Querying XML Documents Made Easy: Nearest Concept Queries

Albrecht Schmidt Martin Kersten Menzo Windhouwer
CWI
Kruislaan 413
NL-1098 SJ Amsterdam
The Netherlands
firstname.lastname@cwi.nl

Abstract

Due to the ubiquity and popularity of XML, users often are in the following situation: they want to query XML documents which contain potentially interesting information but they are unaware of the mark-up structure that is used. For example, it is easy to guess the contents of an XML bibliography file whereas the mark-up depends on the methodological, cultural and personal background of the author(s). Nonetheless, it is this hierarchical structure that forms the basis of XML query languages.

In this paper we exploit the tree structure of XML documents to equip users with a powerful tool, the meet operator, that lets them query databases with whose content they are familiar, but without requiring knowledge of tags and hierarchies. Our approach is based on computing the lowest common ancestor of nodes in the XML syntax tree: e.g., given two strings, we are looking for nodes whose offspring contains these two strings. The novelty of this approach is that the result type is unknown at query formulation time and dependent on the database instance. If the two strings are an author's name and a year, mainly publications of the author in this year are returned. If the two strings are numbers the result mostly consists of publications that have the numbers as year or page numbers. Because the result type of a query is not specified by the user we refer to the lowest common ancestor as nearest concept.

We also present a running example taken from the bibliography domain, and demonstrate that the operator can be implemented efficiently.

1. Introduction

Over the past year, XML has been converging towards the role of the standard data representation format in many World Wide Web applications. XML takes the idea of mark-up further than HTML: it is not used for visual representation of data, but for encoding semantics in documents which makes not only a document's character data but also the tags and the way they are nested an interesting target for query languages. In contrast to other hierarchical data-models (see [2]) like complex data models or the object-oriented models, XML is an incarnation of the semistructured paradigm, which means that the database schema that results from the mapping of a document to a database instance tends to be large and irregular. It may not be immediately clear which parts of the database obey which part of the schema. All this hinders *ad hoc* users and non domain experts in posing meaningful queries, as state-of-the-art query languages do not fully capture the loose schema of many XML data.

The database community realized the demand for additional query formulation aids and proposed regular path expressions [3, 11] to allay the problem. Query languages like XML-QL [10], Lorel [3], XQL [18] or Quilt [9] and others (see [7] for a comparative analysis) all support some flavor of schema wildcards and, thus, relieve the user of the burden of having to specify the complete paths to the data. The commonest way to accomplish this is to allow for specifying sets of paths with UNIX command line-like regular expressions that are evaluated against the actual database. However there are cases when regular expression do not provide the power necessary to get the intended results. Consider the following situation taken from the area of bibliographic databases: A user wants to know what 'Ben Bit' edited or published in '1999', *i.e.*, find the relevant publication record(s) in an XML bibliography, but hasn't got any knowledge of the schema of the the XML file sketched in Figure 1. Therefore the user may try the following query[1]:

[1] Due to the lack of a standard query language for XML we use a variant of SQL enriched with paths and path variables (see [19]). In paths $\xrightarrow{e}$ de-

select $\$t$
from $* \xrightarrow{e} \$t\ \ n,$
$n \xrightarrow{e} * \xrightarrow{e} cdata \xrightarrow{a} string\ \ o_1,$
$n \xrightarrow{e} * \xrightarrow{e} cdata \xrightarrow{a} string\ \ o_2,$
where o_1 **contains** 'Bit'
and o_2 **contains** '1999'

The query binds $\$t$ to the tag names of all nodes whose offspring contains as character data the string 'Bit' and, respectively, '1999'. Evaluated against the example document shown in Figure 1 the answer looks like:

```
<answer>
  <result> article </result>        (o3)
  <result> institute </result>      (o2)
  <result> bibliography </result>   (o1)
  <result> bibliography </result>   (o1)
</answer>
```

Although the answer contains the desired result, it suffers from a serious drawback: we are only interested in a subset of the answers the database generates. Some not so interesting answer elements are implied by the path from the first node that is bound to $\$t$, to the root node: they are ancestor nodes of this first node (*e.g.*, the `institute` and the first `bibliography` elements in the answer set are implied by the `article` element). Even worse, in larger databases the computation might cause a combinatorial explosion of the result size.

One solution to the problem is to refine the query. In general, this involves a fair amount of domain knowledge that cannot be expected of *ad hoc* users. Therefore, we take another path and define a special operator, the *meet* operator, which gives the user more control over the results generated by such queries. For two nodes in the syntax tree o_1 and o_2 the meet operator $meet(o_1, o_2)$ simply returns the lowest ancestor of nodes o_1 and o_2, which we call the *nearest concept* of o_1 and o_2 to indicate that the type, *i.e.*, tag, of the result is not specified by the user. Informally, this node implies all other possible answers. By suitably extending this operator to work on sets of nodes and adding it as a declarative construct to our query language we give the user an opportunity for explorative querying even if he or she has only little or no knowledge of the database schema and content. As [1, 15] point out, there is always the notion of a schema in semi-structured or XML databases, but it may be large, unknown or implicit and therefore opaque to the user.

