Trading with plans and patterns in XQuery optimization

Andrei Arion
INRIA Futurs and Univ. Paris XI, France
Andrei.Arion@inria.fr

Véronique Benzaken
LRI-CNRS, Univ. Paris XI, France
Veronique.Benzaken@lri.fr

Ioana Manolescu
INRIA Futurs, France
Ioana.Manolescu@inria.fr

Yannis Papakonstantinou
UCSD, USA
yannis@cs.ucsd.edu

1. INTRODUCTION

The query optimizer is a central component of a database management system that attempts to determine the most efficient way to execute a query. During the optimization phase the query optimizer builds execution plans - (functional) programs that are interpreted by the evaluation engine to produce the query result. The optimizer decides then which of the possible query plans will be the most efficient in terms in terms of CPU requirements and the number of I/O operations.

A simplified view of the optimization process for XML queries is depicted in Figure 1. To process a query expressed in some query language, first, data access plans are built by examining all the possible access paths (e.g. index scan, sequential scan) of the existing storage modules, views and indices. Then, these plans are combined with the help of algebraic operators (such as selections, joins...) into increasingly larger plans, aiming towards building plans for the complete query (at this step several join orderings and equivalent algebraic alternatives are considered thus further increasing the total number of query evaluation plans that are explored). In the final phase of the optimization process the best plan that is equivalent to the original query is picked for execution.

Significant research and development efforts are currently being invested in implementing efficient XQuery [33] processing methods. Efficient new execution algorithms have been proposed, such as structural joins [3] and their holistic variants [12]. At the logical level, several algebras have been devised in recent years [11, 19, 28].

Query graphs, or trees, such as the classical Query Graph Model (QGM [18]) for optimizing SQL queries, are common abstractions in logical query optimizers. XQuery optimizers often use a form of XQuery graphs, whose basic ingredient is some form of *tree patterns* as in [4, 6, 13, 14, 19, 25]. Tree patterns are convenient for several reasons. They capture

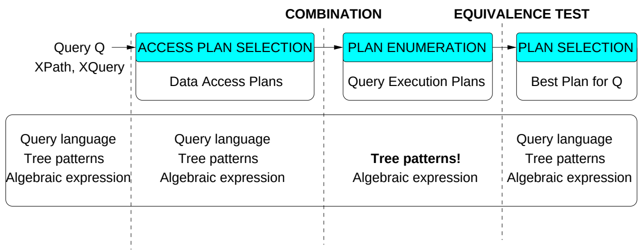


Figure 1: Query processing stages and their correspondences with various existing formalisms.

the elementary navigation operations specific to XML and XML query languages. Their semantics is well-understood, based on the notion of tree embeddings [1]. Such clean semantics have allowed the obtention of important theoretical results concerning tree pattern minimization [4, 14], containment, and rewriting [15, 16, 24, 26, 30]. Tree patterns are also the common abstraction for XML query cardinality estimations [2, 21, 27] and also serve to describe *the contents of persistent XML storage structures*, such as tables, indexes, or materialized views [5, 6]. Many XML structural indexing schemes, in particular, are described by tree patterns [20, 25]. In this context, reasoning about the usefulness of an index for a given query can be done in terms of tree patterns, once the index and the query have been brought to this formalism.

The different formalisms (query language, algebra, and tree patterns) are used in various processing stages and their applications are summarized in Figure 1:

- *Queries* are directly expressed in some query language and recently many algorithms for translating queries to algebraic plans have been devised [7, 11, 28]. Also, the translation of XQuery subsets to tree patterns has been studied in [7, 13].

- *Persistent XML storage structures* are often described in languages like XPath/XQuery (e.g. [32]) or using tree pattern languages [5, 6]. Tree pattern semantics is typically defined by embeddings, and in some cases also by means of algebra [6].

- *Algebraic plans* are naturally expressed in an algebra. [11, 23, 28] and are not easily expressible by a query language. This is because they may refer to features that do not belong to the logical XML data model, such that persistent node identifiers which belong to the physical data model or tuples, which do not belong to the XQuery data model. Other algebraic features (such as group-by) could be expressed based on the logical data model but do not belong directly to the query language.

More precisely, algebraic plans can be divided in two classes:

- *Data access plans* - describe operators that access storage structures such that a native storage or a materialized view, thus they can be also expressed in the formalisms corresponding to the data structures.
- On the other hand, more *complex query plans* do not have equivalence in formalisms other than algebra.

