Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition

Klemens Böhm
Department of Informationsystems
ETH Zürich
8092 Zürich, Switzerland

Kathrin Gayer
Dep. of Economics and Computing Science
Humboldt University Berlin
10178 Berlin, Germany

Karl Aberer
GMD–IPSI
Dolivostr. 15
64293 Darmstadt, Germany

M. Tamer Özsu
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

Declarative access mechanisms for structured document collections and for semi-structured data are becoming increasingly important. In this article, using a rule-based approach for query optimization and applying it to such queries, we deploy knowledge on the Document Type Definition (DTD) to formulate transformation rules for query-algebra terms. Specifically, we look at rules that save navigation along paths by cutting off these paths or by replacing them with access operations to indices, i.e., materialized views on paths. We show for both cases that we correctly apply and completely exploit knowledge on the DTD, and we briefly discuss performance results.

1 Introduction

With the growing number of documents available via WWW and their increasingly larger size, WWW query languages are gaining importance. Many researchers currently investigate such languages, e.g., [2, 14]. The kind of documents they examine are structured documents, i.e., XML [18] and SGML [12] documents as well as HTML documents. XML as well as SGML make explicit the internal logical structure of documents and append arbitrary meta information with such logical document elements. In the context of this work, it is important that SGML documents must have a type, and their logical structure must conform to the definition of the respective document type, leading to a high degree of consistency of the document collection. WWW query languages allow querying the document collection based on the logical structure of documents and on hyperlinks between documents. In this article, we look at queries on the document structure and the retrieval of document elements. The objective of this article is to examine the degree to which knowledge on the Document Type Definition (DTD) is useful for query optimization. While in [2] the authors discuss rewriting techniques for new non-conventional query algebra operators, our techniques use the standard operator set. In [2], another optimization technique is the use of full-text index structures. In turn, this article combines the use of indices in combination with DTD information.

1.1 Motivating Example

For structured document collections, sophisticated (graphical) user interfaces are available. Using these interfaces, one can get not only an overview of the document collection but also of the logical structure of these documents. As an example, the Web interface to the MultiMedia Forum (MMF) – an interactive online journal published at GMD–IPSI [16] – displays the surnames of all authors. Clicking on a particular surname returns the respective documents. For instance, by clicking on the surname “Busse” the following query is evaluated by the query engine: “Find all documents in which the surname of an author contains the value ‘Busse’ ”. The execution of the above query may be very complex. It must be confirmed that the element types `AUTHOR` and `SURNAME` are indeed valid types for elements in MultiMedia Forum documents. Moreover, elements of type `SURNAME` have to contain the string “Busse” and must be descendants of `AUTHOR` elements. Finally, after locating `SURNAME` elements which make the above conditions, the engine has to return the corresponding documents.

If the MMF DTD reveales that either the type `AUTHOR` or the type `SURNAME` or both are not specified, we can infer that the query result is empty without looking at the database state at all. Moreover, if both types are valid, the DTD might show that elements of type `SURNAME` cannot be descendants of `AUTHOR` elements. If this is the case, we know in advance that the execution of this query will end without any document in its result set.

Since the designer of the MMF interface certainly kept track of the above inconsistencies, checking whether elements of type `SURNAME` are exclusively contained in `AUTHOR` elements might be more fruitful in this context. If the DTD certifies that any element of type `SURNAME` has to be contained in an `AUTHOR` element, we can reformulate the above query like follows: "Find all documents in which a surname contains the value 'Busse' ". The rewritten query saves the navigation between `SURNAME` and `AUTHOR` elements. However, if the DTD shows that `SURNAME` elements are not exclusively subelements of `AUTHOR` elements but also descendants, e.g., of `EDITOR` elements we gain a lot by accessing an index for the path (SURNAME,..., AUTHOR). Instead of navigating between `SURNAME` and `AUTHOR` elements, one access to the structure index (SURNAME, AUTHOR) returns all `SURNAME` elements containing the string "Busse" and being descendants of `AUTHOR` elements.

1.2 Scope

Our work in this article is based on the PAT algebra for structured documents. We have extended the original algebra described in [15] with additional operators to specify the existance of hyperlinks between target documents [4] so that it has the characteristics of a Web query language. Looking at the individual operators of the PAT algebra, we examine their role for DTD-based query optimization. It turns out that some PAT operators require special attention. These operators reflect that two element types of a particular DTD have *contains/contained* relationships.

