

Query Optimization in XML Structured-Document Databases

Dunren Che¹, Karl Aberer², M. Tamer Özsu³

¹ Department of Computer Science, Southern Illinois University, Carbondale, IL 62901, USA
e-mail: dche@cs.siu.edu

² School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, (EPFL) CH-1015 Lausanne, Switzerland
e-mail: karl.aberer@epfl.ch

³ School of Computer Science, University of Waterloo, Canada N2L 3G1
e-mail: tozsu@db.uwaterloo.ca

Received: November 3, 2004/Revised version: May 14, 2005

Abstract While the information published in the form of XML-compliant documents keeps fast mounting up, efficient and effective query processing and optimization for XML have now become more important than ever. This article reports our recent advances in XML structured-document query optimization. In this article, we elaborate on a novel approach and the techniques developed for XML query optimization. Our approach performs heuristic-based algebraic transformations on XPath queries, represented as PAT algebraic expressions, to achieve query optimization. This article first presents a comprehensive set of general equivalences with regard to XML documents and XML queries. Based on these equivalences, we developed a large set of deterministic algebraic transformation rules for XML query optimization. Our approach is unique, in that it performs exclusively deterministic transformations on queries for fast optimization. The deterministic nature of the proposed approach straightforwardly renders high optimization efficiency and simplicity in implementation. Our approach is a logical-level one, which is independent of any particular storage model. Therefore, the optimizers developed based on our approach can be easily adapted to a broad range of XML data/information servers to achieve fast query optimization. Experimental study confirms the validity and effectiveness of the proposed approach.

Key words XML query optimization XML query processing XML database Query transformation Deterministic query optimization

1 Introduction

XML (eXtensible Markup Language) [44] has become the *de facto* standard for information/data representation and exchange on the Internet and elsewhere. As a consequence, more and more data sources switch over to XML and express their contents using XML or an XML dialect, e.g., the familiar

HTML, NITF (in the news industry), WeatherML (for weather information), CellML (in bioinformatics), and XMLPay (for Internet-based payments). The rapidly growing XML data sources call for commensurate management solutions that are *XML-aware*. It has been a wide consensus that XML documents/data should obtain the same type of management functionalities as conventional data received from RDBMSs, and the database community is well underway towards this destination.

In recent years, many storage schemes for XML data have been proposed, e.g., mapping XML data to relational [18, 21, 5, 38] or object-relational models [28, 41], using special-purpose databases such as semistructured databases [32, 33], or developing *native* XML databases [19]. A key issue that faces every XML data management system is the optimization of XML queries.

Query optimization in the context of XML databases is extremely challenging. The main reason for this is the high complexity of the XML data model, as compared with other data models, e.g., relational models. This high complexity renders a much enlarged search space for XML query optimization. Furthermore, XML applications are typically Webhooked and have many simultaneous, interactive users. This dynamic nature requires highly efficient XML query processing and optimization. The classical cost-based and heuristic-based approaches yield unacceptably low efficiency when applied to XML data – query optimization itself becomes very time-consuming because of the huge search space for optimization caused by the high complexity of the XML data model. Lots of work related to XML query processing has been done, but the majority is focused on investigation for efficient supporting algorithms [17, 32, 40, 24, 31, 26, 13, 47] and indexing schemes [31, 9, 35, 25, 10]. Complete and systematic work on XML query optimization has hardly been reported. We will discuss related work in detail in Section 7.

1.1 Motivation

The essential difference between XML data and traditional data (e.g., relational data) is the extra structural relationships between the various elements in an XML data source. On the one hand, these structural relationships render high complexity for XML data modeling and query optimization; on the other hand, they imply invaluable opportunities — source of semantic knowledge for efficient XML query optimization — which is, however, often overlooked or inadequately exploited. For example, assume a query asks for the instances of element type $t1$ that are subelements of the instances of element type $t2$ and if we know that every $t1$ element has to be contained in a $t2$ element according to the data sources DTD or XSD (XML Schema Definition), then we can simply return all instances of type $t1$ without checking the containment relationship between the elements of the two types. This is a trivial example of using the structure knowledge of XML data to conduct a *deterministic* transformation on an input query for better evaluation efficiency. Here, by *deterministic*, we mean that the transformation once carried out will bring in definite improvement (simplification in this case) on the input query expression. More complicated cases exist that yield plenty of opportunities for efficient optimization of XML queries, e.g., exploiting an available structure index which is otherwise inapplicable to an input query (A structure index for now can be thought of as a shortcut between two *distant* elements within the structure of the XML data source). Obviously, using a structure index can significantly reduce the cost of evaluating a covered containment operation. For example, let $t1$, $t2$, and $t3$ be three element types; $t1$ is a distant descendent of $t2$, and $t2$ is a direct child of $t3$ (see Fig. 1); if the evaluation of a query needs to check the containment relationship between $t1$ and $t2$, a straightforward but costly way is to traverse all the intermediate “generations” between $t1$ and $t2$; however if a structure index between $t1$ and $t3$ is available, we can bypass the long path (from $t1$ to $t2$) by using the structure index to reach $t2$ ’s parent, $t3$, first, and then get to the target $t2$.

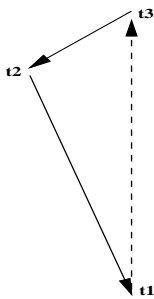


Fig. 1 Bypassing a lengthy path using a ‘shortcut’

The above examples indicate a huge realm where efficient XML query optimization can be pursued by exploiting the structure knowledge of XML data and structure-related semantics carried on by an XML query expression. This obser-

vation motivated the work reported in this writing and set the keynote of this article.

XML query optimization is one of the most challenging issues facing the database research community. It is only obtainable through a systematically and carefully worked-out approach. To this end, we first need to study the equivalence issue related to XML queries and XML data because query transformation has to be based on query equivalence. Secondly, we need to work out a good strategy to efficiently and effectively accomplish XML query optimization by using these equivalences. The strategy turns out to be a real challenge because, as mentioned before, we are now confronting a much enlarged search space, from which equivalent transformations are to be conducted in order to locate an optimal evaluation plan. A pretty straightforward guideline in our mind is to find a way to *radically* prune the search space during query optimization but still be able to obtain a sufficiently good plan though it may not be an optimal one. The solution we come up with is the so-called *deterministic optimization* approach. By this approach, every transformation applied to a query has to be deterministic, in the sense that it is bound to produce nontrivial improvement on the input query in terms of evaluation efficiency. This type of transformation, by its nature, is heuristic-based and relies on proper exploitation of the structure knowledge of XML data. Often in a scenario, a structure index is available but not applicable because of the particular, unfavorable structure pattern of a query. However, the hidden opportunity of eventually applying the index to the query may exist and can be identified by guided transformations that exploit particular structural properties of the source XML data. Deterministic transformations are achieved in our approach via invocation of heuristic transformation rules, which are based on the more general equivalences of XML queries. Thirdly, we need a group of supporting functions to facilitate the implementation of our approach. All these issues are addressed systematically in this article.

1.2 Scope of the Article

In this article we address the issues related to efficient XML query optimization at the logical level. Instead of directly targeting the full-fledged XQuery [2], which is emerging as the standard query language for XML, this article focuses on its core sublanguage, XPath [14], which is small and can make our effort more focused on solving the most challenging problems of XML query optimization.

Transformation-based query optimization is typically applied to queries represented as algebraic expressions. To this end, we adopted the PAT algebra [37] which was developed specifically for structured text access. The key constructs of XPath queries can be mapped to PAT operations. The PAT algebra will be explained later in Section 2.2. Traditionally, a database schema (at conceptual level) specifies the types of entities, the relationships among the entities, and the constraints that are to be enforced by the DBMS. Query optimization typically uses algebraic transformations carried out

on query expressions to improve the queries in terms of evaluation efficiency. Semantic optimization additionally resorts to the exploitation of data semantics (which is usually not captured in the database schema) to obtain further better optimization result. XML provides yet a third dimension for query optimization — the semantic knowledge regarding the structure of XML data — that may be used as an additional source for query optimization. It is this new dimension that this article is intended to explore to achieve XML query optimization.

In this article, we attack the optimization problem of XML queries from a transformation-based perspective — conducting algebraic transformations on query expressions and eventually producing an “optimal” evaluation plan (which is a logical query expression) from the heuristic-point of view. Our goal is achieved primarily through the exploitation of the DTD-knowledge, general and (XML-)specific optimization heuristics, particular structural properties of XML data, and potential structure indices.

The XML query optimizer thus developed based on the presented approach is independent of the specifics of any particular storage model, and can be hosted by a wide range of XML data repositories, e.g., a file system, an RDBMS, an OODBMS, or a native XML database system. For the experimental study of our approach, we adopted the Oracle RDBMS as the host of our test-bed, and conducted a series of experiments.

The major contributions of our work that is to be expounded later on are summarized below:

- Based on the PAT algebra, we introduce a class of equivalences with regard to the nature of XML queries. These equivalences serve not only as the basis of our work, but as a general framework for transformation-based XML query optimization in a broad sense.
- We bring into light a large set of heuristic transformation rules for accomplishing deterministic optimization on XML queries.
- A variety of optimization heuristics are presented together with the various deterministic transformation rules. A highlighted point of our approach is the application of the structure knowledge of XML data for obtaining significant simplifications of a query, enabling new opportunities of applying a structure index, or shortening an imperative path involved in a query.
- We develop highly efficient transformation strategies and procedures for XML query optimization.
- We demonstrate the validity of our approach via analytical expounding and experimental evaluation.
- Last and most important, this article reports the first, complete, and systematic work on an algebraic approach for XML query optimization, to the best of our knowledge.

1.3 Organization of the Article

The remainder of this article is organized as follows: Section 2 sets forth the preliminaries, including an overview of

```
<!DOCTYPE Proceedings [
<!ELEMENT Proceedings (Preface, (Article |
                          ShortPaper)*, AuthorIndex)>
<!ATTLIST Proceedings Target IDREF #IMPLIED>
<!ELEMENT Article (Title, Authors, Abstract,
                   Keywords, Sections...)>
<-- Ellipsis indicates omitted parts -->
<!ATTLIST Article IDName ID #REQUIRED>
<!ELEMENT ShortPaper (Title, Authors, Sections...)>
<!ATTLIST ShortPaper IDName ID #REQUIRED>
<!ELEMENT Authors (Author*)>
<!ELEMENT Author (Name)>
<!ELEMENT Name ((Surname, GivenName?), Addition?)>
<!ELEMENT (Surname | GivenName | Addition) (#PCDATA)>
<!ELEMENT Sections (Section+)>
<!ELEMENT Section (Paragraphs)>
<!ATTLIST Section Title #PCDATA #IMPLIED>
<!ELEMENT Paragraphs (Paragraph+)>
<!ATTLIST Paragraph Title #PCDATA #IMPLIED>
...
]>
```

Fig. 2 Fragment of the Proceedings DTD

XML and related notions and a brief introduction to the PAT algebra. Section 3 discusses the general equivalences, which come from three different sources — set theory, obvious DTD constraints, and specific structural properties of XML data. Section 4 describes our methodology for XML query optimization, including the general control strategy, transformation organization, and related optimization heuristics. Section 5 addresses the implementation issues — mainly, the algorithms adopted in our approach. Section 6 presents the results of experimental evaluation of our approach. Section 7 provides an overview of related work. The article is concluded in Section 8.

2 Preliminaries

In this section, we first shed light on a few basic notions related to XML; we then introduce the PAT algebra [37, 3] that is adopted and extended in our approach. Meanwhile, a sample XML DTD and example queries are introduced.

2.1 Basics of XML and Related Notions

XML identifies data elements by means of “tags”. For a given class of documents, the legal markup tags are normally defined in a DTD/XSD. For ease of presentation, in this article we base our discussion solely on the DTD notion, but the techniques presented are independent of the notion and shall straightforwardly extend to the situation that uses an XSD instead.

Fig. 2 is a fragment of a sample DTD that will be used throughout this article. The logical structure of the documents that follow this DTD is illustrated in Fig. 3.

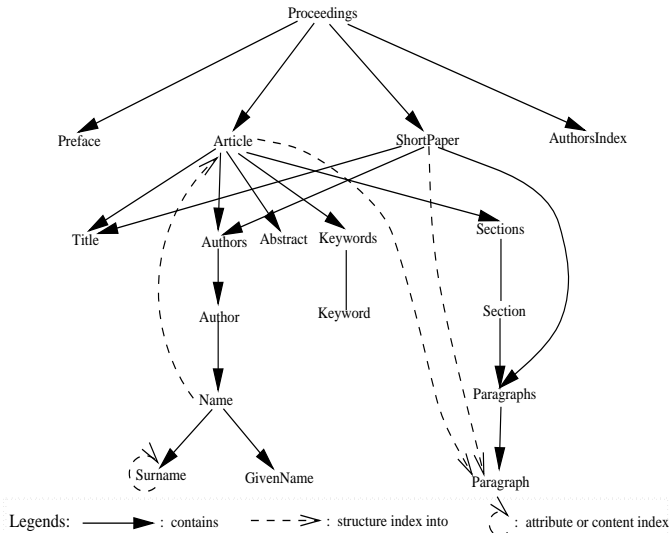


Fig. 3 Logical structure of the Proceedings DTD extended with index information

A DTD identifies three things: entities, attributes, and content model of elements. Elements (types) are the components of XML data and define the allowable tags. Attributes are associated with elements and allow the specification of meta-data for elements. Each element type is specified with a content model that describes the composition structure of the elements of the type and forms the basis of the DTD. Our algorithms (Section 5) concern the *content model* of element types. So in the following we allocate space for the discussion of this concept.

Definition 1 (Content model) Given a set of element type names ETN , the content model defines the structure of elements of each element type by a term of the following structure:

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

where $\langle etn \rangle$ indicates that the content is an element of the element type, named etn ; c_1^* stands for zero or more occurrences of c_1 ; $c_1 ?$ means an optional occurrence of c_1 ; $c_1 | c_2$ represents an occurrence of c_1 or c_2 ; (c_1, c_2) indicates an occurrence of c_1 followed by an occurrence of c_2 ; c_1^+ is short for (c_1, c_1^*) , and $c_1 \& c_2$ is short for $((c_1, c_2) | (c_2, c_1))$.

The comma in a content model is called the sequence connector (SEQ) or SEQ-node if the term is viewed as a tree; “|” is called the OR-connector (OR) or OR-node; “?” is the optional occurrence indicator; “*” is the optional-and-repeatable occurrence indicator. For ease of discussion, we will use etn as a shorter alias of the phrase “element type name”, and treat element *types* and etn ’s as synonyms in our discussion.

Definition 2 (DTD graph) The DTD graph of a DTD is a directed graph $G = (V, E)$. The vertex set V contains all the etn ’s of the DTD, and an edge (ET_i, ET_j) in the edge set E indicates that ET_j occurs in the content model of ET_i . $RT \in V$ is the root etn of the DTD.

By means of a *DTD graph*, we can visualize certain important relationships induced by the DTD, such as the *contains/contained-in* relationship among element types, which is more formally defined below.

Definition 3 (Directly-contains/directly-contained-in) Element type ET_i directly-contains element type ET_j if there exists an edge (ET_i, ET_j) in the DTD graph G . Conversely, ET_j is directly-contained-in ET_i .

We refer to ET_i as an *external type*, and ET_j an *internal type* in the context of a containment relationship.

The more general *contains/contained-in* relationship between element types is the transitive closure of the directly-contains/directly-contained-in relationship on element types.

Definition 4 (Path in DTD graph) A path in a DTD graph G , is a sequence of element types (ET_i, \dots, ET_j) s.t. ET_k directly contains ET_{k+1} , $i \leq k < j$.

The reverse of a path is also referred to as a path when the difference does not need to be stressed.

It is worth to notice that our *containment* and *path* definitions are given at the *type* level, but they exist at the *instance* level, i.e., individual XML elements.

2.2 PAT Algebra

W3C is now finalizing the XQuery proposal [2] as a standard language for querying various XML data sources (including XML document files and databases). XQuery borrowed its core notion, path expression, from XPath. Both XQuery and XPath are essentially *expression* languages — everything is an expression that evaluates to a value. Except for some obscure forms (mostly unusual “axis specifiers”), all XPath expressions are also XQuery expressions. The two languages are based on a data model that consists of a tree structure of various nodes. An XPath expression specifies a pattern that selects a set of XML nodes. Our effort for XML query optimization has been dedicated to a core subclass of XML queries, XPath expressions. We have therefore adopted and extended the PAT algebra [37, 3] for representing XPath queries as PAT expressions so that algebraic transformation/optimization can be performed. However, only a restricted version of the extended PAT is to be reviewed below due to lack of space.

