

Determined: A System with Novel Techniques for XML Query Optimization and Evaluation

Dunren Che and Wen-Chi Hou
Email: {dche, hou}@cs.siu.edu
Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA

Abstract: Purpose — Efficient processing of XML queries is critical for XML data management and related applications. Previously proposed techniques are unsatisfactory. This article presents *Determined* — a new prototype system designed for XML query processing and optimization from a system perspective. With *Determined*, we propose and demonstrate a number of novel techniques for XML query processing.

Design/methodology/approach — Our methodology emphasizes on query pattern minimization, logic-level optimization, and efficient query execution. Accordingly, three lines of investigation have been pursued in the context of *Determined*: (1) XML tree pattern query minimization; (2) logic-level XML query optimization utilizing deterministic transformation; (3) specialized algorithms for fast XML query execution.

Findings — We developed and demonstrated (1) a runtime optimal and powerful algorithm for XML tree pattern query minimization; (2) a unique logic-level XML query optimization approach that solely pursues deterministic query transformation; (3) a group of specialized algorithms for XML query evaluation.

Research limitation/implications — The experiments we conducted so far are still preliminary. Further in-depth, thorough experiments thus are expected, ideally carried out in the setting of a real-world XML DBMS system.

Practical implications — The techniques/approaches we propose can be adapted to real-world XML database systems to enhance the performance of XML query processing.

Originality/value — The reported work integrates various novel techniques for XML query processing/optimization into a single system, and presents our findings from a system perspective.

Keywords: XML query processing, XML query optimization, XML query evaluation, XML query minimization, Deterministic query optimization, Algorithm.

Article Type: Research paper

1 Introduction

With the ever-increasing popularity of XML, there is a concomitant increase in the need for efficiently managing and querying large volumes of XML data. Relational DBMS has proven successful for traditional data. However, the tree-structured model of XML data makes suitable techniques for XML fundamentally different. The research community tends to use the term “Native XML technology” to emphasize the importance of employing the unique features of XML data in the technology. Among others, the following are the two most prominent features of XML data that played important roles in our investigation:

1. Every XML query specifies one or more tree-shaped search patterns, called *tree patterns* or *pattern trees*, typically associated with additional predicates describing the contents of the covered data nodes.

2. XML data is semantic-rich in the sense that various structural relations among the data elements commonly exist (often implied by the DTD/XSD if available).

Feature 1 indicates that efficient tree pattern matching is at the core of XML query processing. Furthermore, the tree patterns specified by user queries are often un-minimal; therefore, minimization of the tree patterns is utmost important for efficient execution of XML queries. The semantic-rich feature is another valuable asset that should be adequately exploited to obtain XML query optimization. In the context of *Determined*, we investigate various aspects of XML query processing, optimization, and evaluation, and incorporate a range of related techniques into a single (prototype) system — *Determined*. Using *Determined* as a test-bed, we demonstrate the validity of the techniques and approaches we proposed for solving the specific problems of XML query optimization and evaluation.

Compared with related systems, *Determined* distinguishes itself at three main points: (1) it involves a powerful and efficient minimization scheme for the tree patterns in XML queries; (2) it comprises a novel semantic optimization module that carries out solely *deterministic* transformation on XML queries for fast optimization; (3) it has a rich set of primitives (algorithms) supporting efficient XML query evaluation, including specialized joins (algorithms) coping with special cases of *containments* such as *pure* and *negated* containments, and powerful holistic twig join algorithms capable of dealing with *all-twigs* (i.e., twigs with AND, OR, and NOT predicates). In *Determined*, queries in turn go through a series of consecutive processing steps (each is implemented in a corresponding module): parsing, minimization, optimization, and execution. (Logical) query optimization in *Determined* comprises three successive phases: normalization, optimization, and simplification, which are all implemented via a set of heuristic-based, deterministic transformation rules at the logic level. As a fully-functioning prototype, *Determined* also encompasses basic modules for XML data indexing and storage management.

Article organization The remainder of this article is organized as follows: Section 2 addresses our motivation and related work. Section 3 sets forth the preliminaries for the subsequent discussion, including integrity constraints (or ICs) in the context of XML, the ePAT algebra adopted by our optimization approach, and sample query equivalences utilized in *Determined*. Section 4 provides an overview of *Determined*, including its system architecture and interfaces. Section 5 addresses the tree pattern minimization issue of XML queries. Section 6 sketches *Determined*'s deterministic approach to XML query optimization. Section 7 presents the specialized supports in *Determined* for special cases of containments and a holistic twig join solution for all-twigs. Section 8 draws some conclusions from our experience and experiments with *Determined*. Section 9 concludes this article.

2 Motivation and Related Work

Query optimization traditionally adopts the cost-based approach such as (Selinger, Astrahan, Chamberlin, Lorie & Price 1979) and aims at obtaining the least expensive — the optimal — evaluation plan for each input query. Typically, queries are represented as algebraic expressions and equivalent transformation is then applied to the query expressions for identifying the optimal plan. Heuristic knowledge may additionally be exploited to limit the search space so that only a subspace (containing highly potential candidates) needs be explored. The purpose of this is to accelerate the optimization process. The nature of XML data requires us to examine not only the contents (or values) but also the structures of the stored XML data during query evaluation. Compared with relational data, XML data has higher complexity because of the extra structural facet of XML data. The consequence of this high complexity is two-fold: (1) the underlying query algebra requires a relatively large set of operators with high interaction between them (in order to be able to capture the rich semantics of XML data); (2) on the other side, this high complexity implies an important asset of knowledge that may help accelerate the processing/optimization of XML queries. A large set of algebraic operators inevitably leads to an enlarged search space during query optimization, which makes

straightforward application of the classic cost-based approach such as (Selinger et al. 1979) to XML queries less feasible (because of efficiency concern). In other words, innovative optimization approaches are much desired, which motivated our idea of applying deterministic transformation on XML query expressions for optimization. In addition, XML queries naturally carry one or more tree patterns (Amer-Yahia, Cho, Lakshmanan & Srivastava 2001) that are typically not minimal. Therefore, effective minimization of the involved tree patterns before the query is sent for further processing is of great importance for the performance of the subsequent processing of the query. Recognizing these challenges and opportunities, we developed a novel XML query minimization scheme, a deterministic query optimization approach, and a group of specialized support algorithms for fast XML query evaluation, which are all integrated into a single system — *Determined*. This article report our new findings from our experience with *Determined*.

In brief, our methodology emphasizes the exploitation of multiple sources of knowledge and mechanisms for XML query processing/optimization. In our work, query minimization is taken as the first necessary step toward to efficient execution of XML queries. Following that, we adopted a novel heuristic-based approach to logic-level optimization of XML queries. This approach exploits a variety of sources of knowledge (including ICs) and conducts *deterministic* transformation on input queries (represented as algebraic expressions) so that an “optimal” query plan is eventually and quickly obtained. At the last stage, we apply specialized support algorithms developed by our team for efficient execution of XML queries that have already gone through the minimization and optimization stages.

2.1 Related Work

Query processing and optimization for XML has been a very active research area and led to many publications. We limit our review herein only to those that are closely related to our study. Our main purpose in this article is to present *Determined* from a system perspective and discuss the kernel techniques in *Determined*. Accordingly, we will address three major technical aspects of *Determined*: tree pattern minimization, deterministic XML query optimization, and specialized algorithms for XML query evaluation.

Tree Pattern Query Minimization

Amer-Yahia et al (Amer-Yahia et al. 2001) did the pioneer work on TPQ (Tree Pattern Query) minimization. In the absence of any ICs (i.e., Integrity Constraints), their algorithm induces a runtime of $O(n^4)$ with respect to the query size n ; when only the required child/descendant and subtype constraints are considered, their algorithm needs $O(n^6)$ time. Later, Ramanan (Ramanan 2002) proposed a more efficient algorithms. Ramanan’s approach is based on the concept of *graph simulation* tailored to XML. His IC-independent algorithm claimed to be runtime optimal — with $O(n^2)$ time, and his IC-present algorithm *MinimizeChase* takes $O(n^4)$ time. However, all so far proposed algorithms for IC-present TPQ minimization (Amer-Yahia et al. 2001, Ramanan 2002) are incapable of handling more than the three basic types of ICs, i.e., required child, required descendant, and subtype. We developed an alternative runtime-optimal algorithm for IC-independent TPQ minimization (Chen & Che 2006a) and companion algorithm for IC-present minimization of $O(n^3)$ time (Chen & Che 2006b) which however can exploit more types of ICs. Recently, we developed a father more powerful minimization scheme that can exploit a much broader spectrum of ICs, and implemented and integrated the scheme into *Determined*.

XML Query Optimization

Lore (McHugh, Abiteboul, Goldman, Quass & Widom 1997, McHugh & Widom 1999) represents a piece of pioneer work on XML query optimization. The Timber native XML DB (Jagadish, Al-Khalifa, Chapman, Lakshmanan, Nierman, Pappas, Patel, Srivastava, Wiwatwattana, Wu & Yu 2002) is another influential prototype system. It has a well-implemented cost-based optimizer that examines alternate structural join orders (Wu, Patel & Jagadish 2003a) using histograms for size estimation (Wu, Patel & Jagadish 2003b). There are other interesting cost-based

models proposed for applying different size-estimation methods (Wang, Jiang, Lu & Yu 2003, Wang, Jiang, Lu & Yu 2004, Zhang, Özsu, Aboulmaga & Ilyas 2006) for XML query optimization. The emphasis of our work is not on the conventional cost-based approaches, but on the leverage of the unique features of XML data on XML query optimization. The approaches we proposed and demonstrated with *Determined* do not exclude but complement other approaches for XML query processing and optimization. In (Chung & Kim 2002), the idea of using the DTD knowledge to guide query processing was proposed. In (Wang & Liu 2003), the authors proposed two types of optimization: path complementing and path shortening.

In addition, commercial systems such as Oracle (Liu, Krishnaprasad & Arora 2005) and DB2 (Balmin, Eliaz, Hornibrook, Lim, Lohman, Simmen, Wang & Zhang 2006) both claim support for native XML processing. The Oracle XQuery optimization approach (Liu et al. 2005) rewrites XQuery into SQL operators and constructs with XML extensions, which are then optimized by the underlying relational optimizer and executed by the underlying relational execution engine. This approach renders a tight integration between XQuery and SQL/XML support within an ORDBMS kernel. DB2 XML (Balmin et al. 2006) claims itself the first truly hybrid system that stores XML documents on disk pages in tree structures, matching the XML data model. Cost-based optimization in DB2 XML is part of a multi-step query compilation process. Major extensions were made to the DB2 UDB compiler to support XQuery and SQL/XML queries. What is particularly interesting in DB2 is the introduction of the new metric called fanout, which is used to determine the number of items that can be reached via XPath navigation. Neither Oracle nor DB2 XML adequately exploits query pattern minimization and/or semantic XML query optimization, and neither provides specialized algorithms for accelerated XML query execution.