The objective of this work is to use knowledge on the DTD for query optimization in order to identify more efficient evaluation strategies for queries posed against SGML repositories. The DTD can be seen as a set of integrity constraints that explicitely specify the arrangement of logical elements within documents. We identify relevant relationships between element types and show how these relations can be derived from the DTD and improve query evaluation time. We define the notions of 1) *exclusivity*, 2) *obligation*, and 3) *entrance location* to start the navigation between two elements of given types. With the first two notions, one does not have to navigate from an element to another to check for its existence since from the DTD is known that the target element must be there. If for two element types neither *obligation* nor *exclusivity* holds true, identifying a third *entrance location* to start the navigation with can as well improve query evaluation. Finally, the use of structure indices between two element types (i.e., materialized views on paths) in combination with the above notions can also avoid scanning the document base and improve query evaluation time.

Pursuing a rule-based approach to query optimization, we formulate rules as input for the query optimizer that reflect the notions sketched above, and prove the correctness of these rules.

Finally, we report on experiences gained with the implementation. We indicate which of the investigated ideas are in fact advantageous for query optimization.

1.3 Further Related Work

In addition to the work on Web query languages, work on query optimization based on database integrity constraints is also relevant [13, 9]. With regard to rule-based query optimization, one can distinguish between application-independent transformation rules and application-specific ones. The latter can be further classified into approaches where rules contain hardcoded application-specific knowledge and those where the application-specific knowledge is extracted from the database schema in the course of optimization. We pursue the second alternative.

We are aware of one piece of work that deals with the use of knowledge on the Document Type Definition to build application-specific optimization rules. In [6], Consens and Milo replace a query-algebra operator with a cheaper one whenever the DTD allows it. They consider a restricted set of DTDs and do not look at different grammar constructors. The optimization in [6] makes use of a special cost model, and the results are not directly transferable to our application scenario. In contrast, the contribution of our paper is independent of any cost model. Another issue is that Consens and Milo make use of the PAT index structures [10]. While most queries can be evaluated in an efficient way, the size of the PAT index depends on the size of the document collection, whereas the size of our index structure depends (roughly) on the number of relevant elements. In other words, the PAT index covers the whole document, but our index can be configured to contain only

```
<!ELEMENT AUTHORS    (AUTHOR*)>
<!ELEMENT AUTHOR     (NAME)>
<!ATTLIST AUTHOR     FUNCTION (AUTHOR | CAMERA | COMPOSER |
EDITOR | PHOTOGR | PRODUCER | REGISS | OTHERS ) "AUTHOR">
<!-- AUTHOR is the default value -->

<!ELEMENT EDITORS    (EDITOR*)>
<!ELEMENT EDITOR     (NAME)>
<!ELEMENT NAME       ((SURNAME,FIRSTNAME?), ADDITION?)>
<!ELEMENT (SURNAME | FIRSTNAME | ADDITION)   (#PCDATA)>
```

Figure 1: **Fragment of sample DTD (Document Type 'MultiMedia Forum')**

```
<AUTHORS><AUTHOR><NAME><SURNAME>Busse</SURNAME><FIRSTNAME>Ralf
</FIRSTNAME></NAME></AUTHOR><AUTHOR><NAME><SURNAME>Huck</SURNAME>
<FIRSTNAME>Gerald</FIRSTNAME></NAME></AUTHOR></AUTHORS>
```

Figure 2: **Fragment of sample SGML document**

the important structures. With an index that covers everything, updates are more difficult to implement.

The remainder of this article has the following structure: In the following section, we briefly describe the relevant notions of structured documents and declarative access to collections of such documents in our database application framework. Section 3 contains a description of our approach to DTD-based query optimization, together with correctness and completeness proofs. Section 4 extends this work by considering indices in combination with DTD knowledge. Section 5 contains further optimization measures. Section 6 concludes the paper and summarizes our implementation experiences.

2 Structured Documents and Declarative Access Mechanisms

SGML. The Standard Generalized Markup Language (SGML) is an ISO standard for document description. An SGML document is portable because its logical structure is described using a markup-based notation, as opposed to its layout structure. SGML allows the definition of arbitrary markup languages for documents of different types. A *Document Type Definition (DTD)* specifies such a markup language. From another perspective, the DTD specifies the valid types of document elements and their logical order within documents.

Figure 1 is a fragment of the MMF Document Type Definition [16]. Figure 2 is a fragment of an SGML document that conforms to this DTD. In this fragment, the `SURNAME` element with textual content 'Busse' *is directly contained in* an element of type `NAME`, and *indirectly contained in* elements of type `AUTHOR` and `AUTHORS`. Conversely, we say that these elements *directly or indirectly contain* the `SURNAME` element.