While the semantics of the operator for two objects are intuitive, it is less clear what happens if there are more than two nodes. This is the case if it is applied to the result of a full-text search. If we apply the original motivation to such an input we will end up with a combinatorial explosion of the result size. Therefore we will also present a generalization of the operator that is tailored towards large amounts of nodes: it delivers both intuitive results and has an efficient execution model.

The structure of this paper is as follows: Section 2 introduces the conceptual and physical data model used in this paper. Section 3 formalizes the notion of meet for various inputs and also presents algorithms. Section 4 expands on these ideas. Then we assess the performance of the algorithms presented and conclude with a review of related work and plans for future work.

2. Conceptual and Physical Data Model

XML documents are normally viewed from two perspectives: a conceptual and a physical one. While the conceptual perspective provides a convenient model for the end user to formulate queries, the physical model is geared towards efficient execution. The conceptual and physical models we present allow for straight-forward and intuitive mappings between one-another and form the basis for the ideas presented in later sections. A more detailed discussion of the models with a performance analysis can be found in [19].

XML documents are commonly represented as syntax trees. With **string** and **int** denoting sets of character strings and integers and **oid** being the set of unique object identifiers (OIDs), we can define a XML document formally (*e.g.*, see [23]):

Definition 1. An *XML document* is a rooted tree $d = (V, E, r, \mathbf{label}_E, \mathbf{label}_A, rank)$ with nodes V and edges $E \subseteq V \times V$ and a distinguished node $r \in V$, the root node. The function $\mathbf{label}_E : V \rightarrow \mathbf{string}$ assigns labels to nodes, *i.e.*, elements; $\mathbf{label}_A : V \rightarrow \mathbf{string} \rightarrow \mathbf{string}$ assigns pairs of strings, attributes and their values, to nodes. Character Data (CDATA) are modeled as a special attribute of *cdata* nodes, $rank : V \rightarrow \mathbf{int}$ establishes a ranking to allow for an order among sibling nodes.

The example document in Figure 1 adheres to this data model: element relationships are displayed as straight lines, attribute relationships as labeled arcs. The other representation details are largely self-explanatory; the assignment of OIDs is arbitrary, *e.g.*, depth-first traversal order. We apply the common simplification not to differentiate between PCDATA and CDATA nor do we take rich datatypes into account.

Before we discuss techniques how to store a syntax graph as a database instance, we introduce the notion of *association*. Associations are a binary modeling construct that allows a storage schema where related information is semantically clustered in one relation. This implies that our

notes an element relationship, $\xrightarrow{a}$ and attribute relationship; $*$ is a schema wildcard and may stand for any sequence of tags.

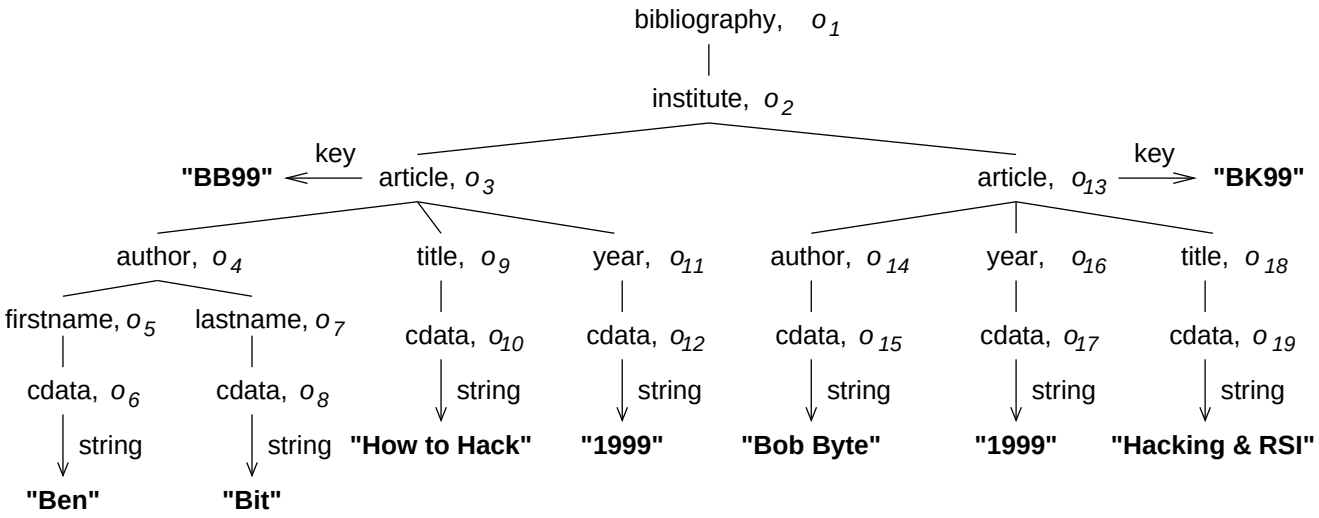

Figure 1. Syntax tree of example document

model is primarily aimed at *associative* retrieval of XML documents as opposed to navigating retrieval. Associations are the basis of the storage schema that is introduced later.