Tree Pattern Query Evaluation

XML possesses a tree-structured data model, and XML queries naturally come with one or more tree patterns (also called *twig* patterns), typically associated with additional predicates on the tree nodes (or their contents). Containment is a primitive operation that forms the edges between the tree nodes. An edge in a twig pattern requires that data elements in the query result must demonstrate the stated containment relation. So, efficient support for the evaluation of the containment operations is critical for XML queries. The containments can be evaluated one-by-one or holistically for a twig pattern as a whole. This leads to two general types of methods: structural joins vs. holistic twig joins.

The algorithms proposed in (Zhang, Naughton, DeWitt, Luo & Lohman 2001) and (Al-Khalifa, Jagadish, Patel, Wu, Koudas & Srivastava 2002) are representative structural join algorithms. However, efficient support for special-case containments (e.g., *pure* and *negated* containments) is rarely seen in literature. *Determined* incorporates a family of special-purpose algorithms for efficient processing of *pure* and *negated* containments.

Holistic twig join algorithms (Bruno, Koudas & Srivastava 2002, Jiang, Wang, Lu & Yu 2003, Jiang, Lu & Wang 2004, Chen, Lu & Ling 2005, Yu, Ling & Lu 2006) have demonstrated superior performance over structural joins due to their effectiveness in dampening irrelevant intermediate results. However, what hinders the practical use of holistic twig join algorithms is that all the algorithms proposed so-far do not have an integral support for all the three logical: AND, OR, and NOT. To date we only see the work of Jiang et al (Jiang et al. 2004) trying to solve the problem of OR in the framework of a holistic twig join algorithm, and the work of Yu et al (Yu et al. 2006) trying to deal with the NOT predicate. We have recently come to an innovative idea for solving the problem of all-twigs holistically. We will discuss this idea with more details later.

Other Related Work

Choosing/designing a proper algebra is an important issue for query optimization in modern database systems. For XML query optimization, TAX (Jagadish, Lakshmanan, Srivastava & Thompson 2001) and XAL (Frasincar, Houben & Pau 2002) are two worth-mentioning algebras. TAX is a tree-based algebra, and XAL is a node-based algebra. In our opinion, there are two kinds of trees that are equally important for XML queries: pattern trees and

operation trees. Pattern trees vest great expressive power to algebraic representation of XML queries, while operation trees form the basis of manipulation for query optimization. ePAT or extended PAT (Salminen & Tompa 1994) is the algebra we adopted in *Determined* for logical XML query optimization.

Other related work includes efficient execution mechanisms for XML queries such as (Florescu, Hillery, Kossmann, Lucas, Riccardi, Westmann, Carey, Sundararajan & Agrawal 2003, Hündling, Sievers & Weske 2005, Fegaras, Dash & Wang 2006) and de-correlating nested XML queries (May, Helmer & Moerkotte 2003, May, Helmer & Moerkotte 2004).

3 Constraints, Algebra, and Equivalences

Algebraic approach has long been used for query optimization. Typically, query equivalences are identified based on an appropriate algebra, and queries (represented as algebraic expressions) are then equivalently transformed using the equivalences in order to identify an optimal query plan. ICs (integrity constraints) represent an additional asset that can be beneficially applied to this process. Constraints, algebra, and equivalences are therefore important issues regarding XML query optimization, and will be briefly addressed in this section as preparation for our subsequent discussion.

3.1 XML Constraints

In *Determined*, we regard DTD/XSD as part of a database schema, and ICs are either explicitly specified or derived from DTDs/XSDs. Generally, ICs capture a certain kind of data semantics. In the context of XML, the data semantics has an additional dimension — structure semantics — corresponding to the structural facet of XML data. The exploitation of the structure semantics forms a unique aspect in XML query optimization. In *Determined*, a great deal of effort has been devoted to the exploitation of the structure semantics for XML query optimization (Che, Aberer & Özsu 2006).

In the following, we review the ICs explored in *Determined*.

- **Required child** (denoted as $t_1 \rightarrow t_2$): in any data tree complying with a given DTD/XSD, every node (element) of type t_1 must have a child node (as subelement) of type t_2 ; analogously for **required descendant** (denoted as $t_1 \Rightarrow t_2$).
- **Subtype** (denoted as $t_1 \leq t_2$): in any data tree complying with a given DTD/XSD, every node of type t_1 must also be of type t_2 . Obviously, subtype is a reflective relation.
- **Required parent** (denoted as $t_1 \leftarrow t_2$): in any data tree complying with a given DTD/XSD, every node of type t_1 must have a parent node (the enclosing element) of type t_2 ; analogously for **required ancestor** (denoted as $t_1 \Leftarrow t_2$).
- **Obligation** (denoted as $t_1 \Downarrow t_2$): this term refers to required child and/or required descendant constraints.
- **Exclusivity** (denoted as $t_1 \Uparrow t_2$): if, according to a given DTD/XSD, the elements of type t_1 are *only* allowed to be the subelements of a specific type, say, t_2 , then we say that the elements of type t_1 are exclusively contained in the elements of type t_2 .

Obviously, obligation reflects the perspective from an enclosing element to the enclosed subelements, while exclusivity reflects the reverse — i.e., from an enclosed element to the enclosing element. Nevertheless, the two concepts are not symmetric, and one cannot be induced from the other.

- **Path constraint**: if, according to a given DTD/schema, we can infer that every path from an element of type t_1 to an element of type t_2 (toward the root) goes through an element of type t_3 in every XML data tree complying

to the DTD/XSD, then we say that a *path constraint* exists between t_1 and t_2 on t_3 , denoted as $t_1 \parallel_{t_3} t_2$, and t_3 is called an **entrance location** (or EL for short) for t_1 and t_2 .

For example consider $a \rightarrow (b, ((b, c)|d)), b \rightarrow ((e|f), (g|h)), e \rightarrow (i)$; in any XML document satisfying this DTD, every path from an a node to an i node certainly contains a b node.

- **Co-occurrence:** if, according to a given DTD/schema, we can infer that every element of type t_1 that has a subelement (child or descendant) of type t_2 must also have a subelement of type t_3 , and vice versa, then we say a *co-occurrence* constraint holds between t_2 and t_3 with regard to t_1 , denoted as $t_2 \sqcap^{t_1} t_3$, or $t_2 \sqcap t_3$ for short if the omission of t_1 does not cause any confusion.

3.2 The ePAT Algebra

W3C is now finalizing its XQuery proposal as a standard query language for XML data. The key notion, *path expression*, in XQuery is borrowed from XPath. Both XQuery and XPath are essentially an expression language — i.e., 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 the same data model that is centered around the tree structure. Our effort for XML query optimization has been focused on a core subclass of XML queries that can be represented using one or more XPath expressions. In *Determined*, XML queries are translated into ePAT algebraic expressions for query optimization and evaluation. ePAT is an extended version of the PAT algebra (Salminen & Tompa 1994) for XML. In the following we present a restricted version of our ePAT to serve the subsequent discussion in this article.

An ePAT expression is generated according to the following grammar:

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

“E” (as well “E1” and “E2”) stands for an ePAT expression. etn , as the only atomic expression, retrieves the whole extent (i.e., all the instances) of the element type named etn . The expression (E) produces the same result as E . \cup , \cap and $-$ are the three standard set operations. ePAT requests type compatibility for its set operations. $\sigma_r(E)$ and $\sigma_{a,r}(E)$ are the two basic selection operators applied to the textual contents and attribute values of elements, respectively. The “r” parameter in the two selection operators introduces a regular expression specifying a matching condition on the textual contents or attribute values of elements, and the “a” in $\sigma_{a,r}(E)$ designates a particular attribute name. \subset returns elements of the first argument that are *contained in* an element decided by the second argument, and \supset returns elements of the first argument that *contain* an element of the second argument. $\not\subset$ and $\not\supset$ are the negated form of \subset and \supset , respectively. Accordingly, $\not\subset$ returns elements of the first argument that are *not* contained in any element decided by the second argument, and $\not\supset$ returns elements of the first argument that do *not* contain any element of the second argument. \bowtie_c is a “power” join operation that can be tailored to almost any specific form of joins such as structural joins (Zhang et al. 2001, Al-Khalifa et al. 2002), unstructural joins¹, and Cartesian products, based on the nature of the join predicate specified by the join condition c . π_{pl} is the projection operator that carries a parameter, pl , which designates a projection list. I connotes the application of a relevant index (e.g., element content indices, attribute value indices, and structure indices). Finally, $-E$ is the “negated” form of E where the E subexpression is limited to only a selection or a containment subexpression. When it is a selection, the negation is logically applied to the predicate of the selection; when it is a containment, say, \supset , the negation is applied to the \supset operation (meaning “does not contain”). Note that $\not\supset$ is not semantically equivalent to \subset by any means. We will show how specialized join algorithms can be used to provide efficient support for “negated containments”.

¹Unstructural joins are joins that are not based on checking any structural relationship.

It is worthwhile to mention that in ePAT, the containment operations, \supset and \subset , are redundant — i.e., they can be represented as a corresponding structural join followed by a leftward projection. Yet, there are advantages of retaining this redundancy (in a similar spirit as the redundant retention of the natural join in the relational algebra): as containments dominate the structural relationships in XML data, special support provided for containments can greatly facilitate exploiting structure semantics of XML data for query optimization and accelerated evaluation.

The following example shows a possible ePAT representation of a real query.

Example. “Find *open_auction* that have not received any bid” (i.e., without any *bidder* subelement). This query is conveniently represented as the following ePAT expression:

$$\textit{open_auction} - (\textit{open_auction} \supset \textit{bidder})$$

Our ePAT algebra has an interesting property: every ePAT expression, say E , evaluates to a set of elements of a single type, namely $\tau(E)$.

3.3 Equivalences

Due to the relatively large set of operators in ePAT (see Section 3.2) and the high interaction among them (commensurate to the high complexity of the XML data model), there can be potentially a huge set of equivalences. Some of them are straightforward, while most are not. When various ICs, as introduced in Section 3.1, are taken as additional resources, one shall expect further more equivalences. Equivalences form the basis of transformation-based query optimization, which dominates the query optimization in modern database systems. In the following, we provide a glimpse of the equivalences exploited in *Determined* by giving a few samples.

Our equivalences are identified from three major sources: set-theory (recall that ePAT is set-based), explicit constraints (trivially derived from explicitly given DTDs/XSDs), and the implied structure knowledge (as a result of deeper reasoning of DTD/XSD).

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

$$\mathcal{E2}. E1 \supset E2 \iff \phi \text{ if } \tau(E1) \text{ does not contain } \tau(E2) \text{ per the DTD/XSD}$$

$\mathcal{E1}$ is derived from the set-based nature of the ePAT operations; it is one of the many forms of the commutativity laws of ePAT (Che et al. 2006). $\mathcal{E2}$ is trivially derived from the given DTD/XSD definition.