By definition, a *path in a document* is a list of elements such that the successor of an element in the list is the element's father in the hierarchical structure described by the DTD. E.g., `(SURNAME, NAME, AUTHOR)` is a path from the sample document in Figure 2. Such a path implies that the content model of element type `NAME` contains the element type `SURNAME`, while the content model of the element type `AUTHOR` contains the element type `NAME`. The content model of an element type may be defined like follows:

Definition 2.1 (Content Model) *A content model is a term of the following structure:*
c -> <element-type name> c_1,c_2 | c_1|c_2 | c_1&c_2 | c_1? | c_1* | c_1+ | (c_1)
where `<element-type name>` *means that the content is an element of the type identified by* `element-type name`. c_1* *stands for an arbitrary number of occurrences of* c_1. *That number may also be zero.* c_1? *means an optional occurrence of* c_1*;* $c_1|c_2$ *an occurrence of* c_1 *or one of* c_2. c_1,c_2 *indicates an occurrence of* c_1 *followed by one of* c_2. *Finally,* 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))*. Using SGML terminology, the comma is the sequence connector (SEQ) or a SEQ-node if the term is seen as a tree; '|' is the OR-connector (OR) or an OR-node; '?' is the optional occurrence indicator; '*' is the optional-and-repeatable occurrence indicator.*

Moreover, elements may have attributes, such as attribute `FUNCTION` for elements of type `AUTHOR` in Figure 1. Line '`<!ATTLIST AUTHOR FUNCTION (AUTHOR | CAMERA ...) ...>`' introduces this attribute. Attributes are of minor importance in this article.

Querying Structured Documents. We use the PAT algebra [15] to query document collections. This alegbra is particularly designed to query structured documents and it is independent of any underlying data model. Moreover, it is user-friendly and expressive. With our variant of the PAT algebra, the following grammar generates query terms:
E $\rightarrow$ <element-type name> | E_1 UNION E_2 | E_1 INTERSECT E_2 | CONTENT_SELECT (E, r) | ATTR_SELECT (E, A, r) | E_1 INCLUDS E_2 | E_1 INCL-IN E_2 | (E_1) | EMPTY

The term <element-type name> stands for the set of all elements of the respective type. `UNION`,

INTERSECT, and DIFF are set operators with the usual semantics. CONTENT_SELECT takes a set of elements and returns those whose textual content contains regular expression r. ATTR_SELECT takes a set of elements and returns those where attribute A contains regular expression r. INCLUDS and INCL-IN take two sets of elements E_1 and E_2 and return the set of elements E_1 INCL-IN $E_2 = \{e_1 \in E_1 \mid \exists e_2 \in E_2 \text{ s.t. } e_1 \text{ is contained in } e_2\}$
E_2 INCLUDS $E_1 = \{e_2 \in E_2 \mid \exists e_1 \in E_1 \text{ s.t. } e_2 \text{ contains } e_1\}$

We say that E_1 is the *internal element type*, whereas E_2 is the *external element type*. EMPTY stands for the empty set.

HyperStorM's Structured Document Database. To store and query structured documents, we have built a database application framework, the HyperStorM Structured Document Database [4], on top of the object-oriented DBMS VODAK [17]. With user perspective, each document element corresponds to a database object, i.e., the user perceives a document in the database as tree-like structure. However, for performance reasons, multiple document elements may be stored in one physical database object.

Furthermore, the database application offers indices for frequently asked document elements. This includes *attribute indices* which store attribute values of document elements. For instance, an index for the attribute FUNCTION of type AUTHOR contains for each document element the corresponding attribute value (e.g., AUTHOR, CAMERA, COMPOSER, etc.). Furthermore, *content indices* can be created which store the textual content of document elements of a particular type. For instance, a *content index* for the element type SURNAME stores the textual data of each corresponding element. Finally, the database application offers the *structure index*. For instance, a *structure index* of element type AUTHOR for the path (SURNAME, AUTHOR) stores all SURNAME elements from which a path (in upward direction) to an AUTHOR elements exists. The *structure index* is useful for evaluating queries with the INCLUDS or INCL-IN operator, and we will review it in Section 4.

Query Processing. PAT expressions posed against HyperStorM's Structured Document Database, are mapped into VQL queries. VQL is VODAK's OQL-like query language [5]. After their mapping to VQL, PAT expressions are processed like follows:

1. Parsing of VQL statement,
2. semantic check of query based on data model and database schema,
3. transformation of VQL query statement to query algebra expression
4. generation of alternative algebra expressions equivalent to the one generated in Step 3 using a set of transformation rules,
5. given the algebra expressions generated in Step 4, identifying the most cost-efficient one, based on a cost model, and
6. query evaluation.

This approach to query processing is similar to the ones described in [7] and [8]. The rules introduced in this article are transformation rules that are used in Step 4 of the above enumeration. Given a list of transformation rules, we use the Volcano Optimizer Generator [11] to generate a query optimizer. Rules may have a condition part that specifies under which conditions the optimizer may apply these rule. We will introduce our optimizations as rules in PAT notation rather than at the Volcano level[1]. This is because PAT expressions are much easier to read, and the PAT level better reflects the optimizations we are aiming.