Definition 2. A pair $(o, \cdot) \in \mathbf{oid} \times (\mathbf{oid} \cup \mathbf{int} \cup \mathbf{string})$ is called an *association*.

The different types of associations describe different parts of the tree: associations of type $\mathbf{oid} \times \mathbf{oid}$ represent edges, *i.e.*, parent-child relationships. Both kinds of leaves, attribute values and character data are modeled by associations of type $\mathbf{oid} \times \mathbf{string}$, while associations of type $\mathbf{oid} \times \mathbf{int}$ are used to preserve the topology of a document.

Definition 3. For an item (string or OID labeled node) o in the syntax tree, we denote the sequence of labels along the path (vertex *and* edge labels) from the root to o with $\wp(o)$.

As an example consider the node with OID o_3 in Figure 1; *e.g.*, $\wp(o_3) = bibliography \xrightarrow{e} institute \xrightarrow{e} article$. We use $\wp(o)$ to describe the position of the element in the graph relative to the root node in terms of the overall schema; it plays a similar role as the type or class in object systems and, therefore, we use $\wp(o)$ to denote the *type* of the association $(\cdot, o)$. The set of all paths in a document is called its *path summary*.

In the rest of the paper, we adhere to the conventional view to identify nodes in the syntax tree with the OIDs assigned to them. However, OIDs by themselves do not indicate in which relation the associations that describe the node are stored. For a given node with OID o we assume that we can derive $\wp(o)$ given an OID o. For a justification see [8] who give an overview of similar techniques in object-oriented and object-relational databases.

We now show how to map the conceptual data model to a physical database instance. The general idea is to store all associations of the same type in one *binary relation*. A relation that contains the tuple $(\cdot, o)$ is named $\wp(o)$, conversely a tuple is stored in exactly one relation.

Definition 4. Given an XML document d, the *Monet transform*[2] is a quadruple $M_t(d) = (r, \mathbf{E}, \mathbf{A}, \mathbf{T})$ where

$$\mathbf{E} = \bigcup_{(o_i, o_j, s) \in \tilde{E}} [\wp(o_i) \xrightarrow{e} s]\langle o_i, o_j \rangle,$$

$$\mathbf{A} = \bigcup_{(o_i, s_1, s_2) \in label_A} [\wp(o_i) \xrightarrow{a} s_1]\langle o_i, s_2 \rangle,$$

$$\mathbf{T} = \bigcup_{(o_i, i) \in rank} [\wp(o_i) \rightarrow rank]\langle o_i, i \rangle$$

r remains the root of the document.

In the preceding definition E and $label_E$ are combined into one set

$$\tilde{E} = \{(o_1, o_2, s) | (o_1, o_2) \in E, s = label_E(o_2)\},$$

$label_A$ is interpreted as a set $\subseteq \mathbf{oid} \times \mathbf{string} \times \mathbf{string}$ as well as $rank \subseteq \mathbf{oid} \times \mathbf{int}$, and $[expr]$ denotes that the value of $expr$ is a relation name. Figure 2 displays the Monet transform of the example document.

Note we can easily switch from the relational perspective of the Monet transform to a convenient object-oriented view, *i.e.*, nodes in the syntax tree seen as objects [22]: we 're-assemble' an object with OID

[2] named after our implementation platform Monet [21]

$$bibliography \xrightarrow{e} institute = \{\langle o_1, o_2\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article = \{\langle o_2, o_3\rangle, \langle o_2, o_{13}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{a} key = \{\langle o_3, \text{“BB99”}\rangle, \langle o_{13}, \text{“BK99”}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author = \{\langle o_3, o_4\rangle, \langle o_{13}, o_{14}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} cdata = \{\langle o_{14}, o_{15}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} cdata \xrightarrow{a} string = \{\langle o_{15}, \text{“Bob Byte”}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} firstname = \{\langle o_4, o_5\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} firstname \xrightarrow{e} cdata = \{\langle o_5, o_6\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} firstname \xrightarrow{e} cdata \xrightarrow{a} string = \{\langle o_6, \text{“Ben”}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} lastname = \{\langle o_4, o_7\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} lastname \xrightarrow{e} cdata = \{\langle o_7, o_8\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} author \xrightarrow{e} lastname \xrightarrow{e} cdata \xrightarrow{a} string = \{\langle o_8, \text{“Bit”}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} title = \{\langle o_3, o_9\rangle, \langle o_{13}, o_{18}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} title \xrightarrow{e} cdata = \{\langle o_9, o_{10}, \langle o_{18}, o_{19}\rangle\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} title \xrightarrow{e} cdata \xrightarrow{a} string = \{\langle o_{10}, \text{“How to Hack”}\rangle, \langle o_{19}, \text{“Hacking \& RSI”}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} year = \{\langle o_3, o_{11}\rangle, \langle o_{13}, o_{16}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} year \xrightarrow{e} cdata = \{\langle o_{11}, o_{12}\rangle, \langle o_{16}, o_{17}\rangle\},$$
$$bibliography \xrightarrow{e} institute \xrightarrow{e} article \xrightarrow{e} year \xrightarrow{e} cdata \xrightarrow{a} string = \{\langle o_{12}, \text{“1999”}\rangle, \langle o_{17}, \text{“1999”}\rangle\}$$