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

“E” (as well “E1” and “E2”) stands for a PAT expression, etn introduces an element type name, “r” is a regular expression representing a matching condition on the textual contents of elements, and “A” designates an attribute and dictates that the matching is to be carried out on attribute values instead of the element contents.

The PAT algebra is *set* oriented, in the sense that each PAT algebra operator and each PAT expression evaluate to a *set* of XML elements. \cup , \cap and $-$ stand for the three standard

set operators, namely, union, intersection and difference. A PAT algebra expression involving a set operator is valid if the two arguments of the operator are *type-compatible*, i.e. both return elements of the same type. We do not assume that PAT algebra queries can only be posed with respect to typed documents. Nevertheless, any PAT expression is required to be type-consistent, and as a consequence, type constraints on the composition of PAT expressions are always imposed with respect to the type compatibility of set operations.

$\sigma_r(E)$ takes a set of elements and returns those whose contents match the regular expression r , while $\sigma_{A,r}(E)$ takes a set of elements and returns those whose value of attribute A matches the regular expression r . Operator \subset returns all elements of the first argument that are contained in an element of the second argument, while \supset returns all elements of the first argument that contain an element of the second argument. The last PAT operator listed, I , signifies the application of an available index (of various types, see Section 4.1).

Among those omitted from the above list are the two other containment operations, \supseteq and \subseteq , representing “directly-contains” and “directly-contained-in”. They stand for special cases of the general containment operations. Due to space consideration, we do not discuss their individual roles in the framework of our optimization approach presented in this article.

More precisely, the semantics of the PAT algebra can be given by using two (partial) functions, $\tau : \mathcal{P} \rightarrow ETN$ and $ext : \mathcal{P} \rightarrow \mathcal{E}$, where \mathcal{P} is the set of PAT expressions, ETN is the set of *etn*'s in an XML database, and \mathcal{E} is the set of all elements in the database.

Definition 5 (Type of PAT expressions) *The type of a PAT expression E is recursively defined as follows:*

$\tau(etn) = etn$.
 $\tau(E1 \cup E2) = \tau(E1)$, if $\tau(E1) = \tau(E2)$ and $\tau(E1)$ defined, otherwise undefined; analogously for \cap and $-$.
 $\tau(\sigma_r(E)) = \tau(E)$, if $\tau(E)$ defined, otherwise undefined; analogously for $\sigma_{A,r}(E)$.
 $\tau(E1 \subset E2) = \tau(E1 \supset E2) = \tau(E1)$, if $\tau(E1)$ and $\tau(E2)$ defined, otherwise undefined.

The semantics of PAT query expression is given by the following definition.

Definition 6 (Extent of PAT expressions) *For a given database, the function ext is recursively defined below:*

$ext(etn) = \{e \in \mathcal{E} \mid \text{the element type of } e \text{ is } etn\}$.
 $ext(E1 \cup E2) = ext(E1) \cup ext(E2)$.
Analogously for \cap and $-$.
 $ext(\sigma_r(E)) = \{e \in ext(E) \mid r \text{ matches a substring in the content of } e\}$.
 $ext(\sigma_{A,r}(E)) = \{e \in ext(E) \mid e \text{ has attribute } A \text{ and } r \text{ matches a substring in the content of the attribute}\}$.
 $ext(E1 \subset E2) = \{e1 \in ext(E1) \mid \exists e2 \in ext(E2) \text{ s.t. } e1 \text{ contained in } e2\}$.
 $ext(E1 \supset E2) = \{e1 \in ext(E1) \mid \exists e2 \in ext(E2) \text{ s.t. } e1 \text{ contains } e2\}$.

The following corollary is straightforward:

Corollary 21 *Each expression with a defined type, say E , evaluates to a set of elements of a single type, namely $\tau(E)$.*

The \supset and \subset operators allow references within XML queries to the structure of XML data and will be in the spotlight in our work for XML query optimization.

We now introduce five queries as running examples in this article. These queries will be represented first as XPath expressions and then as PAT expressions.

Query 1. Find all articles about “Data Warehousing”.

XPath: `//Article[./Title ftcontains “Data Warehousing” or ./Keywords ftcontains “Data Warehousing”]`

PAT: $((Article \supset (\sigma_{Data\ Warehousing'}(Keywords))) \cup (Article \supset (\sigma_{Data\ Warehousing'}(Title))))$

Query 2. Find the abstracts of all articles which have the words “Data Warehouse” in their title.

XPath: `//Article[./Title ftcontains “Data Warehouse”]/Abstract`

PAT: $(Abstract \subset (Article \supset \sigma_{Data\ Warehouse'}(Title)))$

Query 3. Find the paragraphs of the “Introduction” section of each article that have the words “Data Warehousing” in its title.

XPath: `//Articles[./Title ftcontains “Data Warehousing”]/Sections/Section[@title=“Introduction”]/Paragraph`

PAT: $(Paragraph \subset (\sigma_{Title, 'Introduction'}(Section) \subset (Article \supset \sigma_{Data\ Warehousing'}(Title))))$

Query 4. Find all articles in which the surname of an author contains the value “Aberer”.

XPath: `//Article[./Surname ftcontains “Aberer”]`

PAT: $(Article \supset \sigma_{Aberer'}(Surname))$

Query 5. Find all “Summary” paragraphs of all sections (if any).

XPath: `//Section/Paragraph [@title = “Summary”]`

PAT: $(\sigma_{Title, 'Summary'}(Paragraph) \subset Section)$

Query 6. Find all paragraph containing both “OLAP” and “Multidimension” from either Article or ShortPaper.

XPath: `/(article | ShortPaper)// Paragraph[.ftcontains “Multidimension” or .ftcontain “OLAP”]`

PAT: $(\sigma_{r='OLAP'}(Paragraph) \subset Article) \cup (\sigma_{r='OLAP'}(Paragraph) \subset ShortPaper) \cap ((\sigma_{r='Multidimension'}(Paragraph) \subset Article) \cup (\sigma_{r='Multidimension'}(Paragraph) \subset ShortPaper))$

3 XML Query Equivalences

Equivalences form the basis of transformation-based query optimization. We identify three sources from which equivalences are defined.

- The set-theoretic properties of PAT. Equivalences of this type are comparable to those typically used for algebraic

query optimization in relational databases. A simple example is: $A \cup A \iff A$.

- The explicit constraints imposed by a DTD/XSD. In relational query optimization, this type of equivalences does not occur since the consistency of input queries with the database schema is normally guaranteed by the preprocessing (or parsing) of the input queries. Recognizing the possible need of querying schema-less XML data, we do not assume the general existence of the *consistency* notion regarding a query with a given DTD/XSD. Instead, we consider the DTD/XSD as an additional source to exploit for query optimization. A simple example of this type of equivalences is, $A \iff \emptyset$, provided that the element type A is not defined in the DTD.
- A “deeper” exploration of the structural properties of XML data. This is a new dimension that exists only in the context of structured documents. As opposed to the equivalences that simply check the consistency of a query with a given DTD, this class of equivalences explores the structure knowledge of XML data to a deeper level for reconstructing new consistent expressions. The resultant expressions are supposed to be superior in terms of evaluation efficiency, e.g., a structure index is enabled or an involved path gets much shortened in a query. A simple example of this type of equivalences is, $A \supset B \iff A$, if we can infer from the DTD that every A element contains a B element.

With the above sources of equivalences, we apply certain selection criteria to restrict the number of equivalences we may obtain. Otherwise, due to the relatively large number of operators in the PAT algebra and their interactions, we would obtain an unmanageably large set of potential transformation rules to consider in our system. The criteria we observe are as follows:

- The equivalences must have the potential to render a profitable transformation on their input expressions. This improvement might be obvious or at least probable from a heuristic viewpoint.
- The equivalences must not imply transformations that further complicate or expand query expressions. Although it is possible that making a query expression more complex might lead to more efficient evaluation plans, for the sake of simplicity and practicality we rule out such kind of equivalences.
- The equivalences must not require complex conditions to be checked to determine their applicability to an input query since this may cause the optimizer inefficient and hard to implement.
- The equivalences must not merely aim at generating alternative expressions. Instead, they must support deterministic, unidirectional transformations to achieve stepwise improvement on query expressions.

All equivalences take the form “ $E1 \xrightleftharpoons[c2]{c1} E2$ ”, meaning $E2$ is equivalent to $E1$ under condition $c1$, $E1$ is equivalent to

$E2$ under condition $c2$; when both $c1$ and $c2$ are omitted, the equivalence holds unconditionally.

For the purpose of compact presentation, we introduce the following shorthand notations: “ \cup ” stands for \cup and \cap ; “ σ ” stands for σ_r and $\sigma_{A,r}$; “ \supset ” represents \subset and \supset .

When there are more than one occurrence of any of the above meta-symbols in an equivalence or a transformation rule (Section 4.3), all these occurrences have to be interpreted consistently. For example, if there are two occurrences of \cup in one equivalence, if the first occurrence is interpreted as \cup , the second one has to be interpreted as \cup as well.

3.1 Set-Oriented Equivalences

This set of equivalences is based on the set-theoretic properties of the various PAT operators. Most of them are self-evident and do not need a proof. The following table lists all the equivalences of this type.

Subset laws (\mathcal{S} stand for any subset of $\tau(E)$)

$$\mathcal{E}1. \mathcal{S} - E \iff \phi$$

$$\mathcal{E}2. E \cup \mathcal{S} \iff E \text{ or } \mathcal{S} \cup E \iff E$$

$$\mathcal{E}3. E \cap \mathcal{S} \iff \mathcal{S} \text{ or } \mathcal{S} \cap E \iff \mathcal{S}$$

$$\mathcal{E}4. \sigma(\phi) \iff \phi$$

‘-’ specific laws

$$\mathcal{E}5. (E1 - E2) - E2 \iff E1 - E2$$

$$\mathcal{E}6. (E1 - E2) \cup E2 \iff E1 \cup E2$$

Commutativity

$$\mathcal{E}7. E1 \cup E2 \iff E2 \cup E1$$

$$\mathcal{E}8. E1 \cap E2 \iff E2 \cap E1$$

$$\mathcal{E}9. (E1 \supset E2) \supset E3 \iff (E1 \supset E3) \supset E2$$

$$\mathcal{E}10. \sigma_1(\sigma_2(E)) \iff \sigma_2(\sigma_1(E))$$

$$\mathcal{E}11. (E1 \supset E2) \cap E3 \iff (E1 \cap E3) \supset E2$$

Distributivity

$$\mathcal{E}12. E1 \cap (E2 \cup E3) \iff (E1 \cap E2) \cup (E1 \cap E3)$$

$$\mathcal{E}13. (E1 \cup E2) - E3 \iff (E1 - E3) \cup (E2 - E3)$$

$$\mathcal{E}14. E1 \cup (E2 \cap E3) \iff (E1 \cup E2) \cap (E1 \cup E3)$$

$$\mathcal{E}15. E1 - (E2 \cup E3) \iff (E1 - E2) - E3$$

$$\mathcal{E}16. \sigma(E1 \cup E2) \iff \sigma(E1) \cup \sigma(E2)$$

$$\mathcal{E}17. \sigma(E1 \supset E2) \iff \sigma(E1) \supset E2$$

$$\mathcal{E}18. \sigma(E1 - E2) \iff \sigma(E1) - E2$$

$$\mathcal{E}19. (E1 - E2) \cap E3 \iff (E1 \cap E3) - E2$$

$$\mathcal{E}20. (E1 \cup E2) \supset E3 \iff (E1 \supset E3) \cup (E2 \supset E3)$$

$$\mathcal{E}21. E1 \supset (E2 \cup E3) \iff (E1 \supset E2) \cup (E1 \supset E3)$$

$$\mathcal{E}22. (E1 - E2) \supset E3 \iff (E1 \supset E3) - E2$$

$$\mathcal{E}23. E1 \supset (E2 - E3) \iff (E1 \supset E2) - (E1 \supset E3)$$

Associativity

$$\mathcal{E}24. E1 \cup (E2 \cup E3) \iff (E1 \cup E2) \cup E3$$

$$\mathcal{E}25. E1 \cap (E2 \cap E3) \iff (E1 \cap E2) \cap E3$$

The subset laws in the above table summarize the simplification of set expressions involving a *subset*. Notice that from the subset laws [23], more specific laws, like the idempotency laws, can be derived by replacing \mathcal{S} with E or ϕ . Equivalence $\mathcal{E}10$ also applies to the mixed case of σ_r and $\sigma_{A,r}$.

The proof of most above rules is trivial except for \supseteq commutativity rules, which could easily get confused with the properties of \forall operators. Though syntactically similar, the \supset and \subset operators are asymmetric in nature, where the first operand relates to the result type while the second relates to the condition for selecting elements from the first operands' extent.

In the following, as an example, we give the proof of the \supseteq commutativity, $\mathcal{E}9$, regarding the \subset operator only.

Proof (\subset commutativity rule)

$$\begin{aligned} & (E1 \subset E2) \subset E3 \\ &= \{e1 \in \text{ext}(E1 \subset E2) \mid \exists e3 \in \text{ext}(E3) \text{ s.t. } e1 \text{ is contained} \\ & \quad \text{in } e3 \} \\ &= \{e1 \in \text{ext}(E1) \mid \exists e2 \in \text{ext}(E2) \text{ s.t. } e1 \text{ is contained in } e2 \\ & \quad \text{and } \exists e3 \in \text{ext}(E3) \text{ s.t. } e1 \text{ is contained in } e3 \} \\ &= \{e1 \in \text{ext}(E1 \subset E3) \mid \exists e2 \in \text{ext}(E2) \text{ s.t. } e1 \text{ is contained} \\ & \quad \text{in } e2 \} \\ &= (E1 \subset E3) \subset E2 \end{aligned}$$

■

3.2 Equivalences Based on Explicit DTD Constraints

The equivalences given in this subsection are based on checking whether a PAT operator is consistent with the given DTD. An operator can be simply an *etn* or any of the operators, $\sigma_{A,r}$, $\sigma_r(E)$, \subset or \supset . This results in the following equivalences:

$$\mathcal{E}26. \text{etn} \iff \phi \text{ if etn is not defined}$$

$$\mathcal{E}27. \sigma_{A,r}(E) \iff \phi \text{ if 'A' is not an attribute of } \tau(E)$$

$$\mathcal{E}28. \sigma_r(E) \iff \phi \text{ if } \tau(E) \text{ doesn't directly contain predefined value "PCDATA"}$$

$$\mathcal{E}29. E1 \subset E2 \iff \phi \text{ if } \tau(E1) \text{ is not contained in } \tau(E2)$$

$$\mathcal{E}30. E1 \supset E2 \iff \phi \text{ if } \tau(E1) \text{ does not contain } \tau(E2)$$

Note that further equivalences reducing expressions to an empty set exist, but they need to take more than just one operator into account for deducing an empty result, thus are precluded from our consideration according to the criteria mentioned earlier. For example, $(E1 \subset E2) \subset E3 \iff \phi$ if $\tau(E2)$ and $\tau(E3)$ have no containment relationship; this rule is precluded because it needs to investigate two \subset operators.

However, this example represents a rather specific situation: the counterpart of it with regard to the operator \supset does not hold because in the context of XML it is common for an external element to contain multiple internal ones, but an internal element cannot to be shared between two (or more) external elements if the external elements do not hold a containment relationship between themselves.

In principle, it is also possible to exploit DTD knowledge in even more complicated ways in order to maximally expand the search space for the optimization of a given query expression. Again, as this conflicts with our design goal, we do not move further along this direction.

3.3 Equivalences Based on Particular XML Structural Properties

Although in the previous subsection we have ruled out the equivalences that incorporate complicated conditions, there exist specific situations where the structure of XML data may be exploited very profitably towards query optimization and thus are worthy of special consideration. We identify several such special situations and present the equivalences in this subsection.

First, we introduce three important notions, *exclusivity*, *obligation*, and *entrance location* regarding the structures of XML data (these notions were initially introduced in [4]).

In a given DTD, some element types may be shared among others. For example, element type *Authors* in Fig. 3 is contained in element type *Article* and in *ShortPaper*. However, some other types may not be shared, i.e., *exclusively contained in* a single type — this information bears great potential for significant query optimization.

Definition 7 (Exclusivity) *Element type* ET_j *is exclusively contained in element type* ET_i *if each path* (e_j, \dots, e_k) *with* e_j *being an element of type* ET_j *and* e_k *being the document root contains an element of type* ET_i . *Conversely, element type* ET_i *exclusively contains* ET_j *if the condition holds.*

Definition 8 (Free PAT expression) *Let* E *be a PAT expression. E is said to be free, denoted as* $\text{free}(E)$, *if evaluation of the expression* E *returns the full extent of* $\tau(E)$.