3 Using Knowledge on the DTD for Query Optimization

In this article, we use knowledge on the DTD to identify more efficient evaluation strategies for queries posed against SGML repositories. In this section, we define the notions of *exclusivity*, *obligation* and *entrance locations* between element types. We then introduce the concept of *element-type graph* and indicate the relation to *exclusivity* and *entrance locations*. Finally, we describe operations on the DTD that allow to identify all cases of *obligation*.

3.1 Exclusivity, Obligation, and Entrance Location

Exclusivity. Examination of the DTD reveals that some types are shared among others. For example, element type NAME in Figure 1 is contained in element type AUTHOR and in EDITOR. But the types that are not shared, i.e., *exclusively contained in* others, bear potential for query optimization.

Definition 3.1 (Exclusivity) *Element type ET_j is exclusively contained in element type ET_i if each path $(e_j, \ldots, e_k)$ with e_j being an element of type ET_j and*

e_k *being the document root contains an element of type* ET_i. *Conversely, element type* ET_i *exclusively contains* ET_j *if the condition holds.*

Exclusivity serves as a condition for the following transformation rule:

Rule 3.1 *(E_1 INCL-IN E_2)* $\overset{c}{\Longleftrightarrow}$ *(E_1)*
c: (E_1 is exclusively contained in E_2)

If the condition c holds, an element e_1 of the internal type E_1 must be contained in an element e_2 of the external type E_2. Consequently, we can replace expression "E_1 `INCL-IN` E_2" with "E_1" as it yields the same result. Since the queries "E_1 `INCL-IN` E_2" and "E_2 `INCLUDS` E_1" are not equivalent, *exclusivity* is not appropriate to optimize queries such as the second one.

Obligation. Whereas *exclusivity* reflects "the perspective of the internal element type ", the external element type is the starting point with obligation.

Definition 3.2 (obligatorily contains/contained) *Element type* ET_i *obligatorily contains element type* ET_j *if each element of type* ET_i *has to contain in any document one element of type* ET_j. *Conversely, we say that* ET_j *is obligatorily contained in* ET_i.

The following transformation rule makes use of this definition.

Rule 3.2 *(E_1 INCLUDS E_2)* $\overset{c}{\Longleftrightarrow}$ *(E_1)*
c: (ET_1 obligatorily contains ET_2)

Analogous to the previous situation, we cannot use obligation to transform "E_1 `INCL-IN` E_2" to "E_1".

Entrance Locations. If two element types are not related by *exclusivity* and *obligation*, it may be worthwhile to check whether a third element type, called *entrance location*, exists before we start the navigation between the document elements.

Definition 3.3 (Entrance Location) *Element type A is an entrance location for element types B and C if in any document all paths from an element b of type B to element c of type C contain an element a of type A.*

Note that an *entrance location* for types `B` and `C` is not identical with an entrance location for `C` and `B`. The notion *entrance location* stems from experiments with the following rule.

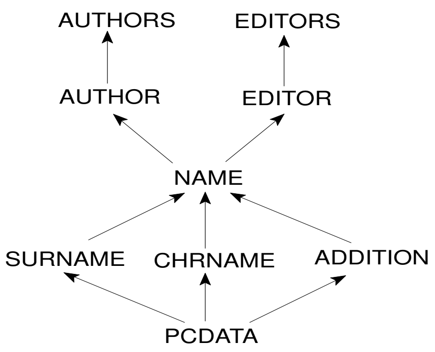

Figure 3: **Element-Type Graph for Sample DTD**

Rule 3.3 *(E_1 INCL-IN E_2)* $\overset{c}{\Longleftrightarrow}$ *(E_1 INCL-IN (E_3 INCL-IN E_2))*
c: (E_3 is entrance location for E_1 and E_2)

We define *entrance locations* for two reasons. The first one is that it is advantageous to begin navigation at instances of E_3 instead of E_1 or E_2 if the document base contains a much smaller number of instances of E_3, as compared to E_1 or E_2. But this situation occurs only in a few special cases. The second reason is that entrance locations may be advantageous in combination with *structure indices*. We will deal with such optimizations in Section 4.

3.2 Identifying all Cases of Exclusivity and Entrance Locations

We can visualize some of the relationships induced by a Document Type Definition by means of an *element-type graph*. Figure 3 contains the element-type graph for the fragment of the sample DTD in Figure 1. The following definitions introduce the semantics of nodes and edges.