Figure 2. Monet transform of the example document

o from those associations whose first component is o. Therefore, it is intuitive to identify an object by its OID; for example, the object $object(o_3) = \{key\langle o_3, \text{“BB99”}\rangle, author\langle o_3, o_9\rangle, title\langle o_3, o_1 1\rangle\}$ is easily converted into an instance of a suitably defined class *article* with members *key*, *author* and *title* or an instance of a DOM tree. Therefore an object can be regarded as a set of associations.

3. Nearest Concept Search

We now formalize the semantics of the meet operator in terms of the data model of the previous section. We start from the simple case of finding the meet, denoted $meet_P$, of a pair of nodes to the more sophisticated case of applying the meet to a set of objects such as the results of a full-text search.

3.1 The Meet-Operator

To simplify the discussion, we abstract from the example query given in the introduction for the time being and limit ourselves to the basic question: Given two nodes in the syntax tree o_1 and o_2, how can we calculate $meet_P(o_1, o_2)$. Later, we come back to the initial question and extend on it.

We now formalize and generalize the ideas sketched in the introductory example. First, we borrow some notation to denote offspring relationships in the schema and in the database instance.

Definition 5. We write $path(o_1) \leq path(o_2)$ if $path(o_2)$ is a prefix of $path(o_1)$ (including $path(o_1) = path(o_2)$). Analogously, $\wp(o_1) \leq \wp(o_2)$ if $\wp(o_2)$ is a prefix of $\wp(o_1)$ (including $\wp(o_1) = \wp(o_2)$).

The difference between $path(o)$ and $\wp(o)$ is that the latter only provides schema information whereas the former includes parts of the actual database instance; another dissimilarity is that in a given association $\langle o, \cdot\rangle$, $\wp(o)$ comes for free by looking at the name of the relation; on the other hand, to derive $path(o)$ in general requires joins to be computed. For example, $path(o_3) = (bibliography, o_1) \xrightarrow{e} (institute, o_2) \xrightarrow{e} (article, o_3)$. We now use $path$ to interrelate any two objects in a document tree:

Definition 6. Let o_1, o_2 and o_3 be objects in an XML syntax tree. Then $o_3 = meet_P(o_1, o_2)$ iff

1. $path(o_1) \leq path(o_3)$,
2. $path(o_2) \leq path(o_3)$ and
3. $\nexists o_4 : path(o_4) \leq path(o_3) \wedge path(o_1) \leq path(o_4) \wedge path(o_2) \leq path(o_4)$.

Note that $meet_P$ does not depend on the order of its arguments. Eventually, we identify the following semantics with the $meet_P$: The *nearest concept* of objects o_1 and o_2 is $\wp(meet_P(o_1, o_2))$.

Examples. Suppose we do a full-text search for "Ben" and "Bit" on the example document. The resulting associations are $a_1 = A\langle o_6, \text{"Ben"}\rangle$ and $a_2 = B\langle o_8, \text{"Bit"}\rangle$ (we abbreviate the relation names with A and B; the full names are easily recovered by looking them up in Figure 2 or Figure 1). After calculating $meet_P(a_1, a_2) = o_4$ we find that the two associations constitute an author's name.

A full-text search for "Bob" and "Byte" returns the associations $a_1 = A\langle o_{15}, \text{"Bob Byte"}\rangle$ and $a_2 = A\langle o_{15}, \text{"Bob Byte"}\rangle$. In this case $meet_P(a_1, a_2) = o_{15}$, which is a cdata node. Fortunately, the hierarchical information included in the Monet XML model immediately exhibits that the cdata node is a son of an author node.

When searching for "Bit" and "1999" the full-text search returns the associations $a_1 = A\langle o_8, \text{"Bit"}\rangle$, $a_2 = B\langle o_{12}, \text{"1999"}\rangle$ and $a_3 = B\langle o_{17}, \text{"1999"}\rangle$. Similarly, $meet_P(a_1, a_2) = o_3$ reveals that Mr "Bit" published an article in "1999"; however, $meet(a_1, meet(a_2, a_3)) = o_2$ only reveals that the three associations are located in the bibliography of an institute. We therefore will discuss variants of the meet operator to produce more intuitive results and filter out trivial or counter-intuitive ones.