The exclusivity and obligation constraints each gives directly an equivalence per their respective definition as listed below (notice the equivalences are by no means symmetric):

$$\mathcal{E3}. E1 \subset E2 \iff E1 \text{ if } (\tau(E1) \uparrow \tau(E2)) \text{ and } \textit{free}(E2) \text{ }^2$$

$$\mathcal{E4}. E1 \supset E2 \iff E1 \text{ if } (\tau(E1) \downarrow \tau(E2)) \text{ and } \textit{free}(E2)$$

The entrance location (or path constraint) alone gives a group of equivalences, and we present $\mathcal{E5}$ below as an example (note that, in $\mathcal{E5}$, operation \supseteq is a generic form that represents \supset or \subset ; this form may appear elsewhere in this article):

$$\mathcal{E5}. E1 \supseteq E2 \iff E1 \supseteq (E3 \supseteq E2) \text{ if } (\tau(E1) \parallel_{E3} \tau(E2))$$

When multiple constraints are collaboratively applied, more equivalences are yielded such as the following:

$$\mathcal{E6}. E1 \subset E2 \iff E1 \subset E3 \text{ if } (\tau(E1) \parallel_{E3} \tau(E2)), (\tau(E3) \uparrow \tau(E2)), \text{ and } \textit{free}(E2)$$

$$\mathcal{E7}. E1 \supset E2 \iff E1 \supset E3 \text{ if } (\tau(E1) \parallel_{E3} \tau(E2)), (\tau(E3) \downarrow \tau(E2)), \text{ and } \textit{free}(E2)$$

²Condition *free*(E) holds iff the expression E evaluates to the full extent of type $\tau(E)$.

4 Architecture and System Overview

We are interested in developing novel techniques for XML query processing. To this end, we developed the *Determined* prototype system as a test-bed for our research. Nevertheless, *Determined* is not intended to be a full-fledged DBMS for XML. In order to quickly have a running test-bed, we built *Determined* on top of Oracle. In other words, we use Oracle only as a handy storage system for fast building of our test-bed, and studying the interaction between native XML processing and the underline RDBMS is not our interest.

Determined comprises most of the major components that can be found in a standard DBMS (except for transaction and recovery management modules). The overall architecture of *Determined* is depicted in Figure 1. In the sequel, we provide a brief description of each of the major components.

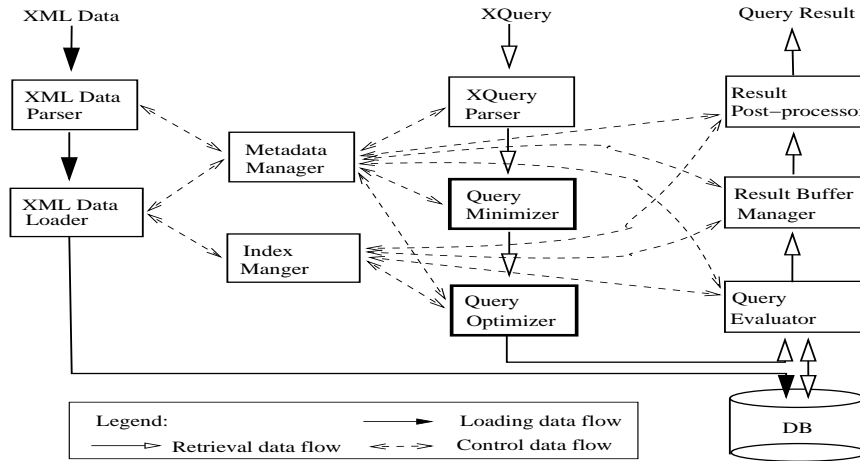


Figure 1: System Architecture of *Determined*

4.1 Data Storage

In *Determined*, the XML Data Parser takes source XML documents as input, validates them and produces a streams of annotated XTM data elements. The DTD/XSD of the source data, if not yet in the system, is extracted simultaneously and stored in the system under the control of the Metadata Manager. The Metadata Manager stores DTD/XSD as a graph, called the *DTD-graph*. In addition to the DTD/XSD, the Metadata Manager also collects and manages other metadata, including a variety of ICs and the index information, etc. Most ICs are originally implied in the DTD/XSD, and are extracted (through pre-computation) from the DTD-graph. The ICs are then explicitly stored together with the DTD-graph, and are made *interrogational*, i.e., they will become responsive to enquiries such as *whether the elements of type A is exclusively contained in the elements of type B* by invoking a method with signature that looks like *A.ex(B)*. The output of the XML Data Parser is then pipelined to the XML Data Loader that will (1) produce DTD/XSD-to-relation mappings, (2) with the help of the Index Manager (which accepts user instructions for indexing) create indices, (3) shred and load XML data into the relations created according the performed mappings. The interaction between the XML Data Loader and the host Oracle RDBMS is through dynamically interpreted SQL scripts.

The XML-to-relation mapping scheme adopted by *Determined* is a straightforward one, named “node table” mapping scheme (because our main interest is not in the mapping but in the upper-level, native XML query processing and optimization). Basically, this mapping scheme maps each (type) node in a DTD-graph to a separate table (or relation), which has a designated column for each attribute of the elements of the mapped type. Each table typically consists of the following columns: *doc-id*, *element-id*, *parent-id*, *element-value*, and an *attribute-value* column

corresponding to each attribute that the type has. *Element-id* encodes the region information (i.e., the *start* and *end* positions) of the elements within the original XML document to facilitate the implementation of structural joins (Zhang et al. 2001, Al-Khalifa et al. 2002) and holistic twig joins. In the adopted mapping scheme, values are “inlined”, which had been proven by Florescu et al (Florescu & Kossmann 1999) to be an effective choice for efficient XML query processing in an RDBMS environment. *Determined* maintains a separate system catalog, named the “mapping table”, which has an entry corresponding to each element type in order to keep important mapping information that may be needed later on during query translation (from optimized ePAT expressions to SQL queries).

4.2 Metadata

In addition to the source XML documents that are mapped to and dispersed over various relations managed by the Oracle RDBMS, *Determined* also maintains a series of metadata, including the following:

- The aforementioned “mapping table” which records detailed mapping information of the element types (including attributes).
- The DTD/XSD of the source documents, which are either explicitly given or extracted from the source data if not explicitly given.
- Various ICs. Many of them capture a variety of semantic knowledge regarding the structure and composition of the source XML documents and elements. ICs play an important role in our approach, and are used for both minimization and semantic optimization.
- Indexing information, which will be exploited during the phase of semantic optimization; the goal here is to enable the application of a potential index (mainly structure indices).

4.3 Indices

Indices have long been used as a primary mechanism to facilitate information retrieval from all kinds of information resources, database systems in particular. In the context of XML databases, queries are formed using three basic types of search criteria, which respectively specify the search conditions on textual contents, attribute values of the data elements, and the structural relationships among the data elements. Accordingly, in *Determined* we provided three basic types of indices to expedite the evaluation of these three types of conditions. These indices include *element content indices*, *attribute value indices*, and *structure indices*.

As each attribute of an element is mapped to a separate column in a corresponding table, the “standard” indexing mechanisms for textual values provided by the underlying RDBMS are straightforwardly used for attribute values; and element content indexing is achieved in the same way.

The rich structural relationships among XML data elements represent a unique and important feature of XML data, and examining these structural relationships during query evaluation is often required and costly. Structure indices can be very helpful for accelerating XML query evaluation in this case. In *Determined*, structure indices are provided to help evaluating various containment relations that are typically specified by XML queries. When frequent examination of the containment relation between two different types of elements is foreseen, a structure index is beneficially built between these two types (of elements). A metaphor for the structure index in *Determined* is the concept of *superhighway* — that should benefit not just the two metropolises directly linked to the two ends of it, but also the surrounding and suburban areas. In *Determined*, a structure index, denoted as $I(t_1, t_2)$, models the pre-computation of a structural join between the elements of the two types, t_1 and t_2 ; such a pre-computed result is kept in a binary relation as part of the system’s metadata.

The XML Data Loader provides a simple user interface to receive user instructions for building structure indexes. Usually, we do not expect to build a structure index for every pair of element types because of the maintenance cost. Only for the containments that are anticipated to be heavily used as query criteria, is a structure index built for a pair of such related types (of elements). During query optimization, beneficial cases of applying a potential structure index is explored. At the evaluation phase, a proper structural join algorithm is called upon for those involved containments that do not have a beneficial structure index to use.

4.4 Query Processing

In *Determined*, upon submission, a user query first gets validated by the XQuery Parser module; if valid it is then translated into an internal form of representation (an ePAT expression tree) by the ePAT Converter (a sub-module of XQuery Parser, not explicitly shown in Figure 1). The ePAT expression then goes through the succeeding two core modules in turn: the Query Minimizer and the Query Optimizer (more details of these two modules will be provided later). Generally, *Determined* performs two different types of optimization at two separate stages: tree pattern minimization by the Query Minimizer, and deterministic optimization by the Query Optimizer. The Query Minimizer module focuses on minimizing the tree pattern(s) that every XML query naturally comes with. The Query Optimizer targets mainly at the opportunity of bringing a structure index into a query and this is achieved by heuristic-based deterministic transformations (Che et al. 2006). The Query Optimizer starts by taking over the output of the Query Minimizer module and conducts three consecutive phases of transformation for optimization: normalization, semantic optimization, and cleaning-up. In the Query Optimizer module, a space is reserved for physical XML query optimization (typically cost-based). But our interest at this time is not in the physical optimization for two considerations: first, the rich semantics of XML queries opens a big realm for extensive, logic-level optimization, which alone is expected to substantially improve the execution performance of XML queries (Che et al. 2006), and thus reduces the necessity for a subsequent physical optimization module; second, there exist acceptably good results of physical XML query optimization such as (Jagadish et al. 2002) that can be integrated into *Determined* to complement the deterministic query optimization approach.

The Query Optimizer produces optimized ePAT expressions, which are then fed to the Query Evaluator. A sub-module, called SQL Converter (not explicitly shown in Figure 1), in the Query Evaluator takes each optimized ePAT expression and translates it into one or more SQL queries. To do so, the SQL Converter needs to access the mapping metadata managed by the Metadata Manager (see Figure 1). The translated SQL queries are then sent to the host query engine (i.e., the Oracle query engine) for execution. The obtained results are kept in a main-memory buffer managed by the Result Buffer Manager (Figure 1). As the Oracle query engine is a closed one, we implemented various structural join algorithms outside the host query engine. These algorithms are called by the Result Buffer Manager to work on the obtained SQL results. The results obtained are finally wrapped up by the Result Post-processor module and delivered to the user in XQuery compliant formats.

Determined has a friendly GUI. Figure 2 shows a snapshot of the system interface, when the ePAT format of an example query (before optimization) and the optimized one (in SQL format) were on display. At the lower part of the figure are the various timing results, e.g., optimization time and execution time of the query, and some statistic information.

As indicated earlier, *Determined* was designed as test-bed to support our team’s investigation on XML query optimization. The core in the Query Optimizer module is a rule-based transformation system. In order to help investigate the behavior of individual rules and alternate orders of the rules in the rule base (the order affects which rule will be chosen next when there are multiple ones available and applicable), *Determined* also provides a convenient interface (see Figure 3) for managing (e.g., to enable/disable a rule) and observing these rules.