One typical case that the $\text{free}(E)$ condition holds is when E is simply an *etn*. In principle, we should have other forms of *free* expressions. As per its definition, other-than-*etn* free expressions, say, E , indicates that the selection condition imposed by the expression E is redundant. According to our optimization approach, after the normalization phase (Section 4) all other-than-*etn* free expressions will be simplified as plain *etn*'s. So, for now we only need to concern free expressions that are plain *etn*'s. The concept of *free* expression plays an important role in many of our equivalences (and transformation rules) like in the following:

$$\mathcal{E}31. E1 \subset E2 \iff E1 \text{ if } \tau(E1) \text{ is exclusively contained in } \tau(E2) \text{ and } \text{free}(E2)$$

This rule's condition states that $\tau(E1)$ is exclusively contained in $\tau(E2)$ and the subexpression $E2$ returns the full extent of $\tau(E2)$. In other words, the \subset operation in $E1 \subset E2$ does not impose any selection condition on $\text{ext}(E1)$, therefore the equivalence holds.

It is obvious from the operators' definitions that the expression " $E1 \subset E2$ " and " $E2 \supset E1$ " are not the same, thus the

exclusivity notion cannot be used to transform queries involving subexpressions of the second form. We need to introduce the counterpart of the exclusivity notion, *obligation*, for this purpose.

While *exclusivity* reflects “a perspective from the internal element type”, i.e., whether an internal element type is exclusively contained in an external one, the *obligation* notion that is to be introduced highlights the perspective from an external element type with regard to the \supset operation.

Definition 9 (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. Conversely, we say that ET_j is obligatorily contained in ET_i .

The concept of obligation gives rise to the following equivalence.

$\mathcal{E}32.$ $E1 \supset E2 \iff E1$ if $\tau(E1)$ obligatorily contains $\tau(E2)$ and $free(E2)$

Analogously, we cannot use obligation to transform “ $E2 \subset E1$ ” to “ $E2$ ”.

If two element types are not related by *exclusivity* and *obligation*, it may be worthwhile to check whether there exists yet a third element type, called *entrance location*, which may bring in significant optimization opportunities.

Definition 10 (Entrance location) For a given DTD, element type E is an entrance location (denoted as *EL* for short) for $\tau(E1)$ and $\tau(E2)$ if in any document complying with the DTD, all paths from an element $e1$ of $\tau(E1)$ to an element $e2$ of $\tau(E2)$ pass through an element e of type E .

As a special case, $\tau(E1)$ and $\tau(E2)$ are entrance locations for themselves. An entrance location is an *etn*, and the *free* condition naturally holds on it (refer to Definition 8).

The notion of *entrance location* gives rise to the following equivalences.

$\mathcal{E}33.$ $E1 \supseteq E2 \iff E1 \supseteq (E3 \supseteq E2)$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$

$\mathcal{E}34.$ $E1 \supseteq E2 \iff (E1 \supseteq E3) \supseteq E2$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$

$\mathcal{E}35.$ $(E1 \supseteq E2) \supseteq E3 \iff E1 \supseteq (E2 \supseteq E3)$ if $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$

$\mathcal{E}36.$ $(E1 \supseteq E2) \supseteq E3 \iff E1 \supseteq (E3 \supseteq E2)$ if $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$

$\mathcal{E}37.$ $(E1 \subset E2) \supset E3 \iff E1 \subset (E2 \supset E3)$ if $\tau(E1)$ is an EL for $\tau(E2)$ and $\tau(E3)$

$\mathcal{E}38.$ $(E1 \supset E3) \subset E2 \iff E1 \supset (E3 \subset E2)$ if $\tau(E1)$ is an EL for $\tau(E2)$ and $\tau(E3)$

It is important to point out: (1) the condition “ E is an entrance location for...” (as used in $\mathcal{E}33$) is not the same as “ $\tau(E)$ is an entrance location for...” (as used in $\mathcal{E}35$) (with the former, E has to be an *etn* and $free(E)$ naturally holds, while with latter, E can be any legal PAT expression); (2)

considering the \supseteq associativity ($\mathcal{E}35$), we could have had an alternative form of $\mathcal{E}36$ which (on its right hand side) associates $E1$ and $E3$ first. $\mathcal{E}37$ and $\mathcal{E}38$ represent two generalized associativity laws of the \supseteq operations.

Usually, introducing an extra element type to a PAT expression is detrimental to the evaluation efficiency of the query. Hence, leftward application of $\mathcal{E}33$ and $\mathcal{E}34$ is obviously favorable, and rightward application may be potentially useful only under a very specific condition. We will address this issue with more details in Section 4.

In the following, we discuss the equivalences that make combined application of *entrance location* and *exclusivity* or *obligation*.

Entrance location and exclusivity

$\mathcal{E}39.$ $E1 \subset E2 \iff E1 \subset E3$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$, $\tau(E3)$ is exclusively contained in $\tau(E2)$, and $free(E2)$

The equality of the two sides in $\mathcal{E}39$ promptly comes from an omitted intermediate term, $E1 \subset (E3 \subset E2)$.

Entrance location and obligation

$\mathcal{E}40.$ $E1 \supset E2 \iff E1 \supset E3$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$, $\tau(E3)$ obligatorily contains $\tau(E2)$, and $free(E2)$

$\mathcal{E}41.$ $E1 \subset E2 \xleftrightarrow[c2]{c1} E1 \supset (E3 \subset E2)$ if $c1$: $\tau(E1)$ obligatorily contains $\tau(E3)$; $c2$: $\tau(E1)$ is an EL for $\tau(E2)$ and $\tau(E3)$ and $free(E3)$ holds

$\mathcal{E}42.$ $E1 \supset E2 \xleftrightarrow[c2]{c1} E1 \subset (E3 \supset E2)$ if $c1$: $\tau(E1)$ is obligatorily contained in $\tau(E3)$; $c2$: $\tau(E1)$ is an EL for $\tau(E3)$ and $\tau(E2)$ and $free(E3)$ holds

As with $\mathcal{E}39$, in $\mathcal{E}40$, a similar intermediate expression, which is omitted, makes the proof of $\mathcal{E}40$ straightforward. Combined use of “entrance location” and “obligation” further brings in $\mathcal{E}41$ and $\mathcal{E}42$. We provide a proof for $\mathcal{E}41$ ($\mathcal{E}42$ can be analogously proven).

Proof ($\mathcal{E}41$):

\Rightarrow : $\forall e1 \in (E1 \subset E2)$, $\exists e2 \in E2$ s.t. $e1 \subset e2$; because $\tau(E1)$ obligatorily contains $\tau(E3)$, $\exists e3 \in E3$ s.t. $e1 \supset e3$, which is the same as to say $e3 \subset e1$; therefore, $e3 \subset e2$; put together, we have $e1 \supset (e3 \subset e2)$, i.e., $e1 \in (E1 \supset (E2 \subset E2))$.

\Leftarrow : $\forall e1 \in (E1 \supset (E3 \subset E2))$, $\exists e3 \in (E3 \subset E2)$ s.t. $e1 \supset e3$ and $\exists e2 \in E2$ s.t. $e3 \subset e2$; in other words, we have both $e3 \subset e1$ and $e3 \subset e2$; because with well-formed XML data, an element is not allowed to be contained in two other elements who do not have a containment between themselves, we shall have either $e1 \subset e2$ or $e1 \supset e2$; as we are given $\tau(E1)$ is an EL for $\tau(E2)$ and $\tau(E3)$, so we have $e1 \subset e2$, i.e., $e1 \in (E1 \subset E2)$. \blacksquare

More complex equivalences of this type may be identified, but they conflict with our design goal and hence are not under further discussion (examples can be found in [4]).

4 Deterministic XML Query Optimization

This section systematically addresses our approach to XML query optimization, including our methodology, control strategy, transformation rules, and the organization of our rule system.

We envisage that the optimizer developed according to our approach is to be used in a highly dynamic environment, like ad-hoc querying or on the WWW. Therefore, the high efficiency of query optimization is the predominant goal of our research. Due to the high complexity of the XML data model and the large search space for an optimal query plan, we set up a very practical goal for our approach: quickly obtain obvious improvements on input queries in terms of evaluation efficiency instead of identifying an optimal plan which can be extremely time-consuming. Under this circumstance, we gave up the classical cost-based and heuristic-based approaches, but resorted to a rather radical way to prune the potentially huge search space for the optimization of XML queries. The strategy we came up with is to perform exclusively deterministic transformations on query expressions. By its nature, this type of transformation is heuristic-based, and is required to lead to obvious, step-by-step improvements on input queries until a final expression, regarded as the “optimal” one, is reached. This idea is well facilitated by the rich semantics of XML data and XML queries.

Enabling an application of a potential structure index into a query is at the core of our approach. Indices have long been used as an effective way of improving the query evaluation efficiency in a database system. However, the temptation to create too many indices in a database should be confined because every index comes with a maintenance cost when the database evolves. A practical solution is to build a small number of indices but exploit them fully during query optimization. With XML structured documents, indices, although available, may not be applicable to a query because of the unfavorable structure pattern that the query is in. Therefore, to a large extent, our optimization effort is to reveal potential indices and to facilitate their application to queries through heuristic-based transformations.

In the following, we first discuss the indexing issue in the context of structured-documents databases.

4.1 Structure Indices

We consider three types of index structures: element content indices, attribute content indices, and structure indices that short-circuit different types of elements related through certain structural relationship, e.g., containment.

Element content indexing has been extensively exploited in IR (information retrieval) systems. Attribute content indexing is analogous to the classical indices as pursued in conventional DBMSs. Both element content index and attribute content index may be generally referred to as *selection* indices. *Structure indexing* is specific to highly structured data like XML data, of which the data model is based on a complex structure, usually a graph or a tree structure. In the XML

data model, there are two general types of structural relationships — containment and referencing. In this article, we limit ourselves only to the containment relationship.

When the containment relationship between the elements of two different element types is expected to be frequently taken as a querying criterion (or has the potential of being used to speedup a group of “nearby” queries), a structure index is typically and favorably built between the two element types. Making the decision of building a structure index between two element types is analogous to that of building a superhighway between two geographical places: (1) the two places should be two important places, e.g., two important cities; (2) the two places should both cover a large population; (3) as many nearby areas as possible should benefit from the superhighway. This metaphor implies that (1) the two element types chosen for indexing should be among the ones that are most frequently referred by potential user queries; (2) they should have relatively large extents; (3) the structure index built should also have the potential of speeding up the evaluation of other query conditions that are structurally “close” to the element types covered by the index.

A structure index may be intuitively thought as a “short-cut” between the elements of two element types, and can be defined either from an external type to an internal one or vice versa, corresponding to the “contains” or “contained-in” relationship, and being called as *downward index* or *upward index*, respectively. As an example, the index from `Article` to `Paragraph` in Fig. 3 can help quickly build up the connections between the articles and their paragraph components.

In the following, we formally define the structure indices as mappings.

Definition 11 (Downward structure index) *Let $T1$ and $T2$ be two element types in a given DTD and $T1$ is external to $T2$, a structure index from type $T1$ to type $T2$ is called a downward structure index and is defined by the following partial mapping: $I_{T2}^{\downarrow}(T1) : ext(T1) \rightarrow \mathcal{P}(ext(T2))$.*

The definition utilizes a *partial* mapping because some $T1$ elements may not contain $T2$ element(s). The index, $I_{Paragraph}^{\downarrow}(Article)$, as an example, is a downward index from type `Article` to type `Paragraph`.

Definition 12 (Upward structure index) *Let $T1$ and $T2$ be two element types in a given DTD and $T1$ is internal to $T2$, a structure index from type $T1$ to type $T2$ is called an upward structure index and is defined by the following partial mapping: $I_{T2}^{\uparrow}(T1) : ext(T1) \rightarrow ext(T2)$.*

In this definition, a partial mapping is used again because of a similar reason as in the previous definition. As an example, $I_{Article}^{\uparrow}(Name)$ represents an upward index from `Name` to `Article`.

Notice that, according to the syntax rule of XML, an external element may contain multiple instances of the same internal element type, but an internal element may not be shared by multiple instances of the same external element type. This fact leads to the asymmetry of the above index definitions,

i.e., the *power set* notion is used in the *downward indexes* but not in *upward indexes*. Nevertheless, for ease of presentation, we will use $I_{T_2}(T_1)$ as a general notation for a structure index defined *from* type T1 and type T2 no matter whether it is *upward* or *downward*, and use $I_{T_1-T_2}$ to denote a structure index *between* T1 and T2. Accordingly, we introduce the following new equivalence to interpolate a structure index operation into a containment query.

ℰ43. $E1 \supseteq E2 \iff I_{\tau(E1)}(E2) \cap E1$ if a structure index between $\tau(E1)$ and $\tau(E2)$ is available

When $free(E1)$ holds, $I_{\tau(E1)}(E2)$ is subset of $ext(\tau(E1))$, the intersection is thus redundant and can be taken away.

A more general form of the expression $E1$ might be $(\sigma_1 \dots \sigma_m(\tau(E1)))$, denoted as $\sigma(\tau(E1))$ for short, where the innermost subexpression, $\tau(E1)$, corresponds to a plain *etn*, while $\sigma_1 \dots \sigma_m$ is a sequence of selection operations defined by the PAT algebra. Under this assumption, we have the following equivalence that pulls up the σ and eliminates the \cap :

ℰ44. $I_{\tau(E1)}(E2) \cap \sigma(\tau(E1)) \iff \sigma(I_{\tau(E1)}(E2))$

The proof of this equivalence is straightforward: after the σ being pulled up, $I_{\tau(E1)}(E2)$ is a subset of $ext(\tau(E1))$ and thus the \cap can be removed.

The equivalence ℰ44 reveals two alternatives for optimization: the left hand side allows the use of an attribute or content index for the selection operation, σ , if the index is available, but unfavorably induces an extra \cap operation; while the right hand side deprives the priority of using an available index on the σ , but saves the cost of evaluating the extra \cap . Which way is better shall be determined by performing estimated cost comparison, which is beyond the scope of this article.

Finally, before we end this subsection, let us point out the relationship between the notion of our structure indices and the structural join algorithm [40]. Structural join is a better alternative to the traditional path-chasing method of evaluating a containment operation. Our structure indices are actually *pre-computed* structural joins. Even with highly skewed data, e.g., with $I_{T_2}(T_1)$, if the majority of T1 elements do not relate to T2 elements via a containment relationship, the use of the structure index is still beneficial because of the “pre-computed” nature of it, although under this circumstance, the superiority of using a structure index gets minimized. Alternatively, we may choose not to build structure indices when certain data is highly skewed according to available statistics.

4.2 Strategy Overview

As indicated, the intrinsic complexity of the XML data model calls upon a large number of operations of an underlying algebra in order to facilitate effective XML query optimization; the operations and their interactions render a much enlarged search space for optimal plans during query optimization, compared with the case in relational databases. To help solve this problem, our approach pursues the heuristic-based

optimization approach to its extreme: *exclusively* apply *deterministic* transformations on queries to achieve fast optimization.

Traditionally, equivalences are used to generate more alternative plans from which an “optimal” one is to be chosen. In our work, equivalences (Section 3) are not to be used in this traditional way, but to derive deterministic transformation rules with the incorporation of relevant optimization heuristics. The deterministic transformation rules are then applied to each query for optimization.

The focus of our approach is not on the optimality of an output query (plan) but on the efficient achievability of a definite, nontrivial improvement on the query. Therefore, the primary goal of our approach is to discover opportunities to the largest extent where dramatic improvements in terms of evaluation efficiency can be obtained by deterministic transformations. This idea resorts to proper exploitation of the structure-based semantics of XML data and XML queries. In general, semantic transformation for query optimization is best conducted starting from a carefully chosen “standard” format. After semantic transformation (which can be dramatic), a cleaning-up phase is typically needed because semantic transformation, while making dramatic changes on input queries, may introduce new (syntactical) redundancies in the query. As a result, our approach organizes logical query optimization into three consecutive phases: normalization, semantic optimization, and simplification (or cleaning-up) as illustrated in Fig. 4. The first phase mainly reorders the operators in an input query. The second phase carries out the critical task — semantic transformations for optimization — with resort to extensive exploitation of the structure knowledge of XML data. The third phase makes a thorough cleaning-up, and the final output is the optimized query. We discuss the details of each phase below.

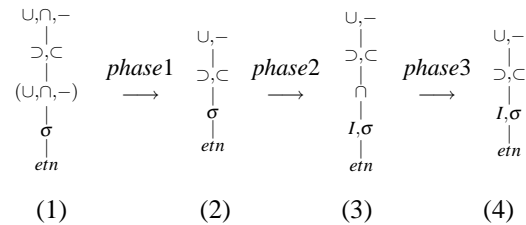


Fig. 4 Illustration of transformation strategy for query optimization

Phase 1: Query normalization. Phase 1 normalizes all incoming queries to obtain pattern (2) of Fig. 4. During this phase, the algebraic properties of the PAT algebra are exploited to achieve the following goals:

1. Isolate different types of operators at different levels (as in pattern (2) of Fig. 4).
2. Completely eliminate the \cap operator from the query expression.
3. More selective operations are pushed down (i.e., performed earlier).