We now consider a variety of interpretations of $o = meet_P(o_1, o_2)$. These possible views make the meet a useful construct in many different application domains. The following enumeration deals with two argument objects only, but the reasoning extends to a larger set of objects as well.

- $path(meet_P(o_1, o_2))$ is the longest common prefix of $path(o_1)$ and $path(o_2)$.
- $path(o_1) - path(o)$ and $path(o_2) - path(o)$ describe the context of o_1 and o_2 with respect to o. Depending on the overall schema, this may describe a *part of* or *is a* relationship or a sequence thereof. (For two paths p_1 and p_2, p_1 prefix of p_2, $p_2 - p_1$ denotes the elements of p_2 that are not included in p_1.)
- $path(o_1) - path(o)$ and $path(o_2) - path(o)$ describe the different contexts we see while traversing from o_1 to o_2 or vice versa. Trivially, this is also the shortest path from o_1 to o_2.
- We can also interpret the $\wp(meet(o_1, o_2)$ as the smallest enclosing context of the input objects.
- Finally, $meet_P(o_1, o_2)$ is the first node on $path(o_1)$ and $path(o_2)$ that contains both o_1 and o_2, *i.e.*, the nearest concept of both nodes.

```
function meet_P (oid o_1, oid o_2) : oid
  if o_1 = o_2 then return o_1
  else
    case
      ℘(o_1) ≤ ℘(o_2) : return meet_P(parent(o_1), o_2)
      ℘(o_2) ≤ ℘(o_1) : return meet_P(o_1, parent(o_2))
      default : return meet_P(parent(o_1), parent(o_2))
    end
  end
end
```

Figure 3. Function $\mathbf{meet}_P$ for a pair of OIDs

3.2 Computation

In this section we present the fundamental algorithms to compute $meet_P$ and two generalizations. Note that the algorithms in this section take advantage of the physical data model introduced earlier. The prefix order among the paths is used to steer the search for the lowest common ancestor so that superfluous look-ups are avoided.

The algorithm displayed in Figure 3 computes $meet_P(o_1, o_2)$ for two objects and will be used as a building block for more general cases. The function $parent(o)$ returns the parent association of the node or association o, basically a hash look-up. A remark on the **case** clause: by comparing $\wp(o_1)$ and $\wp(o_2)$ we are able to find the meet of these two objects as fast as possible as the comparison steers the search direction of the algorithm and avoids superfluous look-ups. As pointed out in [19] this information is provided with only little additional cost at bulk load time.

The previous algorithm operated on two object identifiers. The next step we take is to generalize $\mathbf{meet}_P$ to work with sets of OIDs O_1 and O_2 where all associations in O_i are of the same type, *i.e.*, there is a path p in the path summary that $\forall o \in O_i : \wp(o) = p$. With this set-up, we may generalize the previous algorithm to what is displayed in Figure 4.

This time, the function $parent(O_1, O_2)$ is a shortcut for $join(O_1, O_2)$, a binary join on associations $A_1\langle o_1, o_2\rangle$ and $A_2\langle o_2, o_3\rangle$ so that $join(A_1\langle o_1, o_2\rangle, A_2\langle o_2, o_3\rangle) = A\langle o_1, o_3\rangle$ (the inner columns are projected out, leaving a binary relation – association in our terminology). Note that we avoid a combinatoric explosion of the result size as $\mathbf{meet}_S$ computes minimal meets, *i.e.*, as soon as the first meet of $o_1, o_2, \cdots \in O_1 \cup O_2$ is found subsequent meets are not considered anymore because the elements are removed from the input sets. This generalizes the minimality criterion (3) of Definition 6 to sets of objects while still being invariant of the input order. Also note that we slightly extended the definition of meet: we now call a node meet if it is the lowest common ancestor of *at least* two other nodes. A salient feature of this and the following algorithm is that

```
procedure meet_S (OID O_1, OID O_2) : OID
  for i = 1 to 2
    r := {o | ⋀_{j=1,...,n;n≥2} O_i⟨o, o_j⟩}
    add to result r
    O_i := O_i − r
  end
  if O_1 = ∅ or O_2 = ∅ then return
  I := O_1 ∩ O_2
  if I ≠ ∅ then add to result I
  O_1 := O_1 − I
  O_2 := O_2 − I
  case
    ℘(O_1) ≤ ℘(O_2) : add to result meet_S(parent(O_1), O_2)
    ℘(O_2) ≤ ℘(O_1) : add to result meet_S(O_1, parent(O_2))
    default : add to result meet_S(parent(O_1), parent(O_2))
  end
end
```

Figure 4. Procedure meet_S for two sets of OIDs

they make heavy use of the relational operations of the underlying database engine. In the analysis we will see they indeed perform favorably.