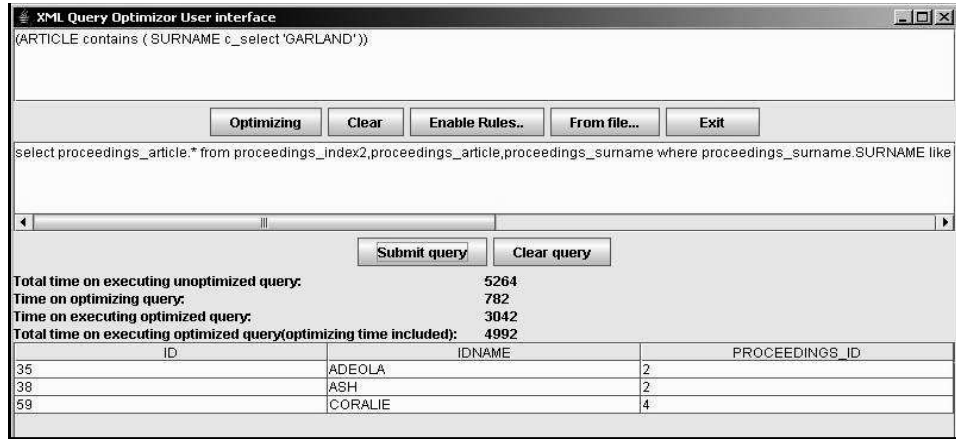


Figure 2: System Interface (1)

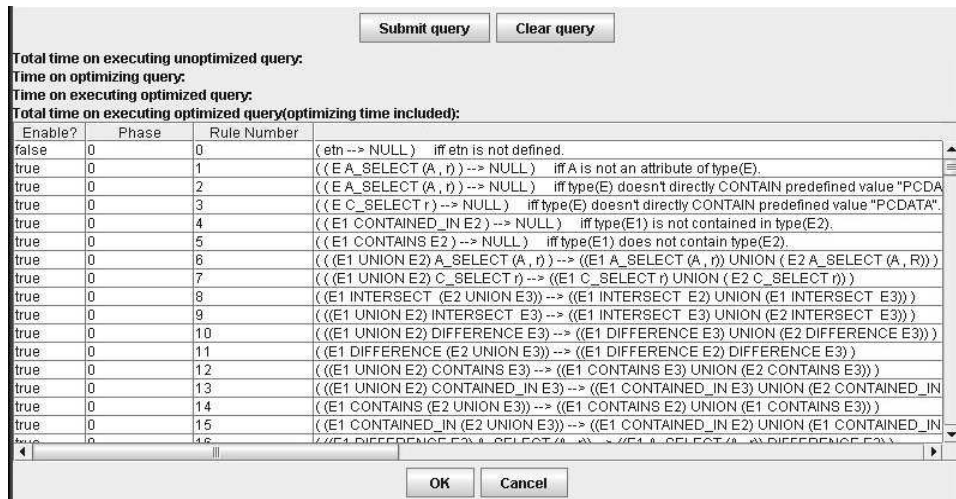


Figure 3: System Interface (2)

5 Tree Pattern Minimization

All XML queries naturally carry one or more tree patterns and the queries are often called TPQs as in (Amer-Yahia et al. 2001) for short. The TPQs given by users are typically not minimal, especially when various ICs are available and considered. This makes tree pattern minimization an integral part of XML query optimization in a broad sense. So far, the most efficient known algorithm proposed for TPQ minimization is the *MinimizeCTPQ* algorithm (Ramanan 2002) (though with flaws). But it only covers the two simplest types of ICs: required children and required descendants. We have developed a novel TPQ minimization scheme (called *Allying*) and successfully integrated it into *Determined*. *Allying* has optimal time complexity, i.e., $O(n^2)$ w.r.t. the query size n , and is the most powerful scheme for TPQ minimization in terms of the wide variety of ICs it can explore. In this section, we provide an overview of *Allying* and illustrate how it works (more details of *Allying* can be found in (Che 2007)).

A TPQ is not minimal if certain redundancy exists. The redundancies can be simply identified as syntactical redundancies (without considering the effect of any ICs) or semantic redundancies (under the consideration of relevant ICs). It has been recognized that to fully discover potential redundancies, augmentation of the input TPQ by incorporating the effect of all relevant ICs is a necessary first-step (Amer-Yahia et al. 2001, Ramanan 2002). Yet, it is this desired augmentation step that at the same time caused significant increase in the complexity of these minimization

algorithms. Ramanan (Ramanan 2002) made the first effort (Ramanan 2002) to avoid physically augmenting an input TPQ by “mimicking” the effect of augmentation, and came up with the runtime-optimal algorithm, *MinimizeCTPQ*. However, this simple “mimicking” mechanism can only incorporate the required child/descendent constraints.

With a given set C of ICs, the closure C^+ of C comprises C and all derived constraints. C^+ can be computed by using the 7 expansion rules proposed by Ramanan (Ramanan 2002) and the 3 additional rules that we added for incorporating the effect of the co-occurrence constraint (Che 2007). As a result, the required child/descendant, subtype, and co-occurrence constraints can all be incorporated into C^+ in the form of required child/descendant, resulting in a DAG (Directed Acyclic Graph) called *constraint graph* (or *IC-graph*), denoted as $G_C = (N_C, E_C)$, where N_C is a set of nodes from Σ (which is the set of all type names under consideration), and E_C is a set of either c-edges (child edges) or d-edges (descendant-edges). The constraint graph G_C may not be a connected one, and does not necessarily cover the entire C^+ (actually, both the EL and the exclusivity constraints cannot be incorporated into the IC-graph G_C). Thus a full representation of the closure C^+ should be such a triple: $C^+ = \langle G_C, EL_C, EX_C \rangle$, where G_C denotes the IC-graph $G_C = (N_C, E_C)$, EL_C represents all EL constraints, and EX_C stands for all exclusivity constraints.

Ramanan’s work (Ramanan 2002) demonstrated that the “graph simulation” concept is an effective mechanism to help identify functionally redundant nodes in a TPQ. What we still need is a more powerful mechanism capable of incorporating a broad spectrum of ICs for minimizing TPQs. Our approach is also based on this graph simulation concept, but we redefine (generalize) the concept at multiple tiers in order to incorporate broader ICs into our minimization scheme. In the following, we present the multi-tier definition of “simulation” which actually consists of three separate definitions, and each subsequent one is based on the previous one.

Definition 5.1 (Tier I: Literal Simulation) *Let $Q = (N, E)$ represent a TPQ with a set N of nodes and a set E of directed edges, and each node $u \in N$ have a type $\tau(u)$ associated with it. We say that node $u \in N$ is literally simulated (or *l-simulated*) by node $v \in N$, denoted as $u \preceq_l v$ (accordingly, v is called an *l-simulator* of u), whenever the following conditions hold:*

1. *If u is the output node, then $v = u$ (an output node is simulated only by itself and is never redundant.)*
2. *Preserve node types: i.e., $\tau(v) \leq \tau(u)$, which says that the *l-simulator* v must have the same or a more specific type associated with it.*
3. *Preserve child edge relationships: if u has a c-child u' , then v has to have a c-child v' such that $u' \preceq_l v'$.*
4. *Preserve descendant edge relationships: if u has a d-child u'' , then v has to have a d-child v'' such that $u'' \preceq_l v''$.*

Definition 5.1 literally considers the structural relationships between the nodes in a TPQ. It incorporates the implication of the subtype constraint (different from that in (Ramanan 2002)).

Definition 5.2 (Tier II: Semantic Simulation) *Let $Q = (N, E)$ be a TPQ with a set N of nodes and a set E of directed edges, and each node $u \in N$ have a type $\tau(u)$ associated with it. We say that node $u \in N$ is semantically simulated (or *s-simulated*) by node $v \in N$, denoted as $u \preceq_s v$ (accordingly, v is called an *s-simulator* of u), whenever the following conditions hold:*

1. *If $u \preceq_l v$, i.e., v is an *l-simulator* of u , then $u \preceq_s v$.*
2. *If u is a leaf and $\tau(u) \sqcap \tau(v)$ (i.e., a co-occurrence holds).*
3. *If u is a leaf child of w , v is a d-child of w , and u is an EL (i.e., entrance location) for $\tau(v)$ and $\tau(w)$.*

Semantic simulation incorporates two additional types of ICs: the co-occurrence and the EL constraints. Bringing in co-occurrence is straightforward, while incorporating EL constraints may need some explanation (interested readers are referred to (Che 2007)). As special cases of EL, required parents/ancestors are trivially slotted in.

Definition 5.3 (Tier III: Augmented Simulation) *Let $Q = (N, E)$ be a TPQ, consisting of a set N of nodes and a set E of directed edges, and each node $u \in N$ has a type $\tau(u)$ associated with it; let C be a set of constraints explicitly given or derived from a relevant DTD/XSD, $C^+ = \langle G'_C, EL'_C, EX'_C \rangle$ is the restriction (or projection) of the closure C^+ to Q and $G'_C = (N'_C, E'_C)$ is the restriction of the constraint graph G_C to Q . We define an a -simulator (augmented simulator) of $u \in N$ to be either an s -simulator of u found in Q or an s -simulator found in G'_C . We use $u \preceq_a v$ to denote that v is an a -simulator of u . Furthermore, v is called a virtual simulator of u if $u \preceq_a v$ and $v \in N'_C$, and an actual simulator of u if $u \preceq_a v$ and $v \in N$.*

Definition 5.3 implies the key idea of our approach — **Allying**. Intuitively, for a node u in a given TPQ Q , we want to identify which node(s) can also act the role of u and thus make u redundant. Some nodes in Q may have *undiscovered* potential of acting as u because of the effect of certain ICs. This potential is usually revealed after the TPQ is augmented by explicitly incorporating the effects of the ICs as discussed in (Amer-Yahia et al. 2001, Ramanan 2002). But we want to avoid this physical augmentation because it would substantially increase the size of Q , which in turn leads to the added time complexity of the minimization schemes. Instead, we let a query node u of type $\tau(u)$ in Q be **allied** with the corresponding type node $\tau(u)$ in G'_C so that node u can resort to $G'_C(\tau(u))$ for revealing its potential of simulating other query node(s) in Q .

Definition 5.4 (Redundant Node) *Let u be any node in TPQ Q , u is redundant if there exists an actual simulator v of u and both u and v are in the same-generation with respect to the nearest common ancestor node in Q .*

Notice that the existence of an actual simulator is a necessary but non-sufficient condition for a query node to be redundant. Using our much generalized simulation concept (Definition 5.1 to 5.3), we can identify most potential redundancies in a TPQ. Yet, there are non simulation-based redundancies that need to be handled differently. In *Determined*, we developed a powerful minimization algorithm (also called *Allying*) that implements the above-presented minimization scheme, exploring broad ICs.

The limitation that *MinimizeCTPQ* (Ramanan 2002) suffers from that the algorithm can only deal with the required child/descendant constraints. Our algorithm greatly exceeds this milestone algorithm (Ramanan 2002) in that much broader ICs are now incorporated into the minimization scheme. We explain how we achieved this goal below.