The contribution of this paper is to show how query optimizers can build and exploit in parallel algebraic plans, and *equivalent corresponding tree patterns*, where equivalence is formally defined under a set of structural constraints on the XML documents[1]. We are aware of no existing technique to connect tree patterns with complex algebraic plans, and this is what we set out to achieve. We argue this connection is crucial for bringing together valuable XML query processing models and results:

- While optimizers naturally work with algebraic plans, in the realm of XML, the presence of constraints describing the tree structure of the data, and possibly other constraints (e.g. types), make reasoning about equivalence and containment hard (or unnatural) *at the level of the algebra*. This is to be contrasted with the relative simplicity of relational algebra equivalence laws [1]. However, equivalence and containment algorithms are available *at the level of patterns* [15, 16, 24, 26, 30].
- Establishing equivalences between tree patterns and algebraic plans allows existing techniques for estimating tree pattern cardinality [2, 21, 27] to apply to the problem of estimating the cardinality of algebraic plans.
- The ability to reason about containment and equivalence of algebraic expressions is crucial, among others, to the process of view-based XML query rewriting, during which we need to decide if a plan is equivalent to the target query. To that effect, we need to determine the equivalent pattern for a given algebraic plan.
- Connecting plans and patterns reduces the tension between what was perceived as two incompatible formalisms, helps comprehension and simplifies future work on optimizer design.

[1]This work is situated in the context of the ULoad framework for exploiting XML materialized views [8], based on a set of structural constraints over the XML documents. These constraints are closely related (and may generalize to) type constraints such as expressed in DTDs and XML Schemas.

Table 1 position our contributions with respect to the relationships between queries and various formalisms established by prior works.

	Query language	Algebra	Tree patterns
Query	direct	[11, 23, 28]	[7, 13, 14]
Data access plans	[10, 32]	[5, 6]	[5, 6, 20]
Complex plans	–	[11, 23, 28]	**this work**

Table 1: Formalisms used in XML query processing

The paper is organized as follows. Section 2 describes the tree patterns, algebra, and structural constraints framing our work. Section 3 is the core of the paper, providing an algorithm for building plans and their equivalent patterns throughout query planning. Section 4 addresses implementation and performance issues, then we conclude.

2. PRELIMINARIES

In this section, we outline the data model, the XAMs tree patterns, the structural constraints, and algebra used in this work.

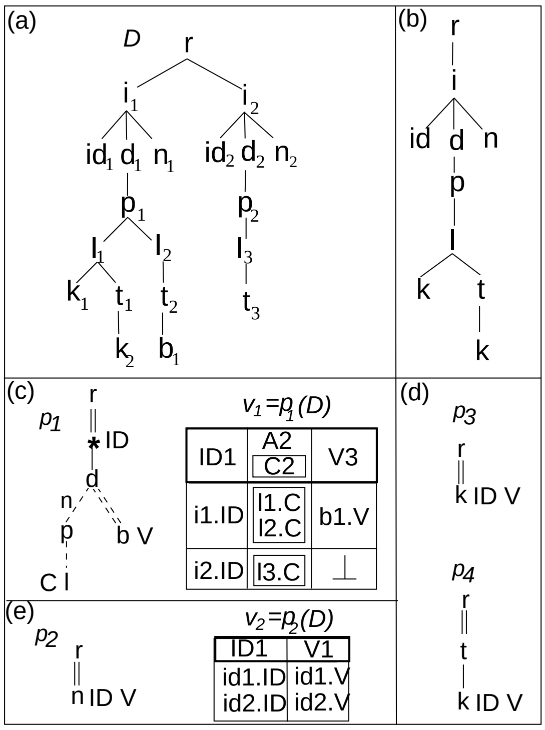
2.1 Data model

We view an XML document as an unranked labeled ordered tree. Every node n has (i) a unique identity from a set $\mathcal{I}$, (ii) a tag $label(n)$ from a set $\mathcal{L}$, which corresponds to the element or attribute name, and (iii) may have a value from a set $\mathcal{A}$, which corresponds to atomic values of the XML document. We consider two special binary predicates on $\mathcal{I}$ values: $i_1 \prec i_2$ is true iff i_1 identifies an XML node which is the parent of the node identified by i_2, and similarly, $i_1 \prec\!\!\prec i_2$ iff the node identified by i_1 is an ancestor of the node identified by i_2.

A simplified XMark [31] document fragment appears in Figure 2(a). For conciseness, we abridge element names to the first letter when possible; thus, i stands for *item*, d for *description*, n for *name*, p for *parlist*, l for *listitem* etc.

2.2 Tree patterns: XAMs

In this work, we use a particular flavor of tree patterns, originally introduced as XML Access Modules (or XAMs) [6]. XAMs semantics is defined both by means of an algebra [6] and using tree embeddings [22]. They can be used both to describe persistent storage modules [6] and to express parts of a complex XQuery query [7]. The advantage of XAMs comes from their important expressive power: like close competitors [13], XAMs feature parent-child and ancestor-descendant edges, optional and required edges, value and structure conditions. However, their semantics gives them an advantage over GTPs, since one XAM may capture several nested XQuery FLWOR blocks [7], whereas each GTP is confined to a single such block.