We now present the most general algorithm of this paper: it calculates the meet of an arbitrary input set of nodes grouped into relations $r_1, \ldots, r_n$ according to the type of association they represent. This approach proves useful when we want to combine the results of full-text queries, which may be distributed over a large number of relation, *i.e.*, we extract from the results of the full-text query starting points from where the user can start displaying and browsing the database. The algorithm is displayed in Figure 5.

In contrast to the previous algorithm, we cannot simply exploit the function $\leq$ to compare the paths to steer the search, because then the algorithm would become dependent on the input order, as the algorithm does not know which subtrees of the document instance are being searched at a particular moment. Therefore, we rather roll up the tree-shaped schema from the bottom by iteratively contracting the offspring of nodes whose only offspring are leaves until we reach the root or the empty set. This way, all nodes that are meets of other nodes are minimal by construction; they are output and not considered anymore, thus, avoiding a combinatorial explosion of the result set and dependence on the input order.

Coming back to the example query, we see that after reformulating the query with the meet operator the cardinality of the answer set reduces (from now on, we interpret the meet operator as an aggregation operation):

select $meet(o_1, o_2)$
from $* \xrightarrow{e} cdata \xrightarrow{a} string\ o_1, * \xrightarrow{e} cdata \xrightarrow{a} string\ o_2$
where o_1 **contains** 'Bit'
and o_2 **contains** '1999'

```
procedure meet ({r_1, ..., r_n}) : OID
  if (n = 0) or (n = 1 and |r_1| = 1) then return
  for i = 1 to n
    r := {o | ⋀_{j=1,...,o;o≥2} r_i⟨o, o_j⟩}
    r_i := r_i − r
    add to result r
  end
  let n be a relation all of whose children are leaves
  w. l. o. g. let r_1, ..., r_l (n ≥ l ≥ 1) be the children of n
  (r_1, ..., r_l) := (parent(r_1), ..., parent(r_l))
  p := r_1
  meets := ∅
  for i = 2 to l
    hits := p ∩ r_i
    r_i := semijoin(r_i, hits)
    meets := meets ∪ hits
    p := p ∪ (r_i − hits)
  end
  add to result hits
  remove empty r_i
  meet(r_1, ..., r_m)
end
```

Figure 5. Procedure meet for arbitrary sets of objects

Evaluated against the example document we now obtain the following result, a true subset of what the solution presented in the introduction with regular path expressions returned:

```
<answer>
  <result> article </result>                    (o_3)
</answer>
```

The generated answer now resembles our initial intuition. With some domain-knowledge (gained by looking at a visualization of the answer) the user can interpret the result as follows: Mr. Bit wrote an article in 1999.

XML documents may also contain references (IDs and IDREFs) that potentially break the tree structure defined by the element relationships. The algorithms we presented only cover element relationships as we believe that they often carry very natural semantics and because the design of the meet algorithms remains clear and intuitive while execution times enable interactive querying. If we interpret the meet operator as some variant of nearest neighbor search, we might find generalizations on graph structures that prove useful in certain application domains. However, the fact that we then have to take care of circular structures may add significant complexity to our algorithms.

Finally, we remark that the meet operator is not expressible in the relational algebra: We need stratified datalog$^\neg$ [2] to calculate it.

4. Extensions and Applications

In large databases our algorithms may still deliver too many unintuitive results. In this section we propose variations of the meet operator to gain more control over what the operator returns. In particular, we propose to extend the meet operator with two parameters: (1) a maximum distance that says how many edges may lie between two input objects, and (2) restrictions of the type of results, *i.e.*, if o is a result candidate we restrict $\wp(o)$ to a certain set of paths R; if $\wp(o) \in R$ we discard o:

$$meet_R(r_1, \ldots, r_n) = \{o \mid o \in meet(r_1, \ldots, r_n) \text{ and } \wp(o) \notin R\}$$

For example, by setting R to $\{bibliography\}$ we can filter out uninteresting matches, *i.e.*, where the meet corresponds to the document root, in large bibliographies. This variant is also used in the case study in Section 5.

Another interesting application of the operator is distance calculation: the number of joins executed while calculating $meet_P(o_1, o_2)$ for two nodes o_1 and o_2 corresponds to the number of edges on the shortest path from o_1 to o_2. So we can define

$$d(o_1, o_2) = \text{number of joins when calculating } meet_P(o_1, o_2).$$

Building on this we can define another restricted version that is occasionally useful to block undesired matches:

$$k\text{-}meet_P(o_1, o_2) = \begin{cases} \perp & \text{if } d(o_1, o_2) > k, \\ meet_P(o_1, o_2) & \text{otherwise,} \end{cases}$$

The number of joins is also a simple yet effective heuristic for establishing a ranking between the result OIDs.