First, let us have a closer look at how *MinimizeCTPQ* works and how our *allying* idea is different. We use the same example that Ramanan used (Ramanan 2002) to make a contrasted illustration. Assume the given constraint set is $C = \{e \Rightarrow d\}$ and let the input TPQ Q be the one shown in Figure 4, where vi ($i = 1, \dots, 8$) denotes the query nodes and a, b, c, \dots refer to the types associated to the nodes, and $v3$ is the output node. With a given TPQ $Q = (N, E)$, *MinimizeCTPQ* relies on a simulation concept (weaker even than our tier-I definition, cf. Definition 5.1) for identifying and removing redundant nodes. For $u \in N$, let $sim(u) \in N$ stand for the set of nodes that simulate u , and $anc(sim(u))$ stand for the set of ancestor nodes of $sim(u)$. Obviously, after performing an augmentation on the TPQ shown in Figure 4 by adding edge (e, d) under node $v7$ according to the given constraint, we have $v3 \in sim(v2)$, thus the whole subtree rooted at $v2$ is redundant and can be removed. *MinimizeCTPQ* introduces *auganc(sim(u))* (i.e., the augmented set of the ancestors of $sim(u)$) in order to *mimic* but not to physically perform the desired augmentation on Q . *auganc(sim(u))* is a superset of $anc(sim(u))$; it also contains all the nodes $v \in N$ such that the constraint $\tau(v) \Rightarrow \tau(u)$ is in C . Here, a subtle point is that the v node that is added to *auganc(sim(u))* has to be an existent node in the input TPQ Q . Applying to this example (Figure 4), as $sim(v5) = v5$, we get

$auganc(sim(v5)) = \{v1, v2, v3, v7\}$. Notice that $v3 \in auganc(sim(v5))$ and $v7 \in auganc(sim(v5))$ due to the constraint, $e \Rightarrow d$. Apparently, the limitation of Ramanan’s *mimicking* idea comes from the fact that it only allows *mimicking* augmentation at one level — that is, only on the original query node, but not also on the *mimicked* nodes. Our algorithm exceeds this via the adoption of our *allying* idea embodied in Definition 5.2. With *allying*, our algorithm identifies simulators of a query node from both the TPQ and the IC-graph, and differentiates actual simulators (found only in the TPQ) from virtual simulators (found only in the IC-graph).

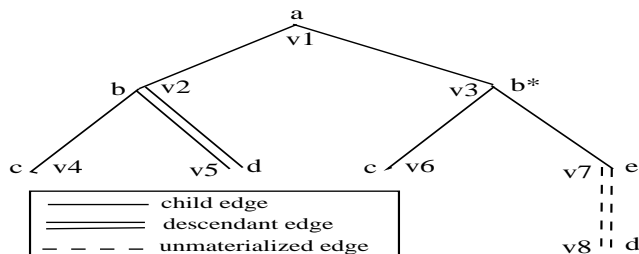


Figure 4: Illustration example 1

6 Deterministic Query Optimization

The ePAT algebra adopted in *Determined* has a large set of operations, which is commensurate to the inherent high complexity of the XML data model. Accordingly, we identify a large set of equivalences that can be used for transformation-based XML query optimization. Traditionally, a query optimizer (like in (Graefe & DeWitt 1987)) performs query optimization through extensive query transformations based on well-defined equivalences. This approach is not very suitable for XML queries because a too large search space would need be explored, leading to unsatisfactory optimization performance. This observation motivated the deterministic optimization approach adopted in *Determined* — that performs only *deterministic* transformations for accelerated XML query optimization. This novel approach is facilitated by the semantic-rich nature (characterized by the wide variety of ICs) of XML data.

Generally, in our approach, equivalences are not directly used to conduct query transformation (this is in contrast to the conventional approaches such as (Graefe & DeWitt 1987) that use equivalences to enumerate alternate query plans). Instead, we developed a set of deterministic transformation rules based on the identified equivalences in combination with various heuristic and relevant ICs. In contrast to the *two-way* nature of equivalences, a deterministic transformation rule, once is applied, produces an output that is required to be sufficiently better than its input form. The input form thus is immediately and permanently pruned — i.e., excluded from further consideration in the future. Therefore, all performed transformations are *deterministic* — have a *one-way* nature. This feature is critical for the high performance of the query optimization *Determined*. Query optimization in our setting consists of a linear sequence of deterministic transformations, which step-by-step improves the input query, leading to the final, optimal query format (logical plan).

Introducing a structure index into a query is a key step in XML query optimization. However, a potential index, though available, may not be applicable to a given query due to the particular form that query is in. In *Determined*, we solve this problem via heuristic-based, deterministic transformations for eventually enabling the application of a potential index to an input query.

The deterministic optimization in *Determined* is obtained through three consecutive phases: *normalization*, *semantic optimization*, and *simplification (or cleaning-up)*, of which the semantic optimization is at the core, which implements the most of our optimization strategies. The normalization phase is a preparation phase that mostly reorders the operators in an input query according to the following general order (from bottom to up in the expression

tree): etn , \cap , $-$, σ , \supseteq , \cup . Interestingly, after normalization, all \cap and $-$ operations sediment to the bottom of the expression tree and eventually get eliminated from the query.

The main phase, semantic optimization, starts with a normalized ePAT expression, explores deeper structure knowledge of the XML data to, first, render a possible opportunity for applying a potential structure index, and, eventually, derive an obviously better substitute format for the input query.

The final phase, simplification, performs a thorough cleaning-up. This task is necessary because the main phase, semantic optimization, may bring in new redundancy while introducing new element types (e.g., as entrance locations) into the query. To this end, a dedicated set of simplification rules are called upon during this final phase.

Now we shed some more light on the kernel task accomplished by the semantic optimization phase. In *Determined*, we differentiate the following three cases regarding using a structure index in a query: (1) a structure index is available and applicable to the input query (this is the most preferred case); (2) an structure index is available but not applicable because the input query is in an unfavorable format; (3) a structure index is available but is not even relevant to the input query, however, some radical transformations can be invoked to make the index relevant and applicable.

We provide an example rule for each of the cases identified above.

R1. $(E1 \supseteq E2) \implies (I_{\tau(E1)}(E2) \cap E1)$ if a structure index $I(\tau_{E1}, \tau_{E2})$ is available

R2. $(E1 \supseteq (E2 \supseteq E3)) \implies ((E1 \supseteq E2) \supseteq E3)$ if $(\tau(E1) \parallel_{\tau(E2)} \tau(E3))$, and $I(\tau_{E1}, \tau_{E2})$ is available

R3. $(E1 \subset E2) \implies (E1 \subset E3)$ if $(\tau(E1) \parallel_{E3} \tau(E2))$, $(\tau(E3) \uparrow \tau(E2))$, $free(E2)$, and $I(\tau(E1), \tau(E3))$ available

R1 applies an available and applicable structure index straightforwardly. R2 makes an available structure index applicable; it makes the subexpressions $E2$ and $E2$ interact directly and thus enables the application of the available structure index. In the case of R3, the available structure index between $\tau(E1)$ and $\tau(E3)$ is not even relevant to the input query, and this rule brings in a new element type $E3$, leading to the application of the potential structure index.

A distinguished feature of our deterministic optimization approach is that it applies exclusively deterministic transformations on queries at the logic level for better optimization efficiency. The structure knowledge of XML data and a variety of optimization heuristics for XML queries are characterized and manipulated more easily at this relatively high level. Because of the determinism so rendered, the typical problems that haunted many rule-based system — the uncontrollable runtime behavior and the high complexity of runtime control — get circumvented successfully. Another interesting point about our approach is that the two set operations, \cap and $-$, eventually get eliminated from the optimized query format. And the third important feature of *Determined* is the adoption of a set of specialized algorithms for *pure* and *negated* containments. We discuss this last feature in the next section.

7 Specialized Support Algorithms

In this section, we address two types of special runtime support algorithms for fast XML query evaluation. The first type is for special cases of the containment operations, and the second type is holistic computation for all-twig patterns that contain AND, OR, and NOT predicates.

7.1 Special Containment Join Algorithms

First, we look at the motivating example introduced in Section 3.2 once again. This query asks for “all `open_auctions` that have not received any bid”. One possible way of representing this query using ePAT is “`open_auction - (open_auction \supset bidder)`”. A reasonable plan for this query is “`open_auction - $\pi_{open_auction}(open_auction \bowtie_{\supset} bidder)$` ”, and the MPMGJN (Zhang et al. 2001) structural join algorithm, for example, may be used for implementing the \bowtie_{\supset} operation. To execute this plan, one needs to, first, retrieve all `open_auction` elements and all `bidder` elements;

next, evaluate the structural join operation using the MPMGJN algorithm; after that, project the join results to `open_auction`; and, finally, calculate the difference between the set of all `open_auction` elements and the set computed at the last step. An alternative plan is “`open_auction` $\not\supset$ `bidder`”. That consists of only a single *negated containment* operation, $\not\supset$, and is obviously more succinct and more efficient. Naturally, we would expect to have a specialized algorithm to directly implement the *negated containment* operation. In *Determined*, we implemented a family of specialized algorithms for special cases of containments. The above example also reveals a truth that negated containments are as important as regular containments for ordinary user queries over XML data sources. We will show that dedicated algorithms designed for special-case containments can lead to superior performance to the general-purpose algorithms as found in (Zhang et al. 2001, Al-Khalifa et al. 2002).

It has been shown (Zhang et al. 2001, Al-Khalifa et al. 2002) that structural joins outperform the simple navigation and the standard RDBMS joins by several orders of magnitude. Negated containment is another important operation that may greatly facilitate XML data querying and thus deserves highly-efficient, specialized implementation. In (Che 2005), we made an initial exploration on this line and developed the negated counterparts of the well-known structural join algorithms, MPMGJN (Zhang et al. 2001) and Stack-Tree-Desc (Al-Khalifa et al. 2002). They have shown noticeable improvement over MPMGJN and Stack-Tree-Desc for computing negated containments involved in XML queries. Our recent study reveals that it is possible to develop further efficient implementations for negated containments — not based on adapting the existing structural join algorithms, but from scratch. Our study has led to a new family of specialized algorithms for special-case containments. We address several of them below.

In order to differentiate our work from the structural joins discussed elsewhere such as (Zhang et al. 2001, Al-Khalifa et al. 2002), we use a different term — *containment joins* — to refer to the join operations for *pure* and *negated* containments. All structural joins return a list of matching *pairs* of elements from two respective lists of input elements (Zhang et al. 2001, Al-Khalifa et al. 2002). In contrast, containment joins return a *set* of elements from only one input list, while the other input list is only used as a filter to the elements in the first list. For example, the query, “find professors who have laboratories”, requires only the professors, thus returning a list of the matching pairs from both input lists is overkill and costly. The specialized algorithms we recently developed for pure and negated containments are more efficient than our prior proposals (Che 2005). When a containment join is the best fit, a structural join algorithm causes extra operations and overhead, besides the inherent low-efficiency of its own implementation. For example, with the above query example, a structural join would return us a list of the matching pairs of professor/laboratory; then we have to perform additional operations such as *projection* and *duplicate elimination* to fulfill the user’s query request. It is even worse for a negated containment because in that case we would have yet to compute the difference between all professors and the professors who have a laboratory to decide those professors who do not have any laboratory.