Definition 3.4 (Element-Type Graph) *An element-type graph for DTD D is a directed graph* $G = (V, K)$. *Its vertices are the names of the element types from D. An edge* (ET_j, ET_i) *in K indicates that* ET_j *occurs in the content model of* ET_i. $RT \in V$ *is the root element-type of D.*

Definition 3.5 (directly contained/contains) *Element type* ET_j *is directly contained in element-type* ET_i *if there exists an edge* (ET_j, ET_i) *in G. Conversely,* ET_i *directly contains* ET_j *if the edge exists.*

Definition 3.6 (contained/contains) *The contained-in relationship (contains relationship) between element types is the transitive closure of* *the directly-contained-in relationship (the directly-contains relationship).*

An element-type graph depicts the contained-in relationships of a DTD. For example, one can see that element types `SURNAME` and `FIRSTNAME` are contained in element type `AUTHOR`. Furthermore, one can infer that element type `SURNAME` has content model `(#PCDATA)` from the fact that it does not contain any other element type.

Theorem 1 *Element type A is entrance location for B and C iff the element-type graph does not contain a path from B to C without vertex A.*

Proof. Suppose there is a path (B, E_1, E_2, ..., E_k, C) from B to C that does not contain vertex A. In this case, we can construct a document with a path from b to c that does not contain a. The content of c is chosen so that it contains an element e_k of type E_k, the content of e_k is chosen so that it contains an element e_{k-1} of type E_{k-1} etc. In the opposite direction, let A be an element type that is not an entrance location for B and C, i.e., there are elements b and c, and the path between them does not contain an a. If an element e_i of type E_i is directly contained in another one of type E_j, there is an edge from E_i to E_j in the element-type graph. ∎

Corollary 1 *An element type B is exclusively contained in element type A iff the element-type graph does not contain a path from B to the root-element type without vertex A.*

Proof. Follows immediately from Theorem 1 if one takes the root element type as C. ∎

3.3 Identifying all Cases of Obligation

To identify all cases of obligation, a deeper look at the occurence indicators (*, ?,+) in the content models is indispensible. Otherwise, we cannot distinguish whether an element requires or just optionally contains a subelement.

Definition 3.7 (Reduced Version of a DTD) *Let D be a DTD. By taking all content models and removing all subtrees whose root has an optional occurrence indicator (?) or an optional-and-repeatable occurrence indicator (*), we obtain a reduced DTD D'.*

Lemma 1 *The obligation relations of a DTD and its reduced version are identical.*

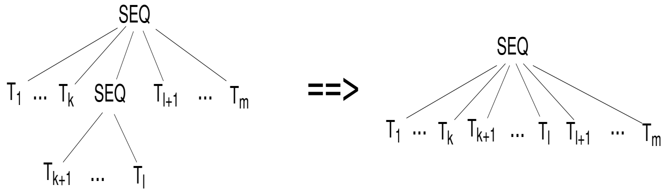

Figure 4: Normalization Step 1

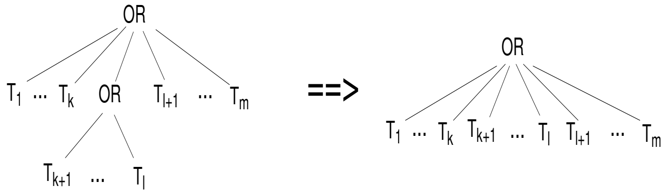

Figure 5: Normalization Step 2

Proof. The proof is by contradiction: Let element type B obligatorily contain element type A in D, but not in D'. If there is a document d of type D' with an element b that does not contain an a, this is a contradiction, since the subtree of d with root b also conforms to D. ∎

Lemma 2 *A reduced version of a content model can be normalized, i.e., transformed to another equivalent content model for which the following holds:*

- *the root is an OR-node,*
- *the children of the root are SEQ-nodes,*
- *the children of SEQ-nodes are leaves, i.e., element types.*

Proof. In the following, we define normalization steps. A sequence of these steps that transform a start tree into a result tree, i.e., a content model into another one, returns the normalized DTD. Three cases have to be considered. Figures 4, 5, and 6 reflect these cases with start trees on the left and target trees on the right. The transformation in Figure 6 requires further explanation: Consider an arbitrary list of nodes containing exactly one child of each OR-node. The order of the OR-nodes in the start tree implies the nodes' ordering in that list. Then there exists a SEQ-node in the target tree whose content is that list. Note that $T_{i,j}$ occurs once in the start tree, but various times in the target tree.