We believe that it is worthwhile to apply additional heuristics like distances in the source file or even more complicated information retrieval techniques to improve the ranking of the answer set. In particular, thesauri are a promising tool to help a user find interesting results, especially to broaden a search that returned too few answers.

Additionally, we mention a convenient application of the meet operator: staying in the bibliography domain, we may want to know whether a certain bibliographical item that we found in one bibliography also lives in another bibliography; however, we have no idea how the relevant information is marked up. So a good approach is to combine the meet operator with fulltext search similar to the introductory example and use the results as a starting point for displaying and browsing.

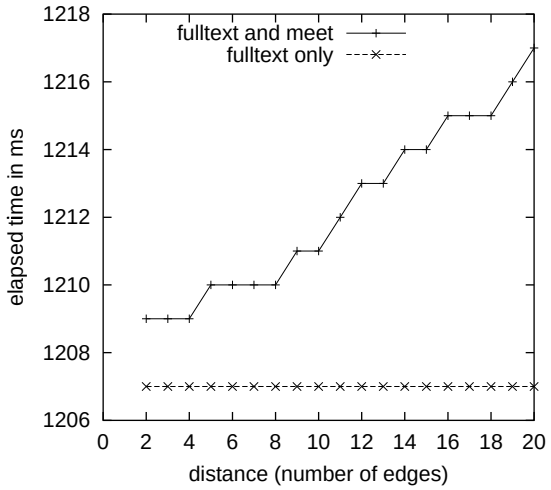

Figure 6. Combining meet and fulltext search (normalized)

5. Performance

In this section, we assess the performance characteristics of two versions the meet operator: $meet_P$ and $meet$. We will see that the costs of these operators are negligible if they are used in combination with a relatively selective fulltext search and that the set-oriented version of the operator scales well, *i.e.*, linear, with respect to the cardinality of the input sets.

We implemented the meet operator on top of the Monet XML module [19] within the Monet database server [6]. The measurements were carried out on an Silicon Graphics 1400 Server with 1 GB main memory, running at 550 MHz. Two XML sources were used: a file of about 200 MB with descriptions of multimedia data items, extracted by feature detectors [20], and the DBLP bibliography, which is available on the Internet [16]. For the first experiment the total main memory requirements of the database server were about 120 MB, the second experiments could be run in 100 MB. Note that only a fraction of the main memory was needed to compute the meet; most of it was necessary for our main memory DBMS to load relations and perform operations on them.

Figure 6 shows the run-time behavior of a typical query such as the one presented in the introduction; however the underlying database is a file of descriptions of multi-media data items. In the plot, we normalized the duration of the full-text search to an average value as its execution varies greatly in relation to the little time the computation of the meet consumes. The figure shows two things: First, the execution time is dominated by the full-text search, which takes 1207 ms as opposed to the 2 ms the computation of

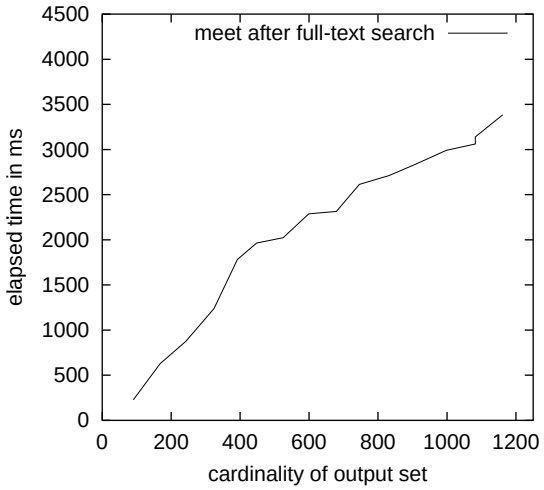

Figure 7. Performance graph of Case Study

the meet of objects with distance two. Secondly, the meet scales well with respect to distance of the objects. Therefore it can serve as a sensible and valuable add-on to an already existing search engine for semi-structured or XML data that comes at little cost.

Case Study. We now take a look at a meet query run against the DBLP bibliography [16]. We prepared the bibliography by bulk loading it into Monet XML as described in [19]. We now want to list all publications in the ICDE proceedings of a certain year. To achieve this, we do a full-text search for the strings "ICDE" and the year and calculate the meets of the results according to algorithm $meet_R$ with the document root excluded from the set of possible results. To demonstrate that the algorithms scales we iteratively extend the search interval from 1999 back to 1984 (note that there was no ICDE in 1985, hence the small step at about 1100 on the x-axis), which gives us control over the size of the result set. The results resemble to a large degree our intuition and consist mostly of the ICDE publications of the respective year (there were just two false positives). The graph in Figure 7 shows the time elapsed for calculating the meet, *e.g.*, for a result set of 1000 publications the computation takes about three seconds (the time the full-text search takes is not included in this figure). Note that the input sets are fairly large: they contain *all* associations whose string component contains the year, *i.e.*, all publications in the bibliography between 1984 and 1999 are involved. This demonstrates that the algorithm scales well to large datasets and is suitable for interactive querying.