In *Determined*, the algorithm that computes the negated containment of form “ $A \not\supset B$ ” is called NICJoin, and the algorithm that computes “ $A \not\supset B$ ” is called NCJoin. NICJoin is the negated counterpart of ICJoin, a specialized algorithm for computing the pure containment operation of form “ $A \supset B$ ”, and NCJoin is the negated counterpart of CJoin, a specialized algorithm for computing the pure containment operation of form “ $A \supset B$ ”. Considering NICJoin and NCJoin are respectively derived from ICJoin and CJoin, in the following we present only ICJoin and CJoin due to space concerns.

Figure 5 and 6 respectively provide the pseudo code of the algorithms corresponding to the two forms of containment joins. In our presentation, we adopt the same numbering scheme as in (Al-Khalifa et al. 2002), i.e., (`DocId`, `StartPos` : `EndPos`, `LevelNum`), which encodes the doc-id, the start and end position of the element, and the nesting level of the element in the DOM tree. With both algorithms, the AList (a list of potential ancestors) and DList (a list of potential descendents) are scanned just once, and there is no need to use any stacks or queues (*cf.* the tree-merge and stack-tree algorithms in (Al-Khalifa et al. 2002)).

Both algorithms have the time complexity $\Theta(|AList| + |DList|)$, while all the structural join algorithms so far proposed such as that in (Zhang et al. 2001, Al-Khalifa et al. 2002) have the complexity of $O(|AList| + |DList| + |OutputList|)$, where $|OutputList|$ alone can be $|AList| * |DList|$, which is far larger than $|AList| + |DList|$ (i.e., *polynomial* vs. *linear* in terms of the average length of the input lists; this causes a performance difference of at least several orders of magnitude). The performance advantage of containment joins over structural joins gets further amplified when we consider the extra operations incurred when a structural join algorithm is used.

ALGORITHM ICJoin(DList, AList)

```

/* This algorithm computes the result of  $A \subset B$  */
/* input: DList is of type A and AList is of type B; both lists are sorted according to startPos, assuming */
/* all elements having the same docId. */
/* output: OList holds a sub list of DList that satisfy  $A \subset B$ . */

01 a = AList→firstNode; OList = NULL;
02 for (d = DList→firstNode; d != NULL; d = d→nextNode) {
03   while(a.end < d.end && a→hasNextNode() ) a = a→nextNode;
04   /* Now find the first possible descendant d of node a: */
05   if (a.begin < d.begin && a.end > d.end) add d to OList
06 } /* end of for loop */

```

Figure 5: The ICJoin algorithm calculating $A \subset B$.

ALGORITHM CJoin(AList, DList)

```

/* This algorithm computes the result of  $A \supset B$  */
/* input: AList is of type A and DList is of type B; both lists are sorted according to startPos, assuming */
/* all elements having the same docId. */
/* output: OList holds a sub list of AList that satisfy  $A \supset B$ . */

01 d = DList→firstNode; OList = NULL;
02 for (a = AList→firstNode; a != NULL; a = a→nextNode) {
03   while(d.begin < a.begin && d→hasNextNode()) d = d→nextNode;
04   /* Now find the first possible descendant d of node a: */
05   if (a.end > d.end && a.begin < d.begin) add a to OList
06 } /* end of for loop */

```

Figure 6: The CJoin algorithm calculating $A \supset B$.

From our containment join algorithms (Figure 5 and 6), we can easily work out their negated counterparts (for $\not\subset$ and $\not\supset$, respectively). The negated containment join algorithms have the same time complexity as their non-negated counterparts. Our new algorithms, NICJoin and NCJoin, not only appear simpler but also more efficient than our earlier proposals (Che 2005).

We have yet to address a related important issue: how to identify the cases that a negated containment join algorithm can be beneficially applied to a query. This issue mainly relates to the *set difference* operation and the *negation* operation, both are denoted by the same operation symbol, ‘-’, in ePAT (refer to Section 3.2). Again, we apply deterministic transformation to render the application of a negated containment join in a similar way to what we did for bringing a potential structure index into a query. The following transformation rules are particularly

defined to serve this purpose.

$$\mathcal{R4.} \quad \neg(E1 \supseteq E2) \implies E1 \not\supseteq E2$$

$$\mathcal{R5.} \quad \neg \sigma(E) \implies \phi(E)$$

$$\mathcal{R6.} \quad E1 - (E2 \supseteq E3) \implies E1 \cap (E2 \not\supseteq E3)$$

$$\mathcal{R7.} \quad E1 - \sigma(E2) \implies E1 \cap \phi(E2)$$

$$\mathcal{R8.} \quad E \cap S \implies S \text{ if } S \text{ is known to be a subset of } E$$

$\mathcal{R4}$ causes the unary negation operation absorbed by a succeeding containment operation (making a negated containment). $\mathcal{R5}$ causes the negation operation absorbed by the predicate of a following selection operation. $\mathcal{R6}$ and $\mathcal{R7}$ deal with the binary operation ‘-’ (stands for set difference). It is important to point out that, because of the set-oriented nature of ePAT and the type compatibility required for the set operations, $\mathcal{R6}$ and $\mathcal{R7}$ often produce an output pattern that matches the input pattern of $\mathcal{R8}$, leading to further simplification on the input expression.

7.2 Holistically Computing All-Twig Patterns

The importance of holistically computing all-twig patterns was identified several years ago (Jiang et al. 2004, Yu et al. 2006), but the finching challenge involved has prevented a complete solution being proposed till now. Recently we came up to the first complete solution. In this subsection, we discuss some basic ideas of this preliminary solution.

By default, the logical connection between the sibling nodes in twig pattern as defined by Bruno et al (Bruno et al. 2002) is AND, and in that sense every twig pattern can be called an *AND-twig*. An AND-twig additionally containing OR-predicates is called an *AND/OR-twig* by Jiang et al (Jiang et al. 2004). Analogously, we call an AND-twig that may contain NOT-predicates an *AND/NOT-twig*, and an AND-twig that may contain both OR-predicates and NOT-predicates an *AND/OR/NOT-twig* or simply an *all-twig*. Specifically, an all-twig may contain all the following types of nodes: (1) QNode — a regular query node in a twig pattern standing for a location step with name test such as “//*subsection*”; (2) ANode — an AND node; (3) ONode — an OR node; (4) NNode — a NOT node that negates the interpretation of whatever follows; (5) NQNode — the combination an NNode and a following QNode; (6) NANode — the combination of an NNode and a following ANode; (7) NONode — the combination of an NNode and a following ONode.

Generally, in an all-twig there may be arbitrarily specified AND, OR, and NOT predicates. It would be extremely difficult (if not all impossible) to develop an efficient algorithm for uniformly dealing with the all-twigs. We find it is possible to harness the “wildness” of the all-twigs through *normalization*. We define a *normalized all-twig* as one in which all the NOT operations, if any, appear only on leaves in the form of NQNodes. We can prove, for an arbitrarily given all-twig, we can obtain its normalized form by using a set of 9 normalization rules. Due to space concern, in the following we give only the first two of the normalization rules:

Rule 1: If a QNode n in an all-twig Q has a NONode child n_i , we can push down the implied NOT predicate by transforming n_i to an ANode and all its child nodes to their corresponding negated forms. This rule is illustrated by the following transformation:

$$A[\text{NOT} (B \text{ OR } C)] \implies A[(\text{NOT } B) \text{ AND } (\text{NOT } C)]$$

Rule 2: If a QNode n has a NQNode child n_i which has an ONode n_j , we can push down the implied NOT predicate by performing a commensurate transformation such as the following one:

$$A[\text{NOT } B[C \text{ OR } D]] \implies A[\text{NOT } B \text{ OR } B[(\text{NOT } C) \text{ AND } (\text{NOT } D)]]$$

With a normalized all-twig, we do not have arbitrarily appearing NOT predicates (NQNodes) in the twig — they are limited to only leaves, while the OR predicates (ONodes) may appear anywhere except for the leaves and the root. We envision two general approaches to the evaluation of such a normalized all-twig: a *genuine holistic* approach

and a split-based *semi-holistic* approach. We have developed algorithms based on both approaches. In the following, we shed slight light on only the semi-holistic approach.

The semi-holistic approach applies the divide-and-conquer method straightforwardly to all the ONodes: i.e., we split a normalized all-twig at every ONode, resulting in a number of special AND/NOT-twigs (each is basically an AND-twig with possible NNodes appearing only on leaves). We then compute these special AND/NOT-twigs using a specially designed holistic twig join algorithm (called *TwigStackN*). At the end, after merging the results computed for all the special AND/NOT-twigs, we obtain the final answer for the original all-twig.

Our *TwigStackN* algorithm is shown in Figure 8. It is based on the classic *TwigStack* algorithm (Bruno et al. 2002), but incorporates essential extension for dealing with the involved NQNodes on leaves.

The basic idea of *TwigStackN* is summarized as follows: when the next node q returned by function *getNext* (Figure 9) is a QNode, *TwigStackN* does the same as *TwigStack* (Bruno et al. 2002), and when q has only NQNode children but no QNode children, before cleaning stack S_q , we need to additionally check if there are satisfied path solutions currently on stack S_q and if yes, output them before they are popped out. This is done by the new lines (1 to 7) added to the *cleanStack* procedure (Figure 9). When q is a QNode leaf, all identified path solutions on the stacks are output just as in the original *TwigStack* algorithm through line 8 to 10 in the main algorithm (Figure 8). When q is an NQNode leaf, this means a match has been found for q and thus disqualifies any output path solutions that contain $top(S_{parent(q)})$ (this is done by line 12 in *TwigStackN*). The operation, *disqualify*, simply puts a mark on each disqualified path so that it will not be considered at the path merging phase by procedure *mergeAllPathSolutions* (line 20, Figure 8). In function *getNext*, n_{max} is computed at line 6 (Figure 8) to help skip those un-contributing elements in stream T_q , but NQNodes are not going to be used for this purpose.

Now we apply *TwigStackN* to the example twig query and the sample data shown in Figure 7. Figure 7(a) shows the twig query pattern, (b) shows the sample data tree, and (c) illustrates the sorted input streams. First, element a_1 enters stack S_A , b_1 enters S_B and then gets popped out because of the next incoming element b_2 . Then, b_2 and d_1 in turn enter S_B and S_D , respectively. Because d_1 is a leaf, path “ $a_1 - b_2 - d_1$ ” is output. Next, element b_3 comes and causes b_2 and d_1 popped out, and then c_1 comes and disqualifies every path that contains b_3 (actually there is no such paths yet). Then b_4 comes and empties S_B , then d_2 enters S_D and path “ $a_1 - b_4 - d_2$ ” is output. Unfortunately, when c_2 comes, it disqualifies every path containing b_4 , so path “ $a_1 - b_4 - d_2$ ” is rejected. Finally, path “ $a_1 - b_2 - d_1$ ” is accepted as the sole solution to the twig pattern.

The complexity result of *TwigStackN* is the same to that of *TwigStack* (details omitted).