The complete elimination of the \cap operator from a PAT expression is an accompanying benefit of applying algebraic transformations on PAT expressions (see Proposition 2 in Section 4.3.1). This represents an interesting property of our extended PAT algebra and a significant type of simplification to XML queries.

Pushing the σ operators to the bottom level can simplify the identification of potential attribute and content indices. In addition, as a heuristic, we assume the selection operators are the most beneficial ones to evaluate first. The isolation of the \supseteq operators at the level above is an important prerequisite for revealing the opportunities of exploiting a potential structure index. Collecting the remaining set operators into the top level is backed by the heuristic of performing the least selective operations as late as possible.

We observe that in most user queries formatted as PAT expressions, the selection operations are specified more early and the set operators are typically applied to the selection results. So we expect most incoming queries are in a format similar to pattern (1) (Fig. 4). That is, the incoming queries are already quite close to the ones obtained after normalization. The normalization phase also performs obvious simplifications, like transforming “ $E \cup E$ ” to a single E , to make the subsequent semantic optimization more efficient.

Phase 2: Semantic transformation. The second phase transforms pattern (2) queries to pattern (3) with the primary goal of identifying the opportunities of applying potential structure indices. When an index operator, denoted by I , is interpolated into a query expression, an intersection is typically reintroduced (which may be eliminated again at the third phase). Identification of applicable element and attribute content indices is fairly simple. The potential applicability of certain structure indices is explored next at this phase. If directly introducing a structure index into a query is not feasible, a reduction of certain paths involved in the query may still be possible and worthwhile, and is pursued lastly at this phase with resort to particular structure knowledge about the XML data.

Phase 3: Query simplification. Semantic transformations typically bring in new subexpressions, including new *etn*’s and new PAT operators, in particular the index and intersection operators. Hence, a further run of the simplification rules of Phase 1 is desired, and additional simplification rules are needed to handle the combination of the index operator with the \cap operator (The \cap may be reintroduced by an index interpolation rule, e.g., $\mathcal{R}49$).

4.3 The Rule System

In this subsection, we discuss the transformation rules exploited by our approach.

Transformation rules are distinct from the more general equivalences (see Section 3). Equivalences are *bidirectional* in the sense that they can be applied from left-hand-side to

right-hand-side, or vice versa. This process is notoriously time-expensive because of the uncontrollable behavior of the rules at runtime. To serve the goal of our approach, we derive transformation rules from the presented equivalences. In contrast to equivalences, transformation rules are *unidirectional*, taking the form “ $(E1) \xRightarrow{C} (E2)$ ”, with an optional pre-condition C , which decides the applicability of the rule to an incoming expression in combination with the rule’s input pattern. All transformation rules in our approach are intrinsically *deterministic* as, once applied, a transformation brings in definite improvement on its input query. Transformation rules are derived from individual (or combination of) equivalences with resort to general or specific optimization heuristics pertaining to XML queries. Relevant and important heuristics will be discussed when the transformation rules are presented.

4.3.1 Rule Group 1: Normalization Rules Query normalization covers three aspects: enforcement of explicit DTD-constraints, operator reordering, and simplification.

Rule group 1.1: Enforcing DTD-constraints

This group of rules directly come from the five corresponding equivalences, $\mathcal{E}26$ to $\mathcal{E}30$, introduced in Section 3.2 pertaining to explicit DTD-constraints. We omit these five rules herein but reserve the numbers, $\mathcal{R}1$ to $\mathcal{R}5$, for keeping this article consistent with our system and other publications.

Rule group 1.2: Operator reordering

This group of rules produce query expressions free of the \cap operator, while the other operators are re-permuted in the following order: σ at the bottom, \supseteq in the middle, and \cup and $-$ (but not \cap) on the top.

Rule group 1.2.1: Pulling-up “ \cup ” and “ $-$ ”. The following rules pull-up the “ \cup ” and “ $-$ ” operators as far as possible because they are relatively expensive operators.

- $\mathcal{R}6.$ $\sigma(E1 \cup E2) \implies \sigma(E1) \cup \sigma(E2)$
- $\mathcal{R}7.$ $E1 \cap (E2 \cup E3) \implies (E1 \cap E2) \cup (E1 \cap E3)$
- $\mathcal{R}8.$ $(E1 \cup E2) \cap E3 \implies (E1 \cap E3) \cup (E2 \cap E3)$
- $\mathcal{R}9.$ $(E1 \cup E2) - E3 \implies (E1 - E3) \cup (E2 - E3)$
- $\mathcal{R}10.$ $E1 - (E2 \cup E3) \implies (E1 - E2) - E3$
- $\mathcal{R}11.$ $(E1 \cup E2) \supseteq E3 \implies (E1 \supseteq E3) \cup (E2 \supseteq E3)$
- $\mathcal{R}12.$ $E1 \supseteq (E2 \cup E3) \implies (E1 \supseteq E2) \cup (E1 \supseteq E3)$
- $\mathcal{R}13.$ $\sigma(E1 - E2) \implies \sigma(E1) - E2$
- $\mathcal{R}14.$ $(E1 - E2) \cap E3 \implies (E1 \cap E3) - E2$
- $\mathcal{R}15.$ $(E1 - E2) \supseteq E3 \implies (E1 \supseteq E3) - E2$
- $\mathcal{R}16.$ $E1 \supseteq (E2 - E3) \implies (E1 \supseteq E2) - (E1 \supseteq E3)$

Proposition 1 *An expression to which the rules in group 1.2.1 cannot be applied, does not contain \cup and $-$ in any argument of σ , \supseteq , and \cap operators.*

Proof. For every possible occurrence of \cup and $-$ in the arguments of other operators, there exists a rule that removes this occurrence. ■

It is worthwhile to point out that the transformation made by $\mathcal{R}16$ may not itself be beneficial, but it facilitates the subsequent semantic transformation phase to discover important optimization opportunities.

Rule group 1.2.2: Removing \cap operators. The following rules push \cap down through all σ and \supseteq operators and eventually get all \cap removed.

$$\mathcal{R}17. (E1 \supseteq E2) \cap E3 \implies (E1 \cap E3) \supseteq E2$$

$$\mathcal{R}18. E1 \cap (E2 \supseteq E3) \implies (E1 \cap E2) \supseteq E3$$

$$\mathcal{R}19. \sigma(E1) \cap E2 \implies \sigma(E1 \cap E2)$$

$$\mathcal{R}20. E1 \cap \sigma(E2) \implies \sigma(E1 \cap E2)$$

$$\mathcal{R}21. E \cap \phi \implies \phi$$

$$\mathcal{R}22. \phi \cap E \implies \phi$$

$$\mathcal{R}23. E \cap E \implies E$$

Proposition 2 *An expression to which the rules in group 1.2.1 and group 1.2.2 cannot be applied, does not contain any \cap operator.*

Proof. For every possible occurrence of other operators in the arguments of a \cap , there exists a rule that removes this occurrence. Thus the resulting expression does not contain any other operator in the arguments of \cap . The only allowed occurrences of \cap must have an *etn* or ϕ as one of its arguments. Due to type compatibility, only the same *etn* can occur as arguments on both sides of a \cap operator at the lowest level. These occurrences are removed by rule $\mathcal{R}21$, $\mathcal{R}22$, $\mathcal{R}23$, thus the resulting expression cannot contain any \cap operator. ■

Rule group 1.2.3: Pushing σ down (below \supseteq). The following rule reorders the σ and \supseteq operators, and the precedence is given to σ .

$$\mathcal{R}24. \sigma(E1 \supseteq E2) \implies \sigma(E1) \supseteq E2 \quad (\text{from } \mathcal{E}17)$$

Rule $\mathcal{R}24$ is based on the following heuristic: \supseteq operators are more expensive than σ operators and thus should be pulled up. This heuristic can be justified by simply counting the number of element accesses imposed by each side of $\mathcal{R}24$: assuming $E1$ and $E2$ have the same cardinality M , \supseteq and σ have the same selectivity $1/N$, the cost of the left-hand side is, $M + M * M + M/N = M * (1 + 1/N + M)$, which is much larger than the cost of the right-hand side, $M + M/N + (M/N) * M = M * (1 + 1/N + M/N)$, as long as the database is not too sparsely populated and the selectivities of the operators are not too low (i.e., $M > 1$, and $1/N$ is sufficiently small).

Rule group 1.2.4: Removing \supseteq operators. The following rules remove redundant \supseteq operators.

$$\mathcal{R}25. (E1 \supseteq E2) \supseteq E2 \implies E1 \supseteq E2$$

$$\mathcal{R}26. (E1 \supseteq E2) \supseteq E1 \implies E1 \supseteq E2$$

$$\mathcal{R}27. E \supseteq \sigma(E) \implies \sigma(E)$$

$$\mathcal{R}28. \sigma(E) \supseteq E \implies \sigma(E)$$

$$\mathcal{R}29. E \supseteq \phi \implies \phi$$

$$\mathcal{R}30. \phi \supseteq E \implies \phi$$

Rule group 1.3: Simplification

We observe that some obvious simplifications carried out at an early stage are beneficial, e.g., $E \subset E \implies E$. But some more complex simplifications if hurriedly carried at an early stage may make the whole optimization process quickly captured at a local optimal. For example, while a leftward application of $\mathcal{E}33$ eliminates $E3$, it takes away the opportunity of using the structure index defined between $\tau(E3)$ and $\tau(E2)$ if the index is available; otherwise, a reverse transformation later on is needed to re-enable the lost opportunity. Therefore, Phase 1 shall only invoke obvious simplifications, which are all related with the set operations. The simplification rules considered at this point are all derived from the subset law. When the subset law is applied to simple augments like E and ϕ , we get the rules that deal with a single operator. When it is applied to subexpression augments like $\sigma(E1, r)$, $(E1 \supseteq E2)$, and $(E1 - E2)$, we obtain relatively complex rules handling double operators. We do not concern further complicated situations as it is against our design goal. The simplification rules are listed below.

$$\mathcal{R}31. \phi - E \implies \phi$$

$$\mathcal{R}32. E - \phi \implies E$$

$$\mathcal{R}33. E - E \implies \phi$$

$$\mathcal{R}34. \phi \cup E \implies E$$

$$\mathcal{R}35. E \cup E \implies E$$

$$\mathcal{R}36. \sigma(E) - E \implies \phi$$

$$\mathcal{R}37. \sigma(E) \cup E \implies E$$

$$\mathcal{R}38. (E1 \supseteq E2) - E1 \implies \phi$$

$$\mathcal{R}39. (E1 \supseteq E2) \cup E1 \implies E1$$

$$\mathcal{R}40. E1 \cup (E1 \supseteq E2) \implies E1$$

$$\mathcal{R}41. (E1 - E2) - E1 \implies \phi$$

$$\mathcal{R}42. (E1 - E2) \cup E1 \implies E1$$

$$\mathcal{R}43. (E1 - E2) - E2 \implies E1 - E2$$

$$\mathcal{R}44. (E1 - E2) \cup E2 \implies E1 \cup E2$$

$$\mathcal{R}45. (E1 \cup E2) \cup E1 \implies E1 \cup E2$$

$$\mathcal{R}46. (E1 \cup E2) \cup E2 \implies E1 \cup E2$$

The above rules are instantiated from different cases of applying the subset laws except for $\mathcal{R}43$ and $\mathcal{R}44$, which are derived from the two “-” equivalences, $\mathcal{E}5$ and $\mathcal{E}6$, respectively.

4.3.2 Rule Group 2: Semantic Transformation Rules Exploiting element and attribute indices to achieve query optimization is relatively simple, while the exploitation of structure indices needs dedicated and specialized transformation rules. Our semantic transformation phase focuses on exploiting the opportunities of using an available structure index or enabling such an opportunity to applying a structure index.

Exploiting indices for σ operators

After Phase 1, with all \cap operations being eliminated and σ operations being pushed down to the bottom, σ operations, if applying to the same base element type, are juxtaposed. The following rule gives precedence to the σ operator for using an available selection index.

R47. $\sigma_1(\sigma_2(E)) \implies \sigma_2(\sigma_1(E))$ if a proper index for σ_1 is available

While R47 makes using an available selection index possible, the following rule makes this possibility a reality.

R48. $\sigma(etn) \implies I_\sigma(etn)$ if the corresponding index I_σ exists

R48 introduces an application of a selection index, denoted as I_σ .

Exploiting structure indices for \supseteq operations

We are now ready to address the most interesting part of our work — semantic transformations — that identify the most profitable opportunity of optimizing a query by introducing structure indices into the query. The following three situations have been differentiated to help identify the opportunity:

- The most beneficial situation is that a structure index is available and is thus applied.
- A slightly less beneficial situation is when a structure index is available but is not superficially applicable because of an unfavorable composition pattern that the query is in, e.g., the two element types covered by a structure index do not directly interact in the query expression. However, the potential can be developed by heuristics-guided query transformations.
- The third situation is when no structure index is available, however, there exists a third element type that can be interpolated, as an EL, to reveal a hidden opportunity of applying a structure index to the query expression.

In the following, we present the transformation rules according to the three situations. All these rules are based on a common heuristic: applying a structure index is the most profitable act in XML query optimization (as structure indices in our setting are pre-computed structural joins).

Exploiting available and applicable structure indices. If between the two *etn*'s related via a containment operation in a query, there exists a structure index that is applicable, we simply apply the index to the query by activating the following transformation:

R49. $E1 \supseteq E2 \implies I_{\tau(E1)}(E2) \cap E1$ if a structure index is available between $\tau(E1)$ and $\tau(E2)$

Enabling available but “inapplicable” structure indices. If a structure index is available but not immediately applicable to a query expression, there may still be the chance to enable this opportunity. In this situation, our strategy is to use

the \supseteq commutativity and associativity rules, individually or in combination, to make an available structure index explicitly applicable. We come up with the following rules handling this situation.

R50. $(E1 \supseteq E2) \supseteq E3 \implies (E1 \supseteq E3) \supseteq E2$

R51. $E1 \supseteq (E2 \supseteq E3) \implies (E1 \supseteq E2) \supseteq E3$ if there exists a structure index between $\tau(E1)$ and $\tau(E2)$, and $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$

R52. $E1 \supseteq (E3 \supseteq E2) \implies (E1 \supseteq E2) \supseteq E3$ if there exists a structure index between $\tau(E1)$ and $\tau(E2)$, and $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$

R53. $E1 \subset (E2 \supset E3) \implies (E1 \subset E2) \supset E3$ if there exists a structure index between $\tau(E1)$ and $\tau(E2)$, and $\tau(E1)$ is an EL for $\tau(E2)$ and $\tau(E3)$

R54. $E1 \supset (E2 \subset E3) \implies (E1 \supset E2) \subset E3$ if there exists a structure index between $\tau(E1)$ and $\tau(E2)$, and $\tau(E1)$ is an EL for $\tau(E2)$ and $\tau(E3)$

Rule R50 comes from E9, while R51 - R54 are the leftward instantiation of the equivalences E35 - E38, respectively.

Enabling “impossible” structure indices. If the element types involved in a query do not allow the use of a structure index, we examine whether it is possible to “import” a new element type to enable such an opportunity. In our framework, a new element type can be imported into a query expression only if it is an entrance location. We have identified eight such cases that render the opportunity of applying a structure index, which is otherwise “impossible”. In the first four cases (Fig. 5) element type substitution is performed, which asks for additional conditions such as obligation or exclusivity being held. Once the transformation is done, the index is then applied by R49 during the next iteration.

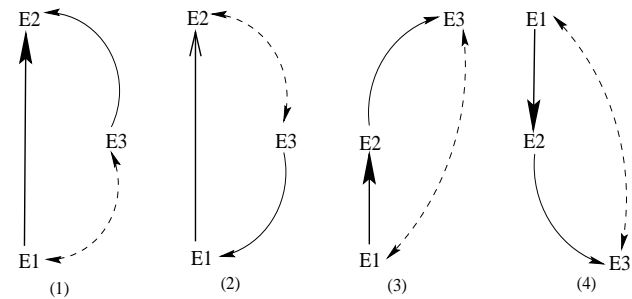


Fig. 5 Exploring the opportunities for using structure indices (case 1 to 4)

R55. $E1 \subset E2 \implies E1 \subset E3$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$ and is exclusively contained in $\tau(E2)$, $free(E2)$ holds, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

R56. $E1 \supset E2 \implies E1 \supset E3$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$ and obligatorily contains $\tau(E2)$, $free(E2)$ holds, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

R57. $E1 \subset E2 \implies E1 \subset E3$ if $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$ and is exclusively contained in $\tau(E3)$, $free(E2)$ holds, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

R58. $E1 \supset E2 \implies E1 \supset E3$ if $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$ and obligatorily contains $\tau(E3)$, $free(E2)$ holds, and a structure index between $\tau(E3)$ and $\tau(E1)$ is available

Rule R55, R56, R57, and R58 are illustrated by Fig. 5. (corresponding to case (1), (2), (3), and (4), respectively). R55 and R57 are derived from E39, while R56 and R58 are from E40.