It is obvious that the depth of a content tree is always reduced by a finite sequence of steps. Hence,

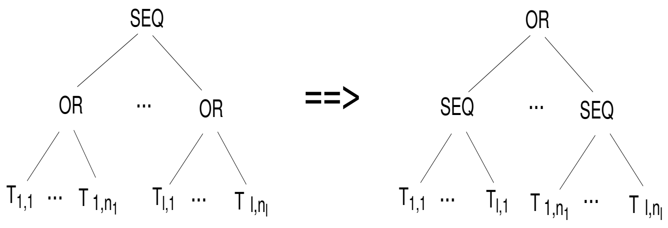

Figure 6: Normalization Step 3

the normalization process terminates. Furthermore, it is easy to conceive that the application of one of these steps does not alter the content represented by a content tree and does lead to normal form as defined above. ∎

In the context of this article, normalization is important. Consider the following definition of an element type `A`, in particular its content model:

```
<!ELEMENT A (C, (B | (B, D))) | (E, B))>
```

Each subtree of an OR connector is not obligatory, when seen in isolation. But careful examination of the content model yields that `B` is obligatorily contained in `A`. Furthermore, to identify all cases of obligation, it is not sufficient to look at individual content models in isolation. Let the following fragment of a DTD be given.

```
<!ELEMENT A (C | D)>
<!ELEMENT C (B)>
<!ELEMENT D (B)>
```

As `A` does not obligatorily contain `C`, one might be tempted to conclude that `A` likewise does not obligatorily contain `B`. But this is not the case.

Definition 3.8 (Extended Content Model) *Let A, B be element types of DTD D. The content model of A extended for B is the result of the following algorithm:*

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;

Theorem 2 *Element type A obligatorily contains element type B iff each SEQ-node in the content model of A that is first normalized and then extended for B contains an occurrence of B.*

Proof.
"⇐" (Correctness)
Given elements a and b of types A and B, respectively, extending the content model of A for B eliminates elements between a and b, but does not affect the fact that b is contained in a. Correctness follows from this observation and Lemmas 1 and 2.
"⇒" (Completeness)
Suppose that a SEQ-node in the content model of A extended for B does not contain an occurrence of B. Then we can construct a document fragment with an a that does not contain a b. ∎

4 Combining DTD Knowledge and the Structure Index

With the optimizations from Section 3, i.e., Rules 3.1 to 3.2, we avoid navigating along paths if we assume an object-oriented representation of structured documents. For query evaluation one still has to access the documents in the database. In this section we show that the combination of knowledge on the Document Type Definition and index structures has further potential for query optimization. The structure index is a materialized view on certain paths in the database. It accelerates the evaluation of queries that select elements (or database objects) contained in certain other elements. In this section, we discuss techniques that use DTD knowledge to transform queries so that the query processor can make use of structure indices for evaluation. These techniques find subpaths of the paths starting from the selected objects for which a structure index exists.

For illustrative purposes, consider the fragment of the MMF DTD in Figure 1. In the MMF DTD, element type `SURNAME` is exclusively contained in `NAME`. Element type `NAME` is shared among element types `EDITOR` and `AUTHOR`. In turn, other element types may exclusively contain these types, e.g., `AUTHORS` exclusively contains `AUTHOR`.

A structure index is a list of elements of a certain type that are contained in elements of another type. Assume that there is a structure index for (`SURNAME`, `AUTHOR`). It allows to quickly answer the query "`SURNAME INCL-IN AUTHOR`". To continue the above example, with this index we can infer from the DTD that these `SURNAME`-elements are exactly the ones contained in an `AUTHORS`-element, and another structure index for (`SURNAME`, `AUTHORS`) would be unnecessary. We are interested in identifying all cases where such overlappings of index structures occur. In more detail, we want to know the circumstances under which terms `(A INCL-IN B)` and `(A INCL-IN (C`

INCL-IN D)) are equivalent. To continue the above example, `A` and `C` correspond to `SURNAME`, `B` to `AUTHORS` and `D` to `AUTHOR`. Replacing '`(A INCL-IN B)`' with '`(A INCL-IN (C INCL-IN D))`' is of interest if there is a (`C`, `D`)-structure index. Then the second term is cheaper to evaluate because the path from elements of type `A` to the ones of `C` is shorter than to the ones of type `B`, and because in the database there may be less elements of type `C` within one of type `D` than elements of type `B`. If `A` and `C` are identical, we expect a performance gain by orders of magnitude, because one does not have to access the documents in the database at all, but can evaluate the query using only the structure index.

The first rule that exploits the structure index is as follows:

Rule 4.1

(A INCL-IN B) $\stackrel{c}{\Longleftrightarrow}$ (A INCL-IN (C INCL-IN D))
c: A is exclusively contained in C (a)
and B is exclusively contained in D (b)
and B is entrance location for C and D (c)

Theorem 3 *The query algebra terms on both sides of Rule 4.1 are equivalent, given that conditions (a), (b), (c) hold.*