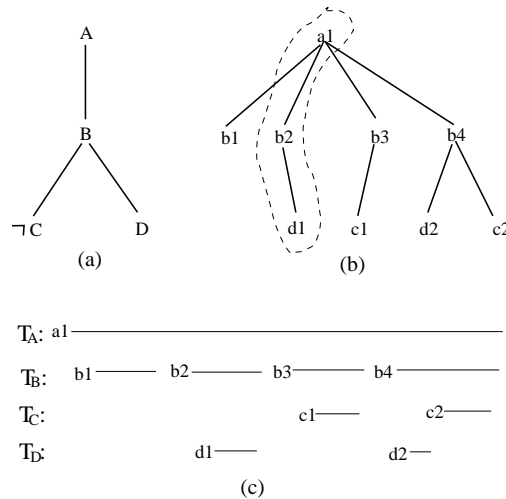


Figure 7: An AND/NOT-twig example with a sample data tree and the element streams

ALGORITHM TwigStackN(*root*)

```
/* This algorithm deals with AND-twigs that may contain NQNode leaves. */
/* Basic idea: for all QNodes, perform exactly the same as in the classic TwigStack algorithm */
/* For NQNode leaf  $q$ , let  $p$  denote the parent QNode and when  $C_p$  is found to contain  $C_q$ , */
/* pop out  $S_q$  and  $S_p$ ; when  $C_p$  is confirmed not to contain  $C_q$ , output  $top(S_p)$  if it */
/* has no QChildren. */

01 while not end(root) do
02    $q = getNext(root)$ ;
05   if isRoot( $q$ ) or not empty( $S_{parent(q)}$ )
06     cleanStack( $S_q, C_q$ );
03   if not isRoot( $q$ )
04     cleanStack( $S_{parent(q)}, C_q$ );
07   push( $S_q, C_q, isRoot(q)? -1 : top(S_{parent(q)})$ );
08   if isQLeaf( $q$ )
09     outputPathSolutions( $S_q$ );
10     pop( $S_q$ );
11   if isNQLeaf( $q$ )
12     disqualify( $S_{parent(q)}$ );
        /* this operation puts a mark on the output paths containing element  $top(S_{parent(q)})$  */
        /* so that they will be excluded from consideration by mergeAllPathSolutions( $\cdot$ ). */
13     pop( $S_q$ );
14     pop( $S_{parent(q)}$ );
15   else advance( $C_q$ );
16 end while
17 for all  $p \in subtreeQNodes(root)$ 
18   if not empty( $S_p$ ) and hasNQChild( $p$ )
19     outputPathSolutions( $S_p$ );
20 mergeAllPathSolutions();
```

Figure 8: Algorithm TwigStackN

8 System Study

In this section, we draw some conclusions from our experience with *Determined*. The experiments we conducted with *Determined* is in a client/serve environment with the following hardware and software settings: the client side runs Oracle SQL*Plus (Release 8.1.6.0.0) for Window XP Pro on a machine with a Celeron processor of 500MHZ and 192MB RAM; the server side runs Oracle8i Enterprise Edition (Release 8.1.6.0.0) on Sun Ultra 5 with an UltraSparc-II CPU of 333MHZ and 512MB RAM.

We used a synthesized dataset that consists of elements types simply named as A, B, C, etc. This dataset is basically the same as Database 2 that we utilized in our early study (Che et al. 2006). The size of the dataset is about 100 MB. In order to obtain a broad coverage of different query patterns, we adopted a set of randomly generated queries that have widely varied width and depth. From over a hundred of these randomly generated queries, we picked (almost randomly) five of them as samples to help the discussion in this section. The patterns of these sample queries are shown in Figure 10, while the ICs and available structure indices relevant to these queries are separately

FUNCTION getNext(q)

```

/* This sub routine returns either a QNode or an NQNode. */
01 if isLeaf(q) return q;
02 for all  $q_i \in \text{children}(q)$ 
03    $n_i = \text{getNext}(q_i)$ ;
04   if  $n_i \neq q_i$  return  $n_i$ ;
05  $n_{min} = \arg \min_{n_i} \{C_{n_i} \rightarrow \text{start}\}$ , for  $n_i$  initialized at line 3;
06  $n_{max} = \arg \max_{n_i} \{C_{n_i} \rightarrow \text{start}\}$ , for  $n_i$  initialized at line 3;
    /* NQNodes are excluded from the computation for  $n_{max}$ . */
07 while  $C_q \rightarrow \text{end} < C_{n_{max}} \rightarrow \text{start}$ 
08    $C_q \rightarrow \text{advance}()$ ;
09 if  $C_q \rightarrow \text{start} < C_{n_{min}}$  return q;
10 else return  $n_{min}$ ;

```

PROCEDURE cleanStack(S_p, C_q)

```

01 if hasNQChildren(p) and not hasQChildren(p)
02   flag = 0;
03   for all  $p_i \in \text{NQChildren}(p)$ 
04     if not empty( $S_{p_i}$ )
05       flag = 1; /* parent element is disqualified */
06   if flag == 0
07     outputPathSolutions( $S_q$ );
08 while not empty( $S_p$ ) and  $\text{top}(S_p).\text{end} < C_p \rightarrow \text{start}$ 
09   pop( $S_p$ );

```

Figure 9: Subroutines *getNext*(q) and *cleanStack*(S_p, C_q) of *TwigStackN*

annotated as follows:

Q1. ICs: $C = \{A \parallel_B D\}$, and exists structure index $I(A, D)$.

Q2. ICs: $C = \{A \parallel_B D, G \rightarrow H\}$, and exists structure index $I(A, D)$.

Q3. ICs: $C = \{A \parallel_B D, G \rightarrow H, B \rightarrow E, K \leq E\}$, and exist structure indices, $I(A, D)$ and $I(A, K)$.

Q4. ICs: $C = \{A \parallel_B D, M \leftarrow K, K \leq E\}$, and exist structure indices $I(A, D)$, $I(A, K)$, and $I(K, P)$.

Q5. ICs: $C = \{A \parallel_B D, G \rightarrow H, B \rightarrow E, K \leq E, M \leftarrow K\}$, and exist structure indices $I(A, D)$, $I(A, K)$, and $I(D, K)$.

Deterministic optimization perfectly fits the nature of XML queries

XML data has rich semantics related to its complex structure. Such structure semantics is a valued source of knowledge that can significantly facilitate heuristic-based, deterministic transformation on XML queries for optimization. The utmost virtue of deterministic transformation is its great potential for superior optimization efficiency, which has been proved by our experimental study. The optimization time of our tested queries mostly took less than a second, while the execution time of them decreased from minutes to seconds in many cases. Figure 11 shows the effects of minimization and optimization on the chosen sample queries.

We envision the following scenarios of applications of the two kernel modules, Query Minimizer and Query Op-

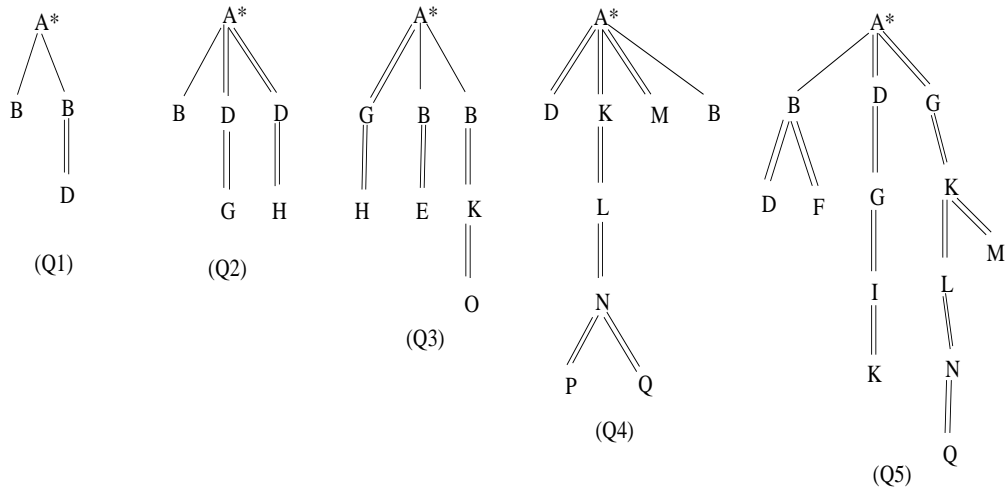


Figure 10: Sample twig query patterns: Q1, Q2, Q3, Q4, Q5

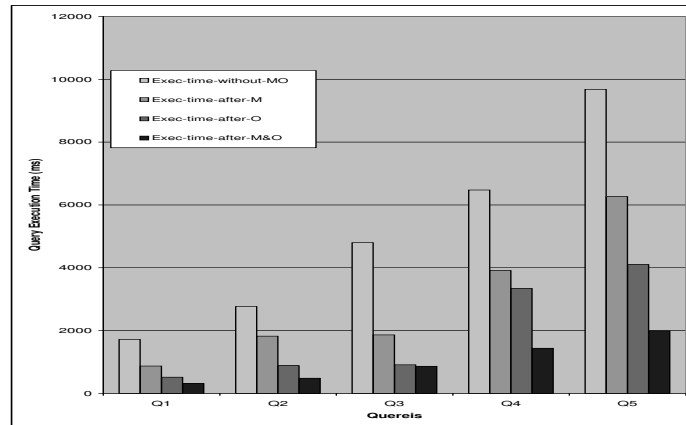


Figure 11: Effects of minimization and optimization on *query performance* (no-minimization&no-optimization vs. minimized-only vs. optimized-only vs. minimization&optimization)

timer in *Determined*: (1) for an XML data repository that does not yet have a query optimization module but desires a simple and fast one, our Query Minimizer module can alone be used; (2) for an XML data repository that would expect a query optimization module with moderate effectiveness and efficiency, our Query Optimizer module can be slotted in; (3) when there is a high requirement for optimization (e.g., for repeatedly run and complex XML queries), our Query Minimizer and Query Optimizer can both be exploited as a bundle.

In order to obtain further better optimization effect, sole logic-level optimization is not enough. Our next goal is to complement the logical optimizer with a compact physical (cost-based) optimizer. The physical optimizer must be a highly specialized one — *aware* of the optimization already performed at the logic level and proactively using this knowledge effectively pruning the search space of the (cost-based) physical optimizer.