The remaining four cases are relatively less beneficial (Fig. 6) since the introduction of a new element type (an EL) makes a sheer expansion of the input expression. The four transformation rules are presented below.

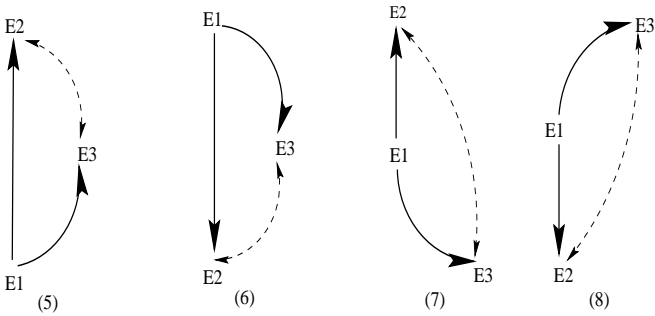


Fig. 6 Exploring the opportunities for using structure indices (case 5 to 8)

R59. $E1 \subset E2 \implies E1 \subset (E3 \subset E2)$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$ and a structure index between $\tau(E3)$ and $\tau(E2)$ is available

R60. $E1 \supset E2 \implies E1 \supset (E3 \supset E2)$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$ and a structure index between $\tau(E3)$ and $\tau(E2)$ is available

R61. $E1 \subset E2 \implies E1 \supset (E3 \subset E2)$ if $\tau(E1)$ obligatorily contains $\tau(E3)$ and a structure index between $\tau(E3)$ and $\tau(E2)$ is available

R62. $E1 \supset E2 \implies E1 \subset (E3 \supset E2)$ if $\tau(E1)$ is obligatorily contained in $\tau(E3)$ and a structure index between $\tau(E3)$ and $\tau(E2)$ is available

Rule R59 - R62 directly come from equivalence E33, E33, E41, E42, respectively. Rule R59 and R60 (corresponding to case (5) and (6) of Fig. 6) introduce a structure index between $\tau(E2)$ and $\tau(E3)$ and retain the portion (from $\tau(E1)$ to $\tau(E3)$) of the original path (from $\tau(E1)$ to $\tau(E2)$). The achieved improvement is obvious. R61 and R62 (corresponding to case (7) and (8) of Fig. 6) introduce a structure index between $\tau(E2)$ and $\tau(E3)$ and an extra path (from $\tau(E1)$ to $\tau(E3)$). A transformation made by R61 and R62 is beneficial only if the newly introduced path is sufficiently shorter than the original path (or cheaper to examine).

Path reduction. If bringing a structure index into a query is impossible, our goal switches to a less favorable one —

reducing the paths involved in a query. This can be done by the following two rules, which are the homologues of rule R55 and R56 (case (1) and (2) of Fig 5). All other cases illustrated by Fig. 5 and 6 are irrelevant to path reduction.

R63. $E1 \subset E2 \implies E1 \subset E3$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$ and is exclusively contained in $\tau(E2)$, and $free(E2)$ holds

R64. $E1 \supset E2 \implies E1 \supset E3$ if $E3$ is an EL for $\tau(E1)$ and $\tau(E2)$ and obligatorily contains $\tau(E2)$, and $free(E2)$ holds

By now, we complete the exposition of an essential class of semantic transformation rules that achieves the primary optimization goal of our approach — applying structure indices to XML queries.

4.3.3 Rule Group 3: Simplification Rules The third phase is intended to carry out a bottom-up cleaning on the output of semantic optimization. Some rules (i.e., R25 - R46) that have been applied during Phase 1, need to be called upon again because after semantic transformation, new redundancies may be introduced by the interpolation of index operators. The rules that enforce explicit DTD-constraints during Phase 1 are excluded because semantic transformation does not normally introduce new inconsistencies. The rules that reorder operators (R6 - R24) are not needed either, as the semantic phase does not change the main structure of a query. Two new types of rules are added to the third phase to achieve thorough simplification. The first type consists of the following two rules in order to remove the \cap operations which may be reintroduced with the interpolation of structure indices.

R65. $I_{\tau(E1)}(E2) \cap E1 \implies I_{\tau(E1)}(E2)$ if $free(E1)$ holds

R66. $I_{\tau(E1)}(E2) \cap \sigma(E1) \implies \sigma(I_{\tau(E1)}(E2))$ if $free(E1)$ holds

R65 is derived from the subset law E3; R66 is from E48. R66 gives precedence to a structure index.

The second type of new rules added to Phase 3 performs simplification by exploiting specific structural knowledge, e.g., exclusivity, obligation, and entrance location. These rules are listed below.

R67. $E1 \subset (E3 \subset E2) \implies E1 \subset E2$ if $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$, and $free(E3)$ (from E33)

R68. $(E1 \subset E3) \subset E2 \implies E1 \subset E2$ if $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$, and $free(E3)$ (from \supseteq associativity E35 and E33)

R69. $(E1 \subset E3) \subset E2 \implies E1 \subset E3$ if $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$, and $free(E2)$ (from \supseteq associativity E36 and E33)

R70. $E1 \supset (E3 \supset E2) \implies E1 \supset E2$ if $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$, and $free(E3)$ (from E33)

R71. $(E1 \supset E3) \supset E2 \implies E1 \supset E2$ if $\tau(E3)$ is an EL for $\tau(E1)$ and $\tau(E2)$, and $free(E3)$ (from \supseteq associativity E35 and E33)

R72. $(E1 \supset E3) \supset E2 \implies E1 \supset E3$ if $\tau(E2)$ is an EL for $\tau(E1)$ and $\tau(E3)$, and $free(E2)$ (from \supset associativity $\mathcal{E}36$ and $\mathcal{E}33$)

R73. $E1 \supset (E3 \subset E2) \implies E1 \subset E2$ if $\tau(E1)$ is an EL for $\tau(E3)$ and $\tau(E2)$, and $free(E3)$ (from $\mathcal{E}41$)

R74. $E1 \subset (E3 \supset E2) \implies E1 \supset E2$ if $\tau(E1)$ is an EL for $\tau(E3)$ and $\tau(E2)$, and $free(E3)$ (from $\mathcal{E}42$)

In the above table, $\mathcal{R}67$ - $\mathcal{R}72$ are listed and will be applied before $\mathcal{R}73$ and $\mathcal{R}74$ because the latter two need to check an extra condition regarding the structure of XML data and thus are less interesting to our system as we are concerned about the performance of our optimizer.

4.4 Examples

In the following we use the query examples introduced in Section 1 to show how our approach works from a logical perspective. These queries were used as benchmark queries in our experimental study (Section 6). For each query, we list the transformation steps that eventually lead to the optimized format of the query. The transformation steps given here are based on the real transformations performed by our test-bed system. For space consideration, we combine consecutive steps that represent basically the same type of transformations into a single *macro* step. The achieved optimization is easily justified according to the optimization heuristics expounded in the article.

The four indices depicted in Fig. 3, i.e., $I_{Article}(Name)$, $I_{Paragraph}(Article)$, $I_{Paragraph}(ShortPaper)$, and the selection index $I_{\sigma_r}(Surname)$ are explored and applied by the transformations.

Query 1 and Query 2 are optimal per se themselves and our optimizer could not perform any further optimization.

Query 3 to 5 are illustrative since the three basic types of document querying patterns[16], i.e., attribute-, content- and structure-query, are all included.

Query 3. Find the paragraphs of the ‘‘Introduction’’ section of each article that has the words ‘‘Data Warehousing’’ in its title.

$$Paragraph \subset (\sigma_{A=Title,r='Introduction'}(Section) \subset (Article \supset \sigma_{r='Data Warehousing'}(Title)))$$

\implies (by $\mathcal{R}51$, \supset associativity)

$$(Paragraph \subset \sigma_{A=Title,r='Introduction'}(Section)) \subset (Article \supset \sigma_{r='Data Warehousing'}(Title))$$

\implies (by $\mathcal{R}50$: \supset commutativity)

$$(Paragraph \subset (Article \supset \sigma_{r='Data Warehousing'}(Title))) \subset \sigma_{A=Title,r='Introduction'}(Section)$$

\implies (by $\mathcal{R}49$: index introduction)

$$(I_{Paragraph}(Article \supset \sigma_{r='Data Warehousing'}(Title))) \cap Paragraph \subset \sigma_{A=Title,r='Introduction'}(Section)$$

\implies (by $\mathcal{R}65$: \cap deletion)

$$(I_{Paragraph}(Article \supset \sigma_{r='Data Warehousing'}(Title))) \subset \sigma_{A=Title,r='Introduction'}(Section)$$

Query 4. Find all articles in which the surname of an author contains the value ‘‘Aberer’’.

$$Article \supset \sigma_{r='Aberer'}(Surname)$$

\implies (by $\mathcal{R}48$, assuming index $I_{\sigma_r}(Surname)$ available)

$$Article \supset I_{\sigma_{r='Aberer'}}(Surname)$$

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

Query 5. Find all ‘‘Summary’’ paragraphs of all sections (if any).

$$\sigma_{A=Title,r='Summary'}(Paragraph) \subset Section$$

\implies (by $\mathcal{R}57$)

$$\sigma_{A=Title,r='Summary'}(Paragraph) \subset Article$$

\implies (by $\mathcal{R}49$)

$$I_{Paragraph}(Article) \cap \sigma_{A=Title,r='Summary'}(Paragraph)$$

\implies (by $\mathcal{R}66$)

$$\sigma_{A=Title,r='Summary'}(I_{Paragraph}(Article))$$

Query 6. Find all paragraph containing both ‘‘OLAP’’ and ‘‘multidimension’’ from either *Article* or *ShortPaper*.

To make the presentation compact, we use the shorter notation σ_1 for $\sigma_{r='OLAP'}$, σ_2 for $\sigma_{r='Multidimension'}$, ‘P’ for ‘Paragraph’, ‘A’ for ‘Article’, and ‘S’ for ‘ShortPaper’ in the following transformations.

$$((\sigma_1(P) \subset A) \cup (\sigma_1(P) \subset S)) \cap ((\sigma_2(P) \subset A) \cup (\sigma_2(P) \subset S)))$$

$$\xrightarrow{step1} (\cup \text{ pulled up by } \mathcal{R}8)$$

$$((\sigma_1(P) \subset A) \cap ((\sigma_2(P) \subset A) \cup (\sigma_2(P) \subset S))) \cup ((\sigma_1(P) \subset S) \cap ((\sigma_2(P) \subset A) \cup (\sigma_2(P) \subset S))))$$

$$\xrightarrow{step2} (\cup \text{ pulled-up by } \mathcal{R}7)$$

$$(((\sigma_1(P) \subset A) \cap (\sigma_2(P) \subset A)) \cup ((\sigma_1(P) \subset A) \cap (\sigma_2(P) \subset S))) \cup (((\sigma_1(P) \subset S) \cap (\sigma_2(P) \subset A)) \cup ((\sigma_1(P) \subset S) \cap (\sigma_2(P) \subset S))))$$

$$\xrightarrow{step3} (\cap \text{ pushed down by } \mathcal{R}17 \text{ and } \mathcal{R}18)$$

$$(((\sigma_1(P) \cap \sigma_2(P)) \subset A) \subset A) \cup (((\sigma_1(P) \cap \sigma_2(P)) \subset A) \subset S) \cup (((\sigma_1(P) \cap \sigma_2(P)) \subset S) \subset A) \cup (((\sigma_1(P) \cap \sigma_2(P)) \subset S) \subset S)$$

$$\xrightarrow{step4} (\text{redundant } \subset \text{ deleted by } \mathcal{R}25)$$

$$(((\sigma_1(P) \cap \sigma_2(P)) \subset A) \cup (((\sigma_1(P) \cap \sigma_2(P)) \subset A) \subset S)) \cup (((\sigma_1(P) \cap \sigma_2(P)) \subset S) \subset A) \cup ((\sigma_1(P) \cap \sigma_2(P)) \subset S)$$

$$\xrightarrow{step5} (\cap \text{ pushed down and deleted by } \mathcal{R}19, \mathcal{R}20 \text{ and } \mathcal{R}23)$$

$$((\sigma_1(\sigma_2(P)) \subset A) \cup ((\sigma_1(\sigma_2(P)) \subset A) \subset S)) \cup$$

$$\begin{aligned}
&(((\sigma_1(\sigma_2(P)) \subset S) \subset A) \cup (\sigma_1(\sigma_2(P)) \subset S)) \\
&\xrightarrow{\text{step6}} (\cup \text{ simplification by } \mathcal{R}39 \text{ and } \mathcal{R}40) \\
&(\sigma_1(\sigma_2(P)) \subset A) \cup (\sigma_1(\sigma_2(P)) \subset S) \\
&\xrightarrow{\text{step7}} (\text{index introduced by } \mathcal{R}49) \\
&((I_P(A) \cap \sigma_1(\sigma_2(P))) \cup (I_P(S) \cap \sigma_1(\sigma_2(P)))) \\
&\xrightarrow{\text{step8}} (\cap \text{ simplification by } \mathcal{R}66) \\
&\sigma_1(\sigma_2(I_P(A))) \cup \sigma_1(\sigma_2(I_P(S)))
\end{aligned}$$

During the above optimization, introduction of $I_P(A)$ and $I_P(S)$ into the query confirms the application of the structure indices $I_{\text{Paragraph}}(\text{Article})$ and $I_{\text{Paragraph}}(\text{ShortPaper})$, respectively. Step 1 to step 6 perform normalization; step 7 explores structure indices using semantic rule $\mathcal{R}49$; and step 8 invokes the simplification rule $\mathcal{R}66$ that gives the precedence of evaluation to the structure index and gets the reintroduced \cap operators eliminated. The optimization achieved via the performed transformations is significant, which will be further demonstrated by our experimental studies.

5 Algorithms

As mentioned earlier, we organize the optimization of PAT query expressions into three phases. One essential characteristic of our methodology is the *deterministic* nature of the transformations accomplished by our heuristic transformation rules. Our system does not need to maintain multiple alternatives for each input expression, but simply invokes applicable rules in a sequential manner and focuses only on the newly generated candidate by each transformation step.

Identification of interesting structural properties of XML data, i.e., exclusivity, obligation, and entrance location, with regard to an input (sub)expressions is another important implementation issue. All important implementation algorithms are discussed in this section.

5.1 Transformation Algorithms

The following algorithms describe the transformation strategy of our approach:

```

optimize(input) := {
  return type: PAT_expression;
  enable all normalization rules
    (but disable all other rules);
  normalized = transform(input);
  enable all semantic rules
    (but disable all other rules);
  improved = transform(normalized);
  enable all simplification rules
    (but disable all other rules);
  optimized = transform(improved);
  return optimized
}

```

```

transform(expr) := {
  return type: PAT_expression;
  for each argument arg of expr
    arg = transform(arg);
  while there are applicable rules in the
  currently enabled rule set
  {
    let r be the first of such rules;
    generate a new expression new_e
      by applying r to expr;
    result = transform(new_e);
  }
  return result;
}

```

To effectively control the search range for the next applicable rule, we attach a “switch” to each transformation rule. A rule is active when its switch is turned on. An inactive rule is excluded from the current search range. The switch of a rule is manipulated by two operations: *enable* and *disable*. Such a simple switch mechanism is an effective way to put the rule system well under control and to achieve better optimization performance.

The order of the rules in our rule system is essential according to our *deterministic transformation* strategy, especially to those that share the same input pattern and same condition. The order of a rule in the rule system is determined according to relevant heuristics. When there are no more applicable and active rules, the `transform` algorithm terminates normally and returns the result expression reached by the last transformation. As long as there are still applicable and active rules, the second for loop in the algorithm will not exit.

5.2 Identifying Exclusivity and Obligation Properties

In Subsection 3.3, we gave the definitions of the three important notions about the structural properties of XML data. These properties can be obtained from the DTD, which may be extracted from the data source if not given.

Based on its definition, examining the exclusivity property between two given element types, e.g., whether B is exclusively contained in A, is simply to check the paths between B and the DTD’s root: if none is found without the occurrence of A then the property holds. The algorithm is straightforward and is thus omitted.