Proof. The situation is depicted in Figure 7. Suppose the right expression would not return an element a of type A that is returned by the left expression. Because of (a), there is an element c of type C that contains a. Since a is not in the result of the right expression, expression `(C INCL-IN D)` does not identify c. In other words, c is not contained in an element of type D. From this, we can conclude that a is not contained in an element of type D. Namely, suppose there would be an element d of type D that would contain a. Then c, which contains a, would be contained in d or it would contain d. The first alternative is not feasible, as just explained. But c cannot contain d because then B could not be an entrance location for C and D. On the other hand, a is contained in an element b of type B, and b is contained in an element of type D, according to (b). In the opposite direction, let a be in the result of the right expression, but not the left one. In other words, a is not contained in an element of type B. But due to (a) a is contained in c of type C, which is contained in an element d of type D. But according to (c), there is an element b of type B between c and d. ∎

The transformation specified by Rule 4.1 requires conditions (a), (b) and (c). However, this does not

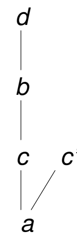

Figure 7: Illustration of Proof of Theorem 3

mean that the query optimizer should always transform the left term to the right one whenever (a), (b) and (c) are fulfilled. Rather, the transformation is advantageous only if a (C, D)-structure index exists. With our implementation, the condition part of the transformation rule checks this.

In the above example that illustrates Rule 4.1 there is a structure index for (`SURNAME`, `AUTHOR`). It is also conceivable that there is a (`NAME`, `AUTHOR`)-structure index, and the query is `SURNAME INCL-IN AUTHORS`. In this case, we can also use the structure index for query evaluation. Rule 4.2 in the sequence reflects this case. Finally we can modify the examples so that `AUTHORS` is the external element type of the structure index instead of `AUTHOR`, and `AUTHOR` is the external element type of the query instead of `AUTHORS`. We refrain from explicitly writing down the rules for these cases due to lack of space.

Rule 4.2

(C INCL-IN B) $\stackrel{c}{\Longleftrightarrow}$ (C INCLUDS (A INCL-IN D))
c: C obligatorily contains A (a)
and B is exclusively contained in D (b)
B is entrance location for C and D (c)

Theorem 4 *The query algebra terms on both sides of Rule 4.2 are equivalent, given that conditions (a), (b), (c) hold.*

Proof. Assume that c is in the result of the right term, but not the left one, meaning that there is no b containing c. Because of (a), there is a c containing a, and because of expression `(A INCL-IN D)` in the right term query a is contained in d. Because of (c) there is a b between c and d. In the opposite direction, (a) implies that c contains a. In consequence, because of the right query, there is no d that contains c. But there is a b that contains c, and because of (b) there is a d that contains b. ∎

5 Other Optimizations Based on the Document Type Definition

The objective of the optimizations described in Sections 3 and 4 has been the transformation of query algebra expressions so that transformation yields a superior query evaluation strategy. The optimizations described in this section eliminate subexpressions of query-algebra expressions that cannot have a solution. In many cases, it depends on the Document Type Definition whether or not a subexpression has a non-empty solution. For instance, the result of `A INCL-IN B` is empty if `A` or `B` are not element types of the respective DTD D or `B` does not contain `A` with regard to D. The respective rule is as follows:

Rule 5.1 *(X INCL-IN Y)* $\stackrel{c}{\Longrightarrow}$ *EMTPY*
c: (all instances of X are of type A, all instances of Y are of type B and A is not contained in B)

We can formulate rules to eliminate non-resolvable subexpressions such as Rule 5.1 for most PAT algebra operators, except `INTERSECT`. With regard to this operator, whether or not `X INTERSECT Y` is always empty does not depend on the DTD but on whether X and Y may contain elements of the same type. This can be checked without information from the DTD.

The `EMPTY`-operator is not used to formulate queries, but to have an expression to which the query optimizer can map query algebra terms that always return the empty set. In general, algebra expressions that contain an `EMPTY`-operator are themselves empty, except for the case that the `UNION`-operator is the root. Rules such as the following ones reflect this:

Rule 5.2 *(X INCL-IN EMPTY)* $\Longrightarrow$ *EMTPY*

Rule 5.3 *(EMPTY INCL-IN Y)* $\Longrightarrow$ *EMTPY*

These rules do not use DTD knowledge, they rather supplement rules that use DTD knowledge.

6 Conclusions

6.1 Classification of the Approaches by their Impact on Query Evaluation Time

In this subsection, we briefly report on experiences gained with our system by classifying our different approaches according to their impact on query evaluation time.

Improvements by orders of magnitude. With optimization based on structure indexing, as described in Section 4, there is an improvement by orders of magnitude if the query can be evaluated using access operations to index structures only. The techniques described in Section 5 also yield such an improvement: the duration of query evaluation is independent of the database size, as expected. In our test, optimization lasts approximately one second to avoid evaluation times of approximately half a second for each document in the database.