$v_1 = p_1(D)$

ID1	A2 / C2	V3
i1.ID	l1.C l2.C	b1.V
i2.ID	l3.C	⊥

$v_2 = p_2(D)$

ID1	V1
id1.ID	id1.V
id2.ID	id2.V

Figure 2: Sample XMark document (a), its path summary (b), and tree patterns (c)-(e).

We illustrate XAMs using the p_1 and p_2 patterns depicted in Figure 2(c). Edges labeled / denote parent-child and // denote descendant relationships. Dashed edges connect optional children to their parents: an XML node matching the parent node in the pattern may lack children matching the optional child pattern node, yet correspond to the pattern node. Finally, XAM edges may be *nested*, with the consequence that the semantics of a XAM is a set of nested tuples. The data matching the pattern child under a nested edge will be nested inside the tuple(s) corresponding to the pattern parent node above the nested edge, as we shortly explain.

Each pattern node is annotated with the list of *information items it stores*, and with the *conditions it imposes* on each node. In Figure 2(c), all nodes satisfy some conditions on their names (in this case the node is simply labeled with the required name), except for the $*$ node in p_1. Nodes may also be annotated with conditions on their values, corresponding to the XPath predicate *text()=c* for some constant c (not illustrated in Figure 2). Thus:

- v_1 stores, for every $/r//*$ element, three information items. (1) Its *persistent identifier* in the column ID_1. We consider identifiers are simple atomic values. (2) The grouped set of the *contents* of its possible d/l descendant nodes. The content of a node denotes the full subtree rooted in the node, which the view may store directly (perhaps in a compact encoding), or as a pointer to the subtree stored elsewhere. In all cases, downward XPath navigation is possible inside a content attribute. In Figure 2, attribute C_2 is nested in the second view attribute A_2, reflecting the nested edge. (3) The *value* (text children) of its possible b (*bold*) descendants. Note the null value (denoted $\perp$) representing the missing b descendants of the i_2 element.
- v_2 stores, for every $/r//n$ element, its identifier ID_1 and value V_1.

A fourth item that a pattern may store from a node is its *name* (not illustrated in Figure 2).

The semantics of a XAM pattern is a set of *nested tuples*, whereas the tuple and set constructor alternate. This is illustrated in Figure 2. The tuple set v_1 corresponds to the semantics of pattern p_1 on the sample document in Figure 2(a), denoted in the Figure 2(c) by $v_1 = p_1(D)$. Observe that attribute A_2 is nested; it contains a table whose single attribute is labeled C_2. In the tuple corresponding to the first i element, the value of A_2 is a table, whose first tuple contains the contents of its l_1 descendant, while the second is the contents of its l_2 descendant etc. The view v_2 corresponds to pattern p_2 on the sample document.

2.3 Path summaries

In this work, we use path summaries (a flavor of Dataguides [17]) to encapsulate structural constraints over XML documents.

The *path summary* $S(D)$ of an XML document D is a tree, whose nodes are labeled with element names from the document. The relationship between D and $S(D)$ can be described based on a function $\phi : D \to S(D)$, recursively defined as follows:

1. ϕ maps the root of D into the root of $S(D)$. The two nodes have the same label.
2. Let $child(n, l)$ be the set of all the l-labeled XML elements in D, children of the XML element n. If $child(n, l)$ is not empty, then $\phi(n)$ has a unique l-labeled child n_l in $S(D)$, and for each $n_i \in child(n, l)$, $\phi(n_i)$ is n_l.
3. Let $att(n, a)$ be the value of the attribute named a of element $n \in D$. Then, $\phi(n)$ has an unique child n_a labeled @a, and for each $n_i \in att(n, a)$, we have $\phi(n_i) = n_a$.

Clearly, ϕ preserves node labels, and parent-child relationships. For every simple path $/l_1/l_2/.../l_k$ in D, there is exactly one node reachable by the same path in $S(D)$. Conversely, each node in $S(D)$ corresponds to a simple path in D.

Figure 2(b) shows the path summary for the XML fragment at its left.

We add to the path summary some more information, conceptually related to schema constraints. More precisely, for any summary nodes x, y such that y is a child of x, we record on the edge x-y whether every node on path x has *exactly one child* on path y (we say the edge is annotated with 1), or *at least one child* on path y (edge annotated with $+$), or may lack y children (* annotation). Such schema-like information enriches the summary and helps decide tree pattern containment [22].