In order to identify all cases of obligation, a deeper look at the content model of element types is indispensable. Otherwise, we cannot distinguish whether an element requires or just optionally contains a subelement. Thus, as a first step, we eliminate all optional occurrence indicators from the content models of the involved element types. This renders the following definition.

Definition 13 (Reduced version of a DTD) *Let D be a DTD. By taking all content models and removing all subexpressions whose root has an optional occurrence indicator (?) or an*

optional-and-repeatable occurrence indicator (*), we obtain a reduced DTD D' .

The following proposition holds [4]:

Lemma 1 *The obligation properties within a DTD and its reduced version are identical.*

For examining the obligation property between element type A and B (given $A \supset B$), we next flatten the content model of A so that the obligatory occurrence of B in A's content model eventually become obvious, and the result is called A's **extended content model** for B, which is obtained from the following algorithm:

```

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

```

Based on the *extended content model* notion, we produce a **normalized form** for the content model of element type A by performing the following steps, which were specified in detail in [4]:

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

These steps do not alter the content represented by a content tree but leads to a normal form that possesses the following properties: (1) the root is an OR-node; (2) the children of the root are SEQ-nodes; (3) the children of SEQ-nodes are leaves, i.e., element types.

In the context of this research, content model normalization is important for revealing implicit obligation property, which is otherwise easy to be missed [4, 12].

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

Our algorithm for examining the obligation property between element type A and B is summarized as below:

```

obligatorily_contains(A,B) := {
  return type: Boolean;
  step 1: produce a reduced version of the DTD;
  step 2: extend the content model of A for B;
  step 3: generate normalized form for the

```

```

  content model of A;
  step 4: search for occurrence of B in each
    SEQ-node of A's content model;
  return true if found, otherwise false;
}

```

5.3 Identifying Relevant Entrance Locations

Most (e.g., $\mathcal{R}51$ to $\mathcal{R}64$) Phase 2 rules rely on the *entrance location* notion. In the third phase, rules $\mathcal{R}67$ to $\mathcal{R}74$ also depend on entrance locations. Thus how to find all *relevant* entrance locations as candidates for enabling the application of one of these rules becomes a key implementation issue to our approach.

As depicted in Fig. 5, the notion, entrance location, as a prerequisite condition for many important transformation rules, is used in two different manners: either a new *etn*, $E3$, as an *entrance location* for $E1$ and $E2$, being used to replace $E2$ (i.e., case (1) and (2)), or the $E2$, an entrance location by itself, being replaced by the new *etn* $E3$ (i.e., case (3) and (4)). Analogously, case (5) and (6) in Fig. 6 are distinct from case (7) and (8).

For terminological uniformity, we generalize our *entrance location* concept so that, when the phrase is generally referred to, it covers all the situations mentioned above.

Definition 14 (Generalized entrance location) *If $E3$ is entrance location for $E1$ and $E2$, we call $E1$ (or $E2$) an outside entrance location for $E3$ and $E2$ (or $E1$), and conversely $E3$ an inset entrance location. When generally referred to, an entrance location is either an inset entrance location or an outside entrance location.*

Therefore, in the remainder of this article, the general term “entrance location” refers to either an *inset entrance location* as in case (1) and (2) of Fig. 5 and case (5) and (6) of Fig. 6, or an *outside entrance location* as in case (3) and (4) of Fig. 5 and case (7) and (8) of Fig. 6.

The algorithm that identifies inset entrance locations between a pair of element types heavily relies on exploring the element type graph, similar to the examination of the exclusivity property. For the sake of efficiency, we assume that each element type ET maintains all the paths that diverge from the element type down to the leaves of the DTD graph under consideration, and denote it as *down_paths(ET)*. The algorithm that identifies inset entrance locations is described as follows:

```

identify_Els(ET1, ET2) := {
  input: element type name ET1 and ET2;
  return type: set of <element type-name>;
  return identify_Els(ET2, ET1) if ET2 is
    external to ET1;
  for each path maintained in down_paths(ET1){
    mark up those containing vertex ET2;
  }
  let s_path be the shortest path among the

```

```

    marked-up ones with regard to the length
    from vertex ET1 to ET2;
  for each intermediate vertex ET contained in
  s_path {
    if it is also contained in all other paths
    marked up {
      collect it into the entrance location set
      el_set;
    }
  }
  return el_set (which contains all the ELs for
  ET1 and ET2);
}

```

This algorithm identifies all inset entrance locations for element type ET1 and ET2. It is evident from the algorithm that an entrance location EL is *relevant* to type ET1 and ET2 if it falls on the shortest path between the two types in the DTD graph.

Although this algorithm identifies only inset entrance locations, outside entrance locations can be similarly determined using a variant of the algorithm that accepts an extra parameter, an *etn*, that relates to the first parameter (as in case (3) and (4)) or the second parameter (as in case (7) and (8)) via an available structure index. Outside entrance locations are needed only for exploiting structure indices that are beyond the direct path between the two element types related via a containment operation.

5.4 Determining Optimal Entrance Locations

For a given expression $E1 \supseteq E2$, multiple entrance locations may be identified and can imply different transformation rules, e.g., $\mathcal{R}55$, $\mathcal{R}57$, $\mathcal{R}59$, $\mathcal{R}61$, and $\mathcal{R}63$ all can be chosen. Therefore we need a certain criterion to choose the optimal entrance location corresponding to the most profitable transformation rule. During the semantic transformation phase, all the candidate entrance locations that enable the application of one of the semantic rules need to be first identified; we then decide the optimal one from all relevant candidates.

For a relevant entrance location, either inset or outside, to be a qualified candidate entrance location, an additional condition — exclusivity or obligation may be needed (e.g., $\mathcal{R}55$ to $\mathcal{R}58$, and $\mathcal{R}63$ and $\mathcal{R}64$).

In the following we give an algorithm that explores all candidate entrance locations and then selects the optimal one from them. Before we proceed, we need to define the concept of an *optimal* entrance location.

Definition 15 (Optimal entrance location) *If multiple candidate entrance locations exist, each may cause an application of one of the rules, $\mathcal{R}55$ through $\mathcal{R}64$, to an expression of form $E1 \supseteq E2$, then the optimal entrance location is determined according to the following criteria:*

(1). *When a structure index can be introduced and the input query expression is not to be expanded (this corresponds*

to $\mathcal{R}55$ through $\mathcal{R}58$), the optimal entrance location is the one having the smallest cardinality of extent.

(2). *When a structure index can be introduced and the input query expression gets expanded (this corresponds to $\mathcal{R}59$ through $\mathcal{R}62$), the optimal entrance location is the one having the smallest cardinality of extent.*

(3). *When the above are not possible (this corresponds to $\mathcal{R}63$ and $\mathcal{R}64$), the optimal is the one that results in the shortest navigation path from E1 to this entrance location (element type) in the DTD graph.*

The above definition with *identify_ELS(ET1,ET2)* together translates to the following algorithm:

```

identify_optimal_EL(E1,E2) := {
  return type: element type name;
  step 1: find relevant inset entrance
  locations;
  step 2: collect candidate entrance locations;
  step 3: select the optimal entrance location
  and return it;
}

```

The details of these steps are addressed below:

Step 1: Find relevant entrance locations

Relevant inset entrance locations are found out by calling *identify_ELS($\tau(E1)$, $\tau(E2)$)*.

Step 2: Collect candidate entrance locations

Assume, beside the diverging paths, each element type t maintains two transitive closure sets, say, *excl_C*(t)* and *obli_C*(t)*, which are the transitive closure of the relationships *exclusively-contained-in* and *obligatorily-contains*, respectively, on the type t . The closures can be computed by the algorithms given earlier in this section. In addition, each element type t maintains a set of *interesting etn's*, annotated as *indices(t)*, which are related to the element type t through an available structure index.

To collect all candidate entrance locations, the following two cases need to be treated separately:

(Case 1: for $\mathcal{R}55$, $\mathcal{R}56$, $\mathcal{R}59$, $\mathcal{R}60$, $\mathcal{R}63$, $\mathcal{R}64$) check each inset entrance location for $E1$ and $E2$ obtained from last step to see if it, say $E3$, is contained in *excl_C*(E2)* or in *obli_C*(E2)*. Next, check whether $E3$ is a member of *indices(E1)*. If yes, mark it as “structure index defined”. For instance, the output of this step for Rule $\mathcal{R}55$ is $rELs(E1,E2) \cap excl_C^*(E2) \cap indices(E1)$, for Rule $\mathcal{R}56$ is $rELs(E1,E2) \cap obli_C^*(E2) \cap indices(E1)$ (for $\mathcal{R}59$ and $\mathcal{R}60$ we do not bother to check the exclusivity or obligation condition), for Rule $\mathcal{R}63$ is $rELs(E1,E2) \cap excl_C^*(E2)$, and for Rule $\mathcal{R}64$ is $rELs(E1,E3) \cap obli_C^*(E2)$.

(Case 2: for $\mathcal{R}57$, $\mathcal{R}58$, $\mathcal{R}61$, and $\mathcal{R}62$) if *indices(E1)* is not empty, for each $E3 \in indices(E1)$ check whether $E3$ is an outside entrance location for $E1$ and $E2$ (in other words, $E2$ is an inset entrance location for $E1$ and $E3$), and whether $E2$ belongs to *excl_C*(E3)* or $E2$ belongs

to $obli_{C^*}(E3)$ (for $\mathcal{R}61$ and $\mathcal{R}62$ we do not bother to check the exclusivity or obligation condition). Mark the element types that satisfy these conditions as “structure index defined”, then output them.

Step 3: Select the optimal entrance location

Again, this step needs to distinguish three general cases according to Definition 15.

- (Case 1) For candidate entrance locations marked as having “structure index defined”, select the one that holds the smallest cardinality and does not expand the input expression (corresponding to $\mathcal{R}55$ to $\mathcal{R}58$).
- (Case 2) For candidate entrance locations marked as having “structure index defined”, select the one that holds the smallest cardinality (corresponding to $\mathcal{R}59$ to $\mathcal{R}62$).
- (Case 3) Otherwise, select the one that minimizes the path between it and the $\tau(E1)$ (for $\mathcal{R}63$ and $\mathcal{R}64$).

6 Experiments

We have made a thorough, analytical presentation of our deterministic approach to XML query optimization based on algebraic transformations. For evaluation purposes, we implemented a test-bed and conducted a series of experiments. We particularly tested the effectiveness, efficiency, and scalability of the presented approach. In this section, we first introduce the test-bed, and then present the detailed results of the experiments.

6.1 Test-bed

We begin with a discussion of a few important choices we made with regard to the implementation of our test-bed.

Due to the inherent, high complexity of XML data models and the relatively limited modeling facilities of relational databases, RDBMSs do not seem to be the ideal choice for hosting XML data. Regardless of this, however, many (if not the majority of) researchers have used an RDBMS as a platform to investigate the techniques of XML data management [18, 21, 5, 38]. The impetus behind this “relational” trend is perhaps the strong desire for a smooth integration of XML data and the legacy relational data sources in a single system. Along the same line, we developed our test-bed based on the Oracle RDBMS. However, the techniques we developed are generic — as our approach is a purely logical-level approach.

Our test-bed consists of the following major functional modules: translation of XPath queries to PAT expressions, PAT query optimization, and translation of (optimized) PAT expressions to SQL queries, which are then sent to the host system for execution.

Storage model. As we adopted the Oracle RDBMS to achieve fast storage management for XML data, a related issue that we shall not circumvent is the mapping issue of XML data to relations. The mapping scheme we chose is a straightforward one, called the “node table” approach. Basically, it

maps each node in a DTD graph to a table. A table consists of the following columns: doc-id, element-id, parent-it, element-value, and a value column for each attribute if the element type has attribute(s). Element-id encodes the regions of elements within a document to facilitate the implementation of the structural join algorithm in our system. In this simple mapping scheme, values are “inlined”, which is proven to be an efficient choice [21]. Our system catalog, called the “mapping table”, keeps an entry for each element type (i.e., a node in the DTD graph) that maintains important mapping information that is needed later on for the translation of (optimized) PAT queries to SQL queries.

Indices. In this test-bed, we support three types of indices: element content indices, attribute content indices, and structure indices (Section 4.1). The implementation of content indices is straightforward: the indexing mechanism of the host DBMS is directly used. Structure indices in our test-bed are simulated as binary relations. For a containment relationship, if no structure index is applicable, our system calls for a structural join algorithm to evaluate the containment relationship. The structural join algorithm implemented is a variant of the MPMGJN algorithm [46]. It is implemented outside the host relational query engine as a post-SQL processing step, which naturally causes some loss of performance of query processing but is the only choice in the current setting as the adopted relational engine is a closed one.

PAT-to-SQL translation. According to our XML-to-relation mapping scheme and the indexing structures we adopted, query translation from optimized PAT expression to SQL is fairly easy. Let us consider several major PAT operators’ translation below. For the two selection operations, σ_r and $\sigma_{A,r}$, our PAT-to-SQL translation module first looks up the system catalog for the table names of relevant element types and the related columns of the tables. Each of the selection operations is then translated into a search condition imposed on a corresponding column of the table. For \supset and \subset , we first check if a corresponding structure index is available. If yes, it substitutes for the containment operation; otherwise, a structural join operation is interpolated. For \cup and $-$, the query is translated to the union or difference of two sub queries (notice that all \cap operation had been eliminated).

6.2 Experimental Environment and Benchmark Databases

We now introduce our experimental environment and the characteristics of our benchmark database. Our experiments were conducted 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.

As expounded, the main goal of our approach is to attain logical optimization of XML queries — primarily for bringing in a structure index into a query through extensive

exploitation of the structure knowledge of XML data. This approach is ideally applied to XML datasets with relatively complex structures and for relatively complex queries. While the complexity of XML data (and the accompanying DTD/XSD) may be characterized by the maximum path length, fan-out, etc., our work mainly refers to the first criterion — path length. For the empirical study, we examined several well-known benchmarks, including XMark [39], XMach [30], XOO7[36], XBench[45], and Michigan benchmark [34]. These benchmarks are found to have either simple DTDs/XSDs or simple queries (e.g., Michigan benchmark), or focus on a performance metric (e.g., throughput in XMach) that is irrelevant to the main theme of our effort. Therefore, in our first round of experiments, we adopted our own synthetic databases.

Our first database, referred to as **Database 1**, is based on the proceedings DTD presented in Section 2. Our experiment with Database 1 was prompted by the desire of knowing how the six running examples (queries) would actually perform.

For experimentation with queries involving longer paths, we implemented our second database (referred to as **Database 2**) based on a more complex DTD that was intentionally designed to have longer paths (see Appendix A). As expected, Database 2 offers a better opportunity of exploiting the potential of our approach for XML query optimization.

Most recently, we conducted experiments with yet a third benchmark database, the XMark benchmark database (referred to as **Database 3**). XMark is chosen from the abovementioned benchmarks because it has a relatively more complex DTD than all the others. Considering space limitation, we provide only sketched experimental results with Database 2 and Database 3 in appendices.

Table 1 lists some important characteristics of Database 1. In the table, we only list the statistics of the database at the two extreme scales (i.e., scale 1 and scale 10). The statistics at the other scales can be easily inferred.

6.3 Performance Results

We now present the performance result with Database 1. The benchmark queries are the six running examples (see Section 4.4). Table 2 shows the selectivity and the optimization time of each benchmark query that we tested with Database 1. Both the selectivity and the optimization time of the queries remain unchanged when database scaling takes effect as our approach is a logical one and the larger scales are generated simply as multiples of smaller scales (the test with varying selectivity is done separately and the result is presented in Section 6.5). Fig. 7 depicts the performance difference between the original queries and their optimized counterparts (optimization time is exclusive and the same for all similar tests in this article). The execution times were obtained when the database was at scale 10 (i.e., with 1000 documents of a total size of 264 MB). Each query was run 10 times and the average elapsed time was measured.

Query 1 and Query 2 by themselves are optimal according to our optimization approach. All other queries achieved

Description	Q1	Q2	Q3	Q4	Q5	Q6
Selectivity(%)	1.84	1.10	0.35	2.24	1.79	1.01
Optimize(sec.)	0.207	0.159	0.862	0.743	0.841	2.252

Table 2 Properties of Database 1 benchmark queries

optimization exactly like we expected, and get executed much faster.

6.4 Scaling Effect

It is interesting to know how our implemented approach responds to database scaling. We did this test using the same set of queries. For this test, Database 1 was populated at 10 different scales: with 100, 200, 300, . . . , 1000 documents, respectively.

The database scaling effect on optimized queries is shown in Fig. 8. As can be seen, our system shows basically a linear scaling behavior. We also tested the scaling effect on the original, non-optimized queries, and got a similar result — linear scaling effect (the details are omitted).