Fair improvements. *Obligation* and *exclusivity* to eliminate the `INCL-IN` operator or `INCLUDS` operator in queries of kind `A INCL-IN B` or `A INCLUDS B` yield a performance improvement of approximately 2 to 3. The improvement is fairly independent of the number of documents. Transformations that introduce the structure index, but do not completely eliminate access operations to the database improve performance by a similar factor. This improvement does not only result from the selectivity of the index, but may also be due to the fact that one does not have to navigate upwards from the internal elements of the index.

Not relevant as optimization technique. With regard to entrance locations, we have not encountered the case that an *entrance location* has significantly less instances than the other element types, which is a prerequisite for this optimization to work. Note that the notion of *entrance location* itself is not irrelevant, as it is used in connection with the structure index.

6.2 Summary and Future Issues

In this article, we have presented rules for optimization of queries on structured documents that make use of knowledge on the Document Type Definition. Most of these rules work in conjunction with the query-algebra operators `INCL-IN` or `INCLUDS`. This is in line with the fact that the *contains/contained* relationship is the universal structuring mechanism in structured documents, be it for raw data or for metadata, comparable to object references in OODBMSs. In Sections 3 to 5, respectively, the optimizations have been as follows:

- Cutting off redundant paths. In the context of structured documents, there are two path directions: the path of the external element and the path of the internal element. In the first case, optimization is for the `INCLUDS` operator and goes along with the notion of *obligation*, whereas in the second case optimization for the `INCL-IN` operator makes use of *exclusivity*.

- Identifying equivalent paths for which a structure index exists. A structure index has some similarities with path index structures in OODBMSs [3], but can also be seen as a materialized view on paths in structured documents. The respective transformation rules are only useful if a structure index indeed exists. In this context, we make use of the notion of *entrance locations*, which we have originally introduced to extend the search space for query evaluation alternatives.
- Eliminating query subexpressions for which a solution does not exist, according to the DTD.

We show that the chosen characteristics of document types can be completely identified using information from the Document Type Definition. To be able to conclude this, we introduce techniques to normalize and simplify the DTD without losing relevant information. In the case of *exclusivity* and *entrance locations*, we extract only the relevant information from the DTD by introducing notions such as element-type graphs.

In the future, we intend to build another optimizer that operates at the PAT level in order to directly compare the alternative approaches to query optimization, to find out about the advantages of a two-phase approach to query optimization and to investigate how to integrate the two optimization components.

References

[1] Karl Aberer and Gisela Fischer. Semantic Query Optimization for Methods in Object-Oriented Database Systems. In *Proceedings of International Conference on Data Engineering*, pages 70–79, 1995.

[2] S. Abiteboul et al. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1), 1997.

[3] E. Bertino and C. Guglielmina. Path-index: An Approach to the Efficient Execution of Object-Oriented Queries. *Data & Knowledge Engineering*, 10(1):1–28, 1993.

[4] Klemens Böhm, Karl Aberer, Erich J. Neuhold, and Xiaoya Yang. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. *VLDB Journal*, 6(4):296–311, 1997.

[5] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994.

[6] Mariano Consens and Tova Milo. Optimizing Queries on Files. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, volume 23, pages 301–312. ACM Press, May 1994. Minneapolis, Minnesota.

[7] D.D. Straube and M.T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems*, 8(4):387–430, 1990.

[8] J.C. Freytag. A Rule-Based View of Query Optimization. In *Proceedings ACM SIGMOD*, pages 173–180, 1987.

[9] Parke Godfrey, Jack Minker, and Lev Novik. An Architecture for a Cooperative Database System. In Witold Litwin and Tore Risch, editors, *Proceedings of First International Conference on Applications of Databases*, number 819 in Lecture Notes in Computer Science, pages 3–24. Springer-Verlag, June 1994.

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

[11] G. Graefe and W.J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the International Conference on Data Engineering*, pages 209–218. IEEE, 1993. Vienna, Austria.

[12] Information Technology - Text and Office Systems - Standardized Generalized Markup Language (SGML), 1986.

[13] J. King. Quist: A system for semantic query optimization in relational databases. In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 510–517, 1981.

[14] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1):54–67, 1997.

[15] A. Salminen and F.W. Tompa. PAT Expressions: an Algebra for Text Search. *Acta Linguistica Hungarica*, 41(1):277–306, 1994.

[16] Klaus Süllow et al. MultiMedia Forum - an Interactive Online Journal. In Christoph Hüser, Wiebke Möhr, and Vincent Quint, editors, *Proceedings of Conference on Electronic Publishing*, pages 413–422. John Wiley & Sons, Ltd., April 1994.

[17] VODAK V 4.0 User Manual. Technical Report 910, GMD-IPSI, April 1995. St. Augustin.

[18] XML. http://www.textuality.com/sgml-erb/WD-xml.html.