Tree pattern minimization is an important aspect of XML query optimization

For an XML database, especially when involving complex and/or multiple DTDs/XSDs and with large sets of ICs, user queries typically contain redundancies (either syntactically or semantically). Sending the minimized form of a query to the subsequent modules for processing (e.g., logical/physical optimization and execution modules) has significance for the overall system performance. Figure 12 depicts the difference between the query optimization

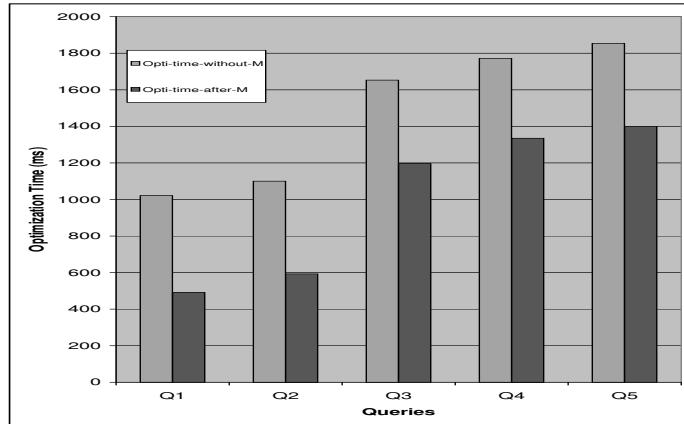


Figure 12: Effect of minimization on *optimization performance* (without minimization vs. with minimization)

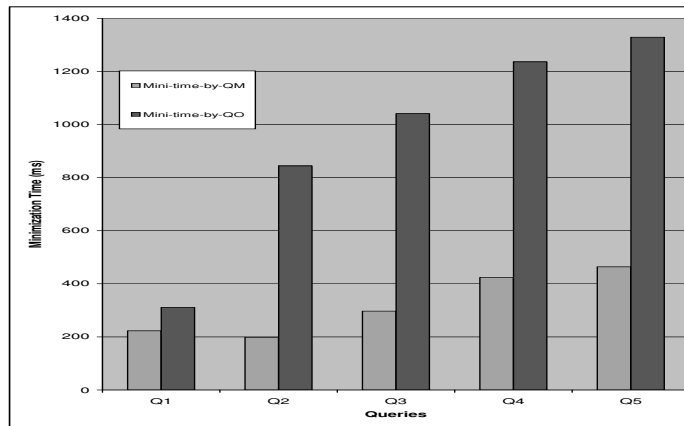


Figure 13: Minimization performance: hard-coded vs. rule-based minimization

without minimization and the query optimization with/after minimization on the five sample queries. The result indicates that the optimization of minimized queries can be several times faster than the non-minimized counterparts.

Hard-coded TPQ minimization outperforms rule-based approach

Rule-based approach has long been a competitor of hard-coded implementation for query optimization. To compare these two approaches with regard to XML query pattern minimization, we duplicated the function of the Query Minimizer module in the Query Optimizer module using a rule-based approach. The experimental result (Figure 13) indicates that hard-coded implementation is the winner: it outperforms the rule-based implementation by at least several times. This result indicates that we should retain the hard-coded Query Minimizer, and remove the functional duplication found in our rule-based Query Optimizer.

Specialized support for special cases of containments is worthwhile

Structural joins have proven their values for XML queries (Zhang et al. 2001, Al-Khalifa et al. 2002). Structural joins are proper for the situation when both operands are required to contribute to the final output. However, for certain special cases, e.g., when *pure* or *negated* containments are encountered, structural join algorithms is overkill and more costly than the specialized algorithms that we have presented in Section 7.

9 Summary

In this article, for the first time, we presented *Determined* as a complete system that integrates TPQ minimization and deterministic XML optimization and specialized support algorithms for XML queries into a single system. The two kernel modules, Query Minimizer and Query Optimizer, in *Determined* can be used either as a single package or as respective plug-ins in other XML repositories for accelerated XML query processing. After a systematic study and presentation of *Determined*, we drew important conclusions from a system perspective. We particularly pointed out the necessity for differentiating special containment operations from the more general structural joins and providing dedicated support algorithms for the special-case containments. We also presented a novel approach to holistic computation of all-twigs (with AND, OR, and NOT predicates) which are common for ordinary XML queries.

As for future work, we are planning to introduce physic-level optimization into the current framework of *Determined* to make our methodology complete and *Determined* a full-fledged query optimization system with the hope that *Determined* can be readily added/adapted to any XML storage system as a front-end.

References

- Al-Khalifa, S., Jagadish, H. V., Patel, J. M., Wu, Y., Koudas, N. & Srivastava, D. (2002), Structural joins: A primitive for efficient xml query pattern matching, *in* ‘Proc. of ICDE’, pp. 141–.
- Amer-Yahia, S., Cho, S., Lakshmanan, L. V. S. & Srivastava, D. (2001), Minimization of tree pattern queries, *in* ‘Proc. of SIGMOD Conference’, pp. 497–508.
- Balmin, A., Eliaz, T., Hornibrook, J., Lim, L., Lohman, G. M., Simmen, D. E., Wang, M. & Zhang, C. (2006), ‘Cost-based optimization in db2 xml’, *IBM Systems Journal* **45**(2), 299–320.
- Bruno, N., Koudas, N. & Srivastava, D. (2002), ‘Holistic twig joins: Optimal xml pattern matching’.
- Che, D. (2005), ‘Efficiently processing xml queries with support for negated containments’, *International Journal of Computer & Information Science* **6**(2), 119–120.
- Che, D. (2007), ‘An efficient algorithm for tree pattern query minimization under broad integrity constraints’, *International Journal of Web Information Systems* **3**(3), 231–256.
- Che, D., Aberer, K. & Özsu, M. T. (2006), ‘Query optimization in xml structured-document databases’, *VLDB J.* **15**(3), 263–289.
- Chen, T., Lu, J. & Ling, T. W. (2005), On boosting holism in xml twig pattern matching using structural indexing techniques, *in* ‘Proc. of SIGMOD Conference’, pp. 455–466.
- Chen, Y. & Che, D. (2006a), ‘Efficient processing of xml tree pattern queries’, *JACIII* **10**(5), 738–743.
- Chen, Y. & Che, D. (2006b), ‘Minimization of xml tree pattern queries in the presence of integrity constraints’, *JACIII* **10**(5), 744–751.
- Chung, T.-S. & Kim, H.-J. (2002), ‘Xml query processing using document type definitions’, *Journal of Systems and Software* **64**(3), 195–205.
- Fegaras, L., Dash, R. K. & Wang, Y. (2006), A fully pipelined xquery processor, *in* ‘Proc. of XIME-P’.
- Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M. J., Sundararajan, A. & Agrawal, G. (2003), The bea/xqrl streaming xquery processor, *in* ‘Proc. of VLDB’, pp. 997–1008.

- Florescu, D. & Kossmann, D. (1999), ‘Storing and querying xml data using an rdmb’, *IEEE Data Eng. Bull.* **22**(3), 27–34.
- Frasincar, F., Houben, G.-J. & Pau, C. (2002), Xal: An algebra for xml query optimization, in ‘Proc. of Australasian Database Conference’.
- Graefe, G. & DeWitt, D. J. (1987), The exodus optimizer generator, in ‘Proc. of SIGMOD Conference’, pp. 160–172.
- Hündling, J., Sievers, J. & Weske, M. (2005), Naxdb - realizing pipelined xquery processing in a native xml database system, in ‘Proc. of XIME-P’.
- Jagadish, H. V., Al-Khalifa, S., Chapman, A., Lakshmanan, L. V. S., Nierman, A., Pappas, S., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y. & Yu, C. (2002), ‘Timber: A native xml database’, *VLDB J.* **11**(4), 274–291.
- Jagadish, H. V., Lakshmanan, L. V. S., Srivastava, D. & Thompson, K. (2001), Tax: A tree algebra for xml, in ‘Proc. of DBPL’, pp. 149–164.
- Jiang, H., Lu, H. & Wang, W. (2004), Efficient processing of twig queries with or-predicates, in ‘Proc. of SIGMOD Conference’, pp. 59–70.
- Jiang, H., Wang, W., Lu, H. & Yu, J. X. (2003), Holistic twig joins on indexed xml documents, in ‘Proc. of VLDB’, pp. 273–284.
- Liu, Z. H., Krishnaprasad, M. & Arora, V. (2005), Native xquery processing in oracle xmldb, in ‘Proc. of SIGMOD Conference’, pp. 828–833.
- May, N., Helmer, S. & Moerkotte, G. (2003), Three cases for query decorrelation in xquery, in ‘Proc. of Xsym’, pp. 70–84.
- May, N., Helmer, S. & Moerkotte, G. (2004), Nested queries and quantifiers in an ordered context, in ‘Proc. of ICDE’, pp. 239–250.
- McHugh, J., Abiteboul, S., Goldman, R., Quass, D. & Widom, J. (1997), ‘Lore: A database management system for semistructured data’, *SIGMOD Record* **26**(3), 54–66.
- McHugh, J. & Widom, J. (1999), Query optimization for xml, in ‘Proc. of VLDB’, pp. 315–326.
- Ramanan, P. (2002), Efficient algorithms for minimizing tree pattern queries, in ‘Proc. of SIGMOD Conference’, pp. 299–309.
- Salminen, A. & Tompa, F. W. (1994), ‘Pat expressions: an algebra for text search’, *Acta Linguistica Hungarica* **41**(1), 277–306.
- Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. & Price, T. G. (1979), Access path selection in a relational database management system, in ‘Proc. of SIGMOD Conference’, pp. 23–34.
- Wang, G. & Liu, M. (2003), Query processing and optimization for regular path expressions, in ‘Proc. of CAiSE’, pp. 30–45.
- Wang, W., Jiang, H., Lu, H. & Yu, J. X. (2003), Containment join size estimation: Models and methods, in ‘Proc. of SIGMOD Conference’, pp. 145–156.
- Wang, W., Jiang, H., Lu, H. & Yu, J. X. (2004), Bloom histogram: Path selectivity estimation for xml data with updates, in ‘Proc. of VLDB’, pp. 240–251.

- Wu, Y., Patel, J. M. & Jagadish, H. V. (2003a), Structural join order selection for xml query optimization, *in* 'Proc. of ICDE', pp. 443–454.
- Wu, Y., Patel, J. M. & Jagadish, H. V. (2003b), 'Using histograms to estimate answer sizes for xml queries', *Inf. Syst.* **28**(1-2), 33–59.
- Yu, T., Ling, T. W. & Lu, J. (2006), Twigstacklist-: A holistic twig join algorithm for twig query with not-predicates on xml data, *in* 'Proc. of DASFAA', pp. 249–263.
- Zhang, C., Naughton, J. F., DeWitt, D. J., Luo, Q. & Lohman, G. M. (2001), On supporting containment queries in relational database management systems, *in* 'Proc. of SIGMOD Conference', pp. 425–436.
- Zhang, N., Özsu, M. T., Aboulnaga, A. & Ilyas, I. F. (2006), Xseed: Accurate and fast cardinality estimation for xpath queries, *in* 'Proc. of ICDE', p. 61.

Dunren Che is an assistant professor at the Department of Computer Science, Southern Illinois University Carbondale (SIUC). Before joining SIUC, he worked as a post-doc/researcher in several world-class institutes for numerous years. He received his PhD from the Beijing University of Aeronautics and Astronautics of China in 1994. He has now published more than 70 peer-reviewed research papers. His research interest has recently been focused very much on XML database technology, particularly XML query processing and optimization. More information about his teaching, research, and publications is available online at <http://www.cs.siu.edu/~dche/>

Wen-Chi Hou received the MS and PhD degrees in computer science and engineering from Case Western Reserve University, Cleveland Ohio, in 1985 and 1989, respectively. He is presently an associated professor of computer science at Southern Illinois University at Carbondale. His interests include statistical databases, mobile databases, XML databases, and data streams processing.