6.5 Effect of Selectivity

How the selectivity of a query affects its execution performance in the framework of our approach is another interesting question. To answer this question, we conducted experiments in the context of Database 1 when it was populated with 1000 documents. Fig. 9 shows the effect of selectivity on the performance of optimized queries. Our experiment shows similar effect of selectivity on the performance of non-optimized queries (the details are omitted). This experiment indicates that the effect of selectivity on query performance is quasi-exponential (or somewhere in between exponential and linear).

6.6 Remarks on Experimental Results

In addition to the test results presented above, we also observe that not all the transformation rules presented in Section 4.3 are equally useful. Some (e.g., $\mathcal{R}49$ and $\mathcal{R}65$) had been very frequently and successfully applied; some (e.g., $\mathcal{R}37$) were just occasionally used, and yet quite a number of others (e.g., $\mathcal{R}21$ and $\mathcal{R}22$) were rarely used. It is interesting to notice that the index application rule $\mathcal{R}49$ and the intersection elimination rule $\mathcal{R}65$ are among the most frequently used ones. Both $\mathcal{R}21$ and $\mathcal{R}22$ handle an empty operand. It is uncommon to have an empty operand involved in a query if the query itself is a meaningful one, which counts for why $\mathcal{R}21$ and $\mathcal{R}22$ are rarely used. However, $\mathcal{R}21$, $\mathcal{R}22$, and the like are still useful rules for handling the uncommon situations that they are intended for.

There are queries that may not be optimized by our approach, for example, Query 1 and Query 2. It is natural for a theoretically already optimal query not to be optimized.

Scale	Size(MB)	#Docs	#Elements	Max degree	Min degree	Depth	Max cardinality	Min cardinality
1	29.43	100	1,158,608	20	1	6	420892	100
10	264.07	1000	10,385,578	20	1	6	3738961	1000

Table 1 Characteristics of Database 1

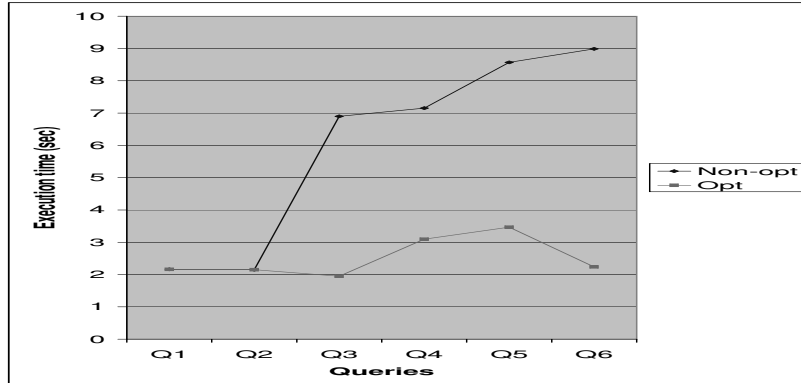


Fig. 7 Performance result with Database 1

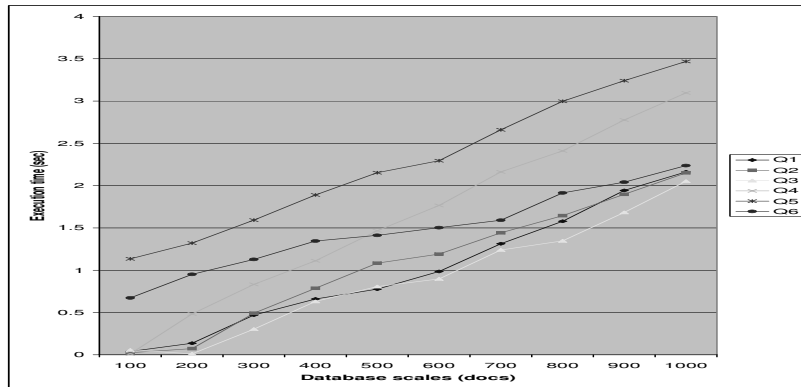


Fig. 8 Scaling of optimized queries with Database 1

In our current prototype implementation, structure indices are simulated by binary relations and in order to apply a structure index to a query, our system relies on introducing two extra relational joins. This is obviously not a very efficient solution. So we are confident with the potential of our optimization approach. Better optimization effect can be achieved if more efficient structures like hashing or B+ trees are directly facilitated for the implementation of our structure indices.

We also notice that when data distribution is highly skewed in the way that only a small portion of the elements of an element type relate to the elements of another element type via a structural relationship, the performance gain we could expect from introducing a structure index into a query is minimized. However, we want to point out that, in principle, using a structure index is always beneficial although relatively uninteresting structure indices are possible. In case an efficient implementation for structure indices is not facilitated (because of, e.g., the *closed-ness* of an adopted host DBMS), uninteresting structure indices can be avoided by carefully checking the database statistics at the time when we reconsider the structural relationships that need index support.

Theoretically, we expect a highly efficient XML query optimizer implemented based on our approach because the approach pursues exclusively deterministic transformations on XML queries, which does not incur any expensive combinatory search (as in a typical rule-based system) for query optimization. Our experimentation indeed assures us of this theoretical expectation: most of the tested queries were optimized in less than 1 second (see Table 2) and even the fairly complicated one, Query 6, took just about 2 seconds for optimization (This result is pretty good especially when considering the hardware platform of our test-bed is not advanced at all.) By and large, we are satisfied with both the effectiveness and efficiency of the test-bed optimizer.

Our approach, as presented, is primarily targeted at two types of optimization: (1) syntactic and semantic simplification on queries (by exploiting structural knowledge of XML data); (2) identifying both obvious and hidden opportunities for introducing a structure index into a query. This approach is most useful when the data source to be handled contains complex (typically, deep and nested) structures.

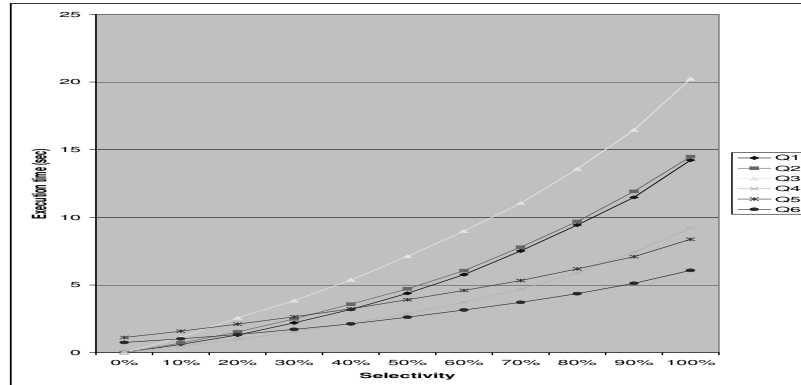


Fig. 9 Effect of selectivity on optimized queries

7 Related Work

In this section, we briefly review some related work and compare our approach with those as reported in the literature.

Lore [32, 33] is a DBMS originally designed for semistructured data and later migrated to XML-based data model. Lore has a fully-implemented cost-based query optimizer that transforms a query into a logical plan, and then explores the (exponential) space of possible physical plans looking for the one with the least estimated cost. Lore is well known by its DataGuide path index that together with stored statistics provides the structure knowledge about the data to help Lore’s optimizer prune its search space for a better plan. The Lore optimizer is cost-based and does not perform logical-level optimization. As mentioned earlier, XML queries carry rich semantics regarding the structure of XML data and, as a source of optimization heuristics, the semantics is best exploited at a more expressive logical representation level. That’s where our approach is distinguished.

The work reported in [15] is one piece of the early work reported on using the DTD knowledge on query transformations. The DTDs considered there are much simpler than the ones of XML. The optimization [15] makes use of a special cost model. In contrast, our approach is independent of specific storage and cost models.

In [17], a comparable strategy for exploiting a grammar specification for optimizing queries on semistructured data is studied. In this study, efforts were made on how to make complete use of the available grammar to *expand* a given query. Our focus is different — we identify transformations that introduce “guaranteed” improvements on query expressions from a heuristic point of view.

In [8], a fresh idea about using the DTD of XML documents to guide the query processor’s behavior was proposed. It is common to model one (or more) element type(s) as a class (in OO modeling) or a table (in relational modeling). In [8], one type of elements (corresponding to one node in the DTD graph) is classified according to the diverging paths of the node. Later, query processing is guided to a more specific class of this type to prune the search space. For example, suppose the content model (with sub-content models being inlined) of element type A is $B|C, D^*$, then elements of

type A can be classified into at least three classes: elements containing only B elements, elements containing only C elements, and those containing a C element followed by one or more D elements. Indeed, this classification information can effectively guide the query engine to narrow its search space and speed up query evaluation; but with a relatively complex DTD, too many classes may be produced. For example, if a node in the DTD graph has a diverging long path like $A/B^*/C^*/D^* \dots$, this path alone would cause a large number of classes being defined. One obvious drawback of this approach is that too many classes need be handled even for a very simple query. Thus the efficiency gained from classification can be substantially compromised. In contrast, our approach exploits the deep level knowledge inferred from the DTDs, i.e., semantics regarding the structures of XML documents to transform queries toward better evaluation plans.

In [42], the notion of a path is a rooted one, i.e., starting from the document root. Two types of optimization were introduced based on this path notion. (1) path complementing: if the path in a “path query” has a complement (i.e., the alternative paths that reach the same end node) and its evaluation is cheaper (based on cost estimation), then the complement substitutes for the original path; (2) path shortening: if the head segment of a path is the unique one that reaches the ending point (node) of the segment, then this segment is removed. Two key concepts were introduced to support above optimization: parent-child extent, which is a set of parent-child pairs at the instance level linked through the same tag, and ancestor-descendent extent, which is a set of ancestor-descendent pairs also at the instance level. If a query comprises of multiple paths, of which each (being a single path sub-query) evaluates to an extent (of either type of the above) and is to be linearly joined with another — this is the so-called *extent join*. The parent-child extent notion is nothing new but a binary access supporting relation or the counterpart of the edge table as proposed in [21]. The ancestor-descendent extent coincides with our notion of “structure index” which was proposed early in [11]. The path shortening idea is similar to our idea of *navigation reducing*, which is discussed in this and other previous papers [11, 12]. Actually, our algorithm [12] is more general and can identify more opportunities for reducing a path as our paths do not need to

be rooted. Path complementing is a new dimension of applying structure knowledge on query optimization but is of little use analytically because the number of joins (which dominate the total cost of queries) are most unlikely to be reduced. In fact, it could be much increased if the complement of a path consists of more than one path. Our approach explores many other aspects of applying structure knowledge of XML data and related heuristics for query optimization as elaborated in this article.

In [7], path constraints were used to convey the semantics of semistructured data and applied to query optimization. It is worth mentioning that in the context of XML data, there are two types of semantics: *data semantics* (i.e., the meanings of data) and *structure semantics* (i.e., the knowledge about the structural properties of the data). The path constraints investigated in [7] is limited to only data semantics. For example, the following constraint rule (borrowed from [7]): $\forall x(emp \cdot _ \cdot manager(r,x) \rightarrow emp \cdot _ \cdot (r,x))$ (The rule can be interpreted as “the manager of any employee is also an employee”). This constraint conveys the meaning of the data rather than the general structure of the data, e.g., *containment* and *referencing*, which are common in XML data. Query optimization applying data semantics is interesting to semistructured data, but is of little use to XML data because within XML data there are no particular data semantics but only general structure semantics, i.e., *containment* and *reference*. However, certain implicit knowledge (or properties) about the structural connections within XML data —like exclusivity, obligation, and entrance location — is of great value for XML query optimization. Applying this type of knowledge to query optimization is one of the focused points of this article, which also makes our work distinct from [7].

Path and tree pattern matching build the core of XML query processing. A bundle of approaches [6,40,46,31,25] 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 lists of nodes. These approaches typically utilize an inverted file structure (as the basis of indexing schemes) to provide lists of nodes related to a certain term or node label. Outstanding of the bundle, the node identification and indexing scheme proposed in [6] facilitate tree or path queries with much improved space efficiency. In [25], path queries are mapped to region queries in the pre/post plane — which is 2-D space determined by the pre- and post-order numbers of the elements within a document; thus a good region indexing scheme, like R-tree, can be ideally used as a support indexing structure. Our work complements all the approaches of *structural joins* in two different ways: (1) a long path involved in a query may be significantly shortened by our transformation-based optimization that exploits particular structural knowledge; (2) our “shortcut” style of structure indices (once built) can serve as a more efficient alternative to structural joins.

In [20], the minimization issue of XPath queries is discussed, and efforts were mainly given to the re-formulation

of queries for avoiding redundant conditions, and a polynomial algorithm was developed for minimizing query patterns involving $/$, $//$, $[]$, and $*$. The mechanism used was to find a homomorphism from query q to p to identify the containment between the two queries (i.e., $p \subseteq q$), which is strategically different from our approach.

In [1], the authors studied the minimization issue of general tree pattern queries, and both constraint dependent and constraint independent minimizations were addressed. They considered two types of constraints derived from a DTD/XSD, namely, required child/descendant (equivalent to what we called obligation, see Section 3.3) and co-occurrence of sibling subelements. The latter is not addressed in our work. However, we exploit two other types of structure knowledge, i.e., exclusivity and entrance location (see Section 3.3), which were not discussed in [1].

Schema-based XPath query optimization was also addressed in [29], where several ideas proposed were similar to ours but a different approach was taken. In [29], the path equivalence class concept was put forward and used to (1) eliminate redundant path conditions; (2) shorten necessary paths; (3) identify unsatisfiability of a query at compile-time. In [29], the optimality of an XPath query was with regard to non-redundant and shortest paths, we have to indicate that using a shorter path is good only when traversal is the only means for evaluating the path. In contrast, we put a lot of efforts on identifying potential structure indices and improving the quality of a query by extensively exploiting structural properties of XML data.

Finally, with regard to the algebras for XML, TAX [27] and XAL [22] are worth mentioning. TAX makes almost a direct map between the rich language facilities of XQuery and its algebraic operators. TAX takes trees as basic manipulation units and is at a very high level of representation. In fact, we may call TAX merely as a “representation algebra” for XML queries. In contrast, XAL is an algebra for XML query optimization because optimization equivalences can be conveniently expressed using the algebra. XAL, like our PAT, is at a relatively low level of representation. XAL is more expressive than PAT but is not just right for us to carry out our particular optimization ideas. Our current version of extended PAT is good for XPath queries, but is not powerful enough for general XQuery queries. We are in a process of extending our PAT, like adding new operators for more general joins (e.g., twig join and value-based joins) and XML construction.

To the best of our knowledge, our work is the only one that uses algebraic transformations so intensively and extensively on achieving XML query optimization.

8 Conclusion

In this article, we elaborated on an innovative optimization approach for an important subclass of XML queries, mostly XPath related queries. The proposed approach is based on heuristic, logical-level, algebraic transformations on XML queries represented as PAT expressions. Every transformation per-

formed is *deterministic* in the sense that it achieves an important improvement on its input query from a heuristic point of view. All transformations are accomplished by invoking a corresponding transformation rule, and the process is a recursive one. All rules have a solid basis — algebraic equivalences. In the context of our approach, optimization consists of three consecutive transformations phases: normalization, semantic optimization, and simplification. Experimental results demonstrate good optimization efficiency and good scalability of our approach. Experimental study reveals that after optimization most queries got executed more than 3 times faster, and the approach showed basically linear scalability in terms of query performance and database size. Analysis indicates that all optimized queries are indeed optimal from an analytical or heuristic point of view.

At present, we are in a process of extending the PAT algebra to make it more expressive and more suitable for XML query optimization and especially for embodying our particular optimization methodology. Meanwhile, we are enriching the implemented test-bed system to make it one (or closer to a) full-fledged XML DBMS equipped with advanced query optimization mechanisms.

acknowledgements We would like first to thank Dr. Klemens Böhm, our former colleague at GMD-IPSI, for his earlier inspiring work in a related project. We acknowledge the following graduate students at Southern Illinois University for their valued contribution to the implementation of the test-bed system: Santosh Ramakrishna, Yung-Chuan Lee, Aleksandr Kulygin, and Yongyan Huang.