We finally remark that the performance behavior of the meet may differ on different underlying physical data models: not all XML-to-database mappings preserve as much information as the Monet model. However, we expect queries with small result sets to perform favorably on many relational models.

6. Related Work

There have been a number of attempts to make querying XML documents or semi-structured data easier for users. In [12] the authors enrich XML-QL with keyword search on subtrees of certain tags. The DBMS Lore [17] also supports keyword and distance search. The difference to our work is that the result types have to be made explicit in the queries, which is what the meet operator avoids and hence allows simpler query formulation. Furthermore, by restricting the result types, the operator can be used to implement keyword search as a special case. In [13] the authors present algorithms for proximity search in graphs; their queries follow a 'Find *objects from* S_1 Near *objects from* S_2' pattern where the user has to specify sets S_1 and S_2; therefore formulating these queries also requires more domain-knowledge than is needed for meet queries.

Another view on our work is that we are trying to exploit the inherent semantics encoded in the tag hierarchies; a very interesting approach to combining knowledge from outside the database with internal knowledge is [14]. However, this approach is of different nature and only complementary to ours.

The algorithmic problem of finding lowest common ancestors yet novel to XML processing as a query primitive has a long history in databases and code optimization, see [5, 4]. We also assume that especially relational XML Query processors that support XQL's *before* and *after* predicates already provide some of the functionality a full implementation of the meet operator requires.

7. Conclusion

We have introduced the $meet$ operator, a tool that lets users query XML databases with whose content they are familiar with but whose schema or structure they are unaware of. We have shown that it neatly fits current XML datamodels and that query languages can be easily extended to incorporate the additional functionality. Furthermore, we demonstrated that the algorithms yield useful results on real world data and scale well, enabling interactive querying. The novelty of our work is that the result type of the query is not specified by the user but dependent on the database instance queried. Therefore we referred to meet queries as nearest concept queries.

Future research will include further investigations into expanding the applications of the meet operator with respect to information retrieval techniques; some aspects are already present in this paper like ranking and restrictions. We

are also looking at how to incorporate views and IDREFs, which may break the tree structure of the database, into the search process.

Acknowledgements. The authors would like thank Zbigniew R. Struzik and Florian Waas for discussions on the paper and the reviewers for their constructive comments.

References

[1] S. Abiteboul. Querying Semi-Structured Data. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 1–18, 1997.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On Finding Lowest Common Ancestors in Trees. *SIAM Journal of Computing*, 5(1):115–132, 1976.

[5] A. V. Aho, Y. Sagiv, T. G. Szymanski, and J. D. Ullman. Inferring a Tree from Lowest Common Ancestors with an Application to the Optimization of Relational Expressions. *SIAM Journal of Computing.*, 10(3):405–421, 1981.

[6] P. Boncz and M. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, 1999.

[7] A. Bonifati and S. Ceri. Comparative Analysis of Five XML Query Languages. *ACM SIGMOD Record*, 1(29):68–79, 2000.

[8] R. Braumandl, J. Claußen, A. Kemper, and D. Kossmann. Functional-join processing. *The VLDB Journal*, 8(3-4):156–177, 2000.

[9] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *International Workshop on the Web and Databases (In conjunction with ACM SIGMOD)*, pages 53–62, Dallas, TX, USA, 2000.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. available at `http://www.w3.org/TR/NOTE-xml-ql/`.

[11] M. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 14–23, 1998.

[12] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. In *Proceedings of the International World Wide Web Conference*, pages 119–135, 2000.

[13] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. García-Molina. Proximity Search in Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 26–37, 1998.

[14] R. Goldman and J. Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proceedings ACM SIGMOD International Conference on Management of Data*, 2000. 285–296.

[15] S. Grumbach and G. Mecca. In Search of the Lost Schema. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 314–331, 1999.

[16] M. Ley. DBLP Bibliography. `http://www.informatik.uni-trier.de:8000/~ley/db/`.

[17] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, , and J. Widom. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66, 1997.

[18] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`, 1998.

[19] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In *International Workshop on the Web and Databases (In conjunction with ACM SIGMOD)*, pages 47–52, Dallas, TX, USA, 2000.

[20] A. Schmidt, M. Windhouwer, and M. Kersten. Feature Grammars. In *Proceedings of the International Conference on Systems Analysis and Synthesis*, Orlando, Florida, 1999.

[21] M. Kersten *et al.* Monet Homepage. `http://www.monetdb.org`.

[22] W3C. Document Object Model (DOM). `http://www.w3.org/DOM/`.

[23] W3C. Extensible Markup Language (XML) 1.0. available at `http://www.w3.org/TR/1998/REC-xml-19980210`.