "Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition", in *Proc. of IEEE Intl Forum on Research and Technology Advances in Digital Libraries*, Santa Barbara, CA, Apr. 1998, pp. 196-205.
- [4] J.-M. Bremer, and M. Gertz, "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](http://www.db.cs.ucdavis.edu/papers/TR_CSE-2003-04_BremerGertz.pdf)).
- [5] P. Buneman, W. Fan, S. Weinstein, "Query Optimization for Semistructured Data Using Path Constraints in a Deterministic Data Model", in *Proc. of DBPL Conf.*, Edinburgh, Scotland, UK, Sep. 1999, pp. 208-223.
- [6] T.-S. Chung, H.-J. Kim, "XML query processing using document type definitions", *Journal of Systems and Software*, vol. 64, no. 3, pp. 195-205, 2002
- [7] C.Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions", in *Proc. of ICDE*, San Jose, CA, Feb. 2002, pp. 235-244.
- [8] C. Y. Chan, M. N. Garofalakis, and R. Rastogi, "RE-Tree: An Efficient Index Structure for Regular Expressions", in *Proc. of VLDB Conf.*, Hong Kong, China, Aug. 2002, pp. 263-274.

- [9] D. Che, "Implementation Issues of a Deterministic Transformation System for Structured Document Query Optimization", in *Proc. of IDEAS*, Hong Kong, China, Jul. 2003, pp. 268-277.
- [10] D. Che, "Accomplishing Deterministic XML Query Optimization", *J. of Computer Science and Technology*, vol. 20, no. 3, pp.357-366, May 2005.
- [11] D. Che, K. Aberer, and M. T. Özsu, "Query Optimization in XML Structured-Document Databases", *The VLDB Journal* (in press).
- [12] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents", in *Proc. of VLDB Conf.*, Hong Kong, China, Aug. 2002, pp. 263-274.
- [13] J. Clark, and S. DeRose, "XML Path Language (XPath) Version 1.0", 1999 (<http://www.w3.org/TR/1999/REC-xpath-19991116>).
- [14] M. Consens, and T. Milo, "Optimizing Queries on Files", in *Proc. of ACM SIGMOD Conf.*, Minneapolis, Minnesota, May 1994, pp. 301-312.
- [15] M. F. Fernandez, and D. Suci, "Optimizing Regular Path Expressions Using Graph Schemas", in *Proc. of ICDE*, Orlando, Florida, Feb. 1998, pp. 14-23.
- [16] F. Frasincar, G.-J. Houben, *et al*, "XAL: an Algebra for XML Query Optimization", in *Proc. of 13th Australasian DB Conf.*, Melbourne, Victoria, Jan. 2002.
- [17] G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries", in *Proc. of VLDB Conf.*, Hong Kong, China, Aug. 2002, pp. 95-106.
- [18] G. Graefe, and D. DeWitt, "The EXODUS optimizer generator", in *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, pp. 160-172.
- [19] T. Grust, "Accelerating XPath location steps", in *Proc. of ACM SIGMOD Conf.*, Madison, WI, June 2002, pp. 109-120.
- [20] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu, "Approximate XML joins", in *Proc. of ACM SIGMOD Conf.*, Madison, WI, Jun. 2002, pp. 287-298.
- [21] H.V. Jagadish, L.V.S. Lakshmanan, D. Srivastava, and K. Thompson, "TAX: A Tree Algebra for XML", in *Proc. of DBPL Conf.*, Rome, Italy, Sep. 2001, pp. 149-164.
- [22] Q. Li, and B. Moon. "Indexing and Querying XML Data for Regular Path Expressions", in *Proc. of VLDB Conf.*, Rome, Italy, Sep. 2001, pp. 361-370.
- [23] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: a Database Management System for Semistructured Data", *SIGMOD Record*, vol. 26, no. 3, pp. 54-66, Sep. 1997.
- [24] J. McHugh, and J. Widom. "Query Optimization for XML", in *Proc. of VLDB Conf.*, Edinburgh, Scotland, Sep. 1999, pp. 315-326.
- [25] T. Milo, and Dan Suci, "Index Structures for Path Expressions", in *Proc. of ICDT*, Jerusalem, Israel, Jan. 1999, pp. 277-295.
- [26] A. Salminen, and F. W. Tompa, "PAT Expressions: an Algebra for Text Search", *Acta Linguistica Hungarica*, vol. 41, no.1, pp. 277-306, 1994.
- [27] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management", in *Proc. of VLDB Conf.*, Hong Kong, China, Aug. 2002, pp. 974-985.
- [28] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price, "Access Path Selection in a Relational Database Management System", in *Proc. of ACM SIGMOD Conf.*, Boston, Massachusetts, May/June. 1979, pp. 23-34.
- [29] 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", in *Proc. of ICDE Conf.*, San Jose, Feb. 2002, pp. 141-152.
- [30] G. Wang, and M. Liu, "Query Processing and Optimization for Regular Path Expressions", in *Proc. of CAISE*, Klagenfurt, Austria, Jun. 2003, pp. 30-45.
- [31] C. Zhang, J. F. Naughton, *et al*, "On Supporting Containment Queries in Relational Database Management Systems", in *Proc. of ACM SIGMOD Conf.*, Santa Barbara, CA, Jun. 2001, pp. 425 - 436
- [32] University of Michigan, *The TIMBER system*. <http://www.eecs.umich.edu/db/timber/>.



**Dunren Che** is an assistant professor in the Department of Computer Science, Southern Illinois University at Carbondale, USA. He received his PhD in Computer Science from Beijing University of Aeronautics and Astronautics, Beijing, China in 1994. Afterwards, he gained several years of postdoctoral research experience from various research institutes (including,

Tsinghua University at Beijing, China, IPSI Institute of German National Research Center for Information Technology in Germany, and Johns Hopkins University at Baltimore, USA). His research interests include database, persistent and intelligent agent, Bioinformatics, and Web development. His most recent research is focused on XML database management systems, especially XML query processing and optimization.