References

1. S Amer-Yahia, S Cho, L Lakshmanan, D Srivastava (2001) Minimization of tree pattern queries. In: Proceedings of ACM Conf. on Management of Data (SIGMOD), 2001, pp:497-508.
2. S Boag, D Chamberlin, MF Fernandez, D Florescu, J Robie, J Simeon (2003) XQuery 1.0: An XML Query Language, 2003 (<http://www.w3.org/TR/xquery/>).
3. K Böhm, K Aberer, EJ Neuhold, X Yang (1997) Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. The VLDB Journal 6(4):296 - 311 (November 1997).
4. K Böhm, K Aberer, M T Özsu, K Gayer (1998) Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition. In: Proceedings of IEEE International Forum on Research and Technology Advances in Digital Libraries, April 22-24, 1998, Santa Barbara, California, pp:196-205.
5. P Bohannon, J Freire, P Roy, J Simon (2002) From XML Schema To Relations: A Cost-Based Approach to XML Storage. In: Proceedings of the 18th International Conference on Data Engineering (ICDE'02), 2002, pp:64-75.
6. JM Bremer, M Gertz (2003) An Efficient XML Node Identification and Indexing Scheme. University of California at Davis, Technical Report (http://www.db.cs.ucdavis.edu/papers/TR_CSE-2003-04_BremerGertz.pdf).
7. P Buneman, W Fan, S Weinstein (1999) Query Optimization for Semistructured Data Using Path Constraints in a Deterministic Data Model. In: Proceedings Of DBPL, pp:208-223.
8. T-S Chung, H-J Kim (2002) XML query processing using document type definitions. Journal of Systems and Software 64(3): 195-205.
9. C-Y Chan, P Felber, M Garofalakis, and R Rastogi (2002) Efficient Filtering of XML Documents with XPath Expressions. In: Proceedings of International Conference on Data Engineering, San Jose, California, February 2002, pp:235-244.
10. C-Y Chan, MN Garofalakis, R Rastogi (2002) RE-Tree: An Efficient Index Structure for Regular Expressions. The VLDB Journal 12(2):102-119.
11. D Che and K Aberer (1999) A Heuristics-Based Approach to Query Optimization in Structured Document Databases. In: Proceedings Of International Database Engineering and Application Symposium, Montreal, Canada, August 2-4, 1999 pp:24-33.
12. D Che (2003) Implementation Issues of a Deterministic Transformation System for Structured Document Query Optimization. In: Proceedings Of the Seventh International Database Engineering and Applications Symposium, July 16-18, 2003, Hong Kong, China, pp:268-277.
13. S-Y Chien, Z Vagena, D Zhang, V J Tsotras, C Zaniolo (2002) Efficient Structural Joins on Indexed XML Documents. In: Proceedings of 28th International Conference on VLDB, Hong Kong, China, 2002, pp: 263-274.
14. J Clark, S DeRose (1999) XML Path Language (XPath) Version 1.0. (<http://www.w3.org/TR/1999/REC-xpath-19991116>).
15. M Consens, T Milo (1994) Optimizing Queries on Files. In: Proceedings of ACM SIGMOD International Conference on Management of Data, May 1994 pp:301-312.
16. T Dao (1998) An Indexing Model for Structured Documents to support Queries on Content, Structure and Attributes. In: Proceedings of IEEE International Forum on Research and Technology Advances in Digital Libraries, Santa Barbara, California, April 22-24, 1998 pp:88-97.
17. M F Fernandez, D Suci (1998) Optimizing Regular Path Expressions Using Graph Schemas. In: Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA, pp:14-23.
18. M Fernandez, W Tan, D Suci (2000) SilkRoute: Trading between Relations and XML. In: Proceedings of the 9th Int. World Wide Web Conference, Amsterdam, May, 2000.
19. T Fiebig, S Helmer, C-C Kanne, G Moerkotte, J Neumann, R Schiele, T Westmann (2002) Anatomy of a native XML base management system. The VLDB Journal 11(4):292-314.
20. S Flesca, F Furfaro, E Masciari (2003) On the minimization of Xpath queries. In: Proceedings of VLDB, pp:153-164, 2003, pp153-164.
21. D Florescu, D Kossmann (1999) Storing and Querying XML Data Using an RDMBS. IEEE Engineering Bulletin 22(3):27-34.
22. F Frasincar, G-J Houben, C Pau (2002) XAL: an Algebra for XML Query Optimization. In: Proceedings of 13th Australasian Database Conference, Melbourne, Australia.
23. L J Gerstein (1987) Discrete Mathematics and Algebraic Structures. W H Freeman and Company, New York, 1987.
24. G Gottlob, C Koch, R Pichler (2002) Efficient Algorithms for Processing XPath Queries. In: Proceedings of VLDB, Hongkong, China.
25. T Grust (2002) Accelerating XPath location steps. In: Proceedings ACM SIGMOD International Conference on Management of Data, pp:109-120.
26. S Guha, H V Jagadish, N Koudas, D Srivastava, T Yu (2002) Approximate XML joins. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp:287-298.

27. H V Jagadish, L V S Lakshmanan, D Srivastava, K Thompson (2001) TAX: A Tree Algebra for XML. In: Proceedings of DBPL Conference, Rome, Italy, 2001, pp:149-164.
28. M Klettke, H Meyer (2000) XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In: Proceedings of International Workshop on the Web and Databases (WebDB), Dallas, May, 2000.
29. A Kwong, M Gertz (2001) Schema-based optimization of XPath expressions. Technical report, Dept. of Computer Science, University Of California, 2001.
30. XMach-1: Benchmarking XML Data Management Systems (<http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html>).
31. Q Li, B Moon (2001) Indexing and Querying XML Data for Regular Path Expressions. In: Proceedings Of the 27th International Conference on Very Large Databases, Rome, Italy, September 2000, pp:361-370.
32. J McHugh, S Abiteboul, R Goldman, D Quass, J Widom (1997) Lore: A Database Management System for Semistructured Data. SIGMOD Record, September 1997, 26(3):54-66.
33. J McHugh, J Widom (1999) Query Optimization for XML. In: Proceedings of the 25th International Conference on Very Large Databases, Edinburgh, Scotland, September 1999, pp:315-326.
34. The Michigan Benchmark (<http://www.eecs.umich.edu/db/mbench/description.html>).
35. T Milo, D Suci (1999) Index Structures for Path Expressions. In: Proceedings of ICDT, pp:277-295.
36. The XOO7 Benchmark (<http://www.comp.nus.edu.sg/ebh/XOO7.html>).
37. A Salminen, F W Tompa (1994) PAT Expressions: an Algebra for Text Search. Acta Linguistica Hungarica 41(1):277-306.
38. J Shanmugasundaram, K Tufte, G He, C Zhang, D DeWitt, J Naughton (1999) Relational Databases for Querying XML Documents: Limitations and Opportunities. In: Proceedings of VLDB, pp:302-314.
39. A R Schmidt, F Waas, M L Kersten, M J Carey, I Manolescu, R Busse (2002) XMark: A Benchmark for XML Data Management. In: Proceedings of the International Conference on Very Large Data Bases, Hong Kong, China, August 2002, pp:974-985.
40. D Srivastava, S Al-Khalifa, H V Jagadish, N Koudas, J M Patel, Y Wu (2002) Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In: Proceedings of ICDE, 2002.
41. B Surjanto, N Ritter, H Loeser (2000) XML Content Management based on Object-Relational Database Technology. In: Proceedings of the 1st International Conference On Web Information Systems Engineering (WISE), Hongkong, June 2000.
42. G Wang, M Liu (2003) Query Processing and Optimization for Regular Path Expressions. In: Proceedings of CAiSE, 2003, pp:30-45.
43. Standard Generalized Markup Language (<http://xml.coverpages.org/sgml.html>).
44. Extensible Markup Language (<http://xml.coverpages.org/xml.html>).
45. B B Yao, M T Özsu, N Khandelwal (2004) XBench Benchmark and Performance Testing of XML DBMSs. In: Proceedings of 20th International Conference on Data Engineering, Boston, MA, March 2004, pp:621-632.
46. C Zhang, J F Naughton, D J DeWitt, Q Luo, G M Lohman (2001) On Supporting Containment Queries in Relational Database Management Systems. ACM SIGMOD Record 30(2):425-436.

47. N Zhang, V Kacholia, M T Vzs (2004) A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In: Proceedings of 20th International Conference on Data Engineering, Boston, MA, March 2004, pp:56-65.

Appendix A: Experimental result with Database 2

The DTD graph of Database 2 is shown in Fig. 10, which has a larger depth, compared with Database 1. The characteristics of Database 2 are shown in Table 3. We carefully designed 16 queries that cover various lengths of paths. Corresponding to each maximal length, we designed two queries, e.g., the first pair of queries covers a maximal length of 1, and the last pair of queries covers a maximal length of 8, which is the maximal length of paths possible according to the DTD given in Fig. 10. The benchmark queries are provided in Table 4, and the performance result is shown in Fig. 11.

	Queries (in PAT expressions)	Selectivity	Optimization (sec)
Q1	$A \supset \sigma_{r='ALISA'}(C)$	12.50%	0.131
Q2	$K \supset \sigma_{r='BARBIE'}(M)$	6.31%	0.125
Q3	$\sigma_{r='GARFIELD'}(H) \subset D$	15.62%	0.206
Q4	$(D \supset \sigma_{r='GYPSY'}(F)) \cup (D \supset \sigma_{r='GYPSY'}(H))$	17.76%	0.246
Q5	$A \supset \sigma_{A='Title', r='DONALD'}(F)$	12.50%	0.264
Q6	$I \subset \sigma_{A='B', r='COMFORT'}(B)$	6.33%	0.249
Q7	$A \supset \sigma_{A='I', r='GARLAND'}(I)$	12.50%	0.186
Q8	$D \supset \sigma_{A='M', r='CARLTON'}(M)$	8.41%	0.197
Q9	$B \supset \sigma_{A='M', r='GARLAND'}(M)$	35.71%	0.294
Q10	$(B \supset I) \supset \sigma_{r='CARLTON'}(M)$	72.00%	0.462
Q11	$(\sigma_{r='COSMO'}(P) \subset K) \cup (\sigma_{r='COSMO'}(P) \subset D)$	3.09%	0.846
Q12	$A \supset (D \supset \sigma_{r='CARLTON'}(M))$	62.50%	0.594
Q13	$(A \supset I) \supset (\sigma_{r='CARLTON'}(N))$	84.00%	0.246
Q14	$(B \supset \sigma_{r='ABU'}(J)) \cup (B \supset \sigma_{r='ADAM'}(Q))$	96.43%	0.349
Q15	$\sigma_{r='BLAZE'}(P) \subset A$	2.50%	0.543
Q16	$(A \supset (I \supset M)) \supset \sigma_{r='CARLTON'}(P)$	100.00%	0.242

Table 4 Benchmark queries of Database 2

Remarks. We observe some queries' execution, e.g., Q11 (before optimization), took much longer than other queries, e.g., Q1, and some query like Q11 was much better optimized than other queries like Q16. We provide explanation below. With basically a tree shaped structure like the one shown in Fig. 10, an element type at a lower level corresponds to a much bigger cardinality than a higher level one (the increasing is exponential). This explains why the non-optimized counterpart of Q11 took so much more time to evaluate than Q1 (Q11 needs to examine the elements of those types that are at the bottom of the tree while Q1 queries the elements of those types that are on top of the tree). Coming to the optimization part, both parts of Q11 can use a structure index (I_{K-P} and I_{D-P} , respectively), so the achieved op-

Size(MB)	#Docs	#Elements	Max degree	Min degree	Depth	Max cardinality	Min cardinality
102.03	20	4588009	5	1	9	796941	20

Table 3 Characteristics of Database 2

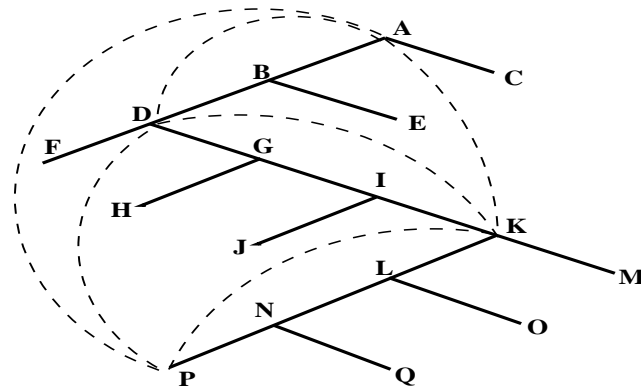


Fig. 10 DTD graph of Database 2 (dashed lines denoting structure indices)

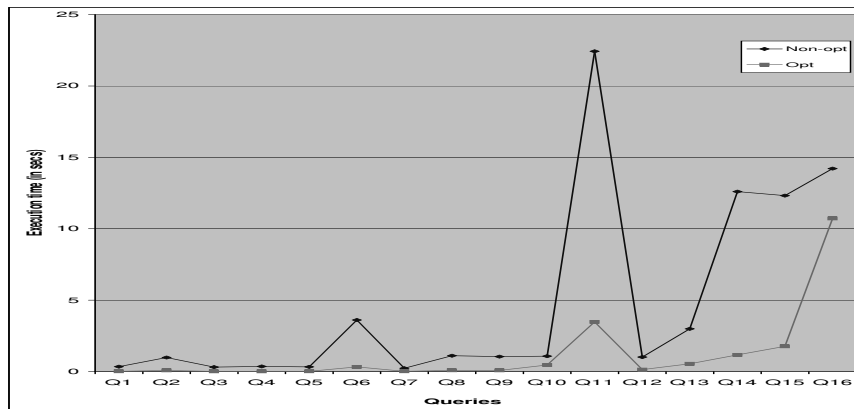


Fig. 11 Performance result with Database 2

timization is significant. With Q16, it goes through A-I-M-P to get the result and can not directly benefit from any available index (but can still benefit from index I_{A-K} and I_{K-P}), so the optimization is not that good when compared to query 11.

Furthermore, we observe that while the max length of paths involved in a query basically increases (from Q1 to Q16), the execution time (of both optimized and non-optimized queries) does not increase accordingly. This can be explained by taking Q11 for example. Q11 is a union of 2 subqueries, one covers a path of length 3 (K-P) and the other covers a path of length 6 (D-P), so the non-optimized execution time is even more than a query consisting of a single path of length 9 (while the max length is 8 according to the DTD of Database 2). On the other hand, queries involving longer paths do not necessarily get better optimized. This is because the applicability of most beneficial transformations on a query is decided based not only on the paths involved but on rather complicated conditions (e.g., obligation, exclusivity, and entrance location). Finally, our deterministic strategy is indeed efficient, with all benchmark queries of Database 2 being optimized within a second (Table 4).

Appendix B: Experimental result with Database 3 (XMark)

Database 3 is based the XMark benchmark at scaling factor 1.0 (100 MB). 12 of the 20 XMark queries were adapted and applied to our test-bed. The selectivity and optimization time of these queries are listed in Table 5, and the performance result is shown in Fig. 12.

Remarks. Queries involving lengthy paths benefit the most from our optimization approach when structure indices are adequately defined. Many of the XMark benchmark queries are very simple (e.g., Query 7 does not involve any containment operation), and they cannot be further optimized at the logical level in any heuristic viewpoint. This explains why the effect of optimization (as shown in Fig. 12) is generally not as good as in Database 1 and Database 2. We pick up a few representative examples to furnish further explanation. Query 5, 7, and 20 are least (actually not) optimized because they contain very simple paths and can not prompt any meaningful optimization at the logical level. Query 6, 15, and 16 are optimized the most because of the involved complex paths and the available structure indices, which together facilitate the expected optimization.

Description	Q1	Q2	Q5	Q6	Q7	Q10	Q13	Q14	Q15	Q16	Q17	Q20
Selectivity(%)	1	100	5	100	100	0	100	16	12	14	4	29
Optimization(sec)	0.086	0.348	0.115	0.284	0.045	0.512	0.267	0.183	0.421	0.843	0.104	0.092

Table 5 Properties of the benchmark queries of Database 3

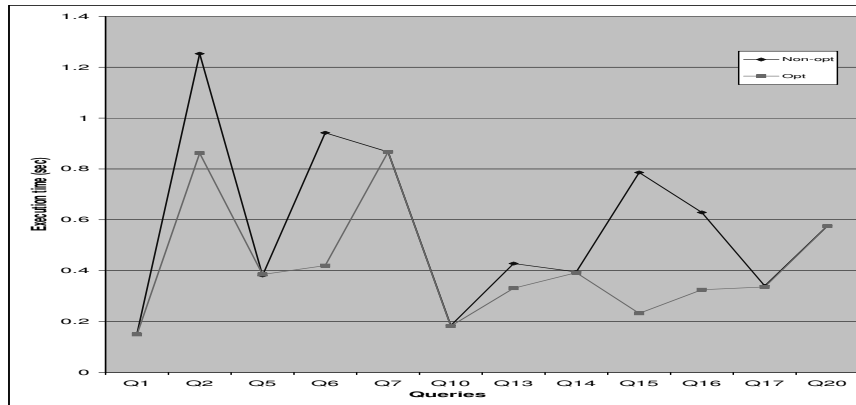


Fig. 12 Performance result with Database 3 (XMark)