2.4 Tree pattern path annotations

As stated before, tree pattern containment and equivalence reasoning is greatly helped by the presence of structural XML constraints. More formally, let $\mathcal{C}$ be a set of structural constraints, and p_1, p_2 be two patterns. We say $p_1 \subseteq p_2$ (or p_1 is contained in p_2) if for any XML document D, $p_1(D) \subseteq p_2(D)$. The latter inclusion is of course defined in terms of nested tuple sets. Pattern equivalence is defined as two-way containment. In the presence of constraints, we say $p_1 \subseteq_{\mathcal{C}} p_2$ if for any XML document D on which the constraints $\mathcal{C}$ hold, we have $p_1(D) \subseteq p_2(D)$. Equivalence under constraints is defined as two-way constrained containment.

In our setting, a path summary plays the role of the constraint set $\mathcal{C}$. More precisely, a summary constrains the set of children that a node on a given path may have, and may constrain the cardinality of such parent-child relationships, too, if the corresponding edge is labeled with 1 or $+$. Let S be a path summary. Containment under S constraints, denoted $p_1 \subseteq_S p_2$, is true for two patterns p_1 and p_2 as soon as for any document D whose path summary coincides with S, $p_1(D) \subseteq p_2(D)$. Algorithms for deciding tree pattern containment under such constraints are described in a separate work [22].

Containment testing is based on the result of a static analysis step, performed on the tree patterns based on path summaries. This static analysis identifies the summary paths on which each pattern node may have matches; it bears some similarity to a typing process. The result of the static analysis is a pattern path annotation, defined next.

Definition 2.1. Let *tp* be a tree pattern and S be a summary. The path annotation of *tp* based on S is a function $\alpha(tp, \cdot) : tp \to \mathcal{P}(S)$, associating to every *tp* node a set of S nodes as follows. For any node $n \in tp$ and path $p \in S$, $p \in \alpha(tp, n)$ *iff* for some document D conforming to S, there exists an XML node on path p in D, matching n. □

For instance, let r be the root node in p_2 in Figure 2(c). If $\alpha(p_2, \cdot)$ is the path annotation of p_2 based on S, $\alpha(p_2, r) = \{/r\}$. Similarly, if n is the leaf node (labeled n) in p_2, $\alpha(p_2, n) = \{/r/i/n\}$. Finally, the annotation of the k node in pattern p_3 in Figure 2 is: $\{/r/i/d/p/l/k, /r/i/d/p/l/t/k\}$.

The following simple examples show the benefits of path annotations when judging containment. Assume we need to know whether $p_3 \equiv_S p_4$ in Figure 2. The annotation of the k node in p_4 is $\{/r/i/d/p/l/t/k\}$, therefore $p_4 \subseteq_S p_3$ but $p_3 \not\subseteq_S p_4$.

In a more subtle example, let p_5 be a pattern returning the ID of all d nodes ($//d$ in XPath syntax), and p_6 be a pattern returning the IDs of all parents of some d nodes ($// * [d]$ in XPath). The path annotation of the return node both in p_5 and p_6 is $\{/r/i\}$, since only i elements may have d children. Therefore, we conclude that $p_6 \subseteq_S p_5$. If, moreover, the summary edge between i and d in Figure 2(b) is labeled 1 or $+$, then we conclude $p_5 \equiv_S p_6$.

Path annotations can be efficiently computed, evaluating in a streaming fashion the pattern on the path summary. The time and space complexity of computing the paths associated to all nodes of a pattern P is $O(|P| \times |PS|)$. For the detailed algorithm of computing the path annotations we direct the reader to [9].

2.5 Algebra

Without loss of generality, we set this work in the context of the nested tuple algebra described in [23]. An atomic attribute may take its values either all from $\mathcal{I}$, or all from $\mathcal{A}$ (recall Section 2.1). To every nested relation r, corresponds a $Scan(r)$ operator, also denoted r, returning the (possibly nested) tuples of r. Other standard operators are the cartesian product $\times$, the union $\cup$ and the set difference $\setminus$ (which do not eliminate duplicates).

We consider predicates of the form $A_i \ \theta \ c$ or $A_i \ \theta \ A_j$, where c is a constant. θ ranges over the comparators $\{=, \leq, \geq, <, >, \prec, \prec\!\!\prec\}$, and $\prec$, $\prec\!\!\prec$ only apply to $\mathcal{I}$ values.

Let *pred* be a predicate over atomic attributes from r, or r and s. Selections σ_{pred} have the usual semantics. A join $r \bowtie_{pred} s$ is defined as $\sigma_{pred}(r \times s)$. For convenience, we will also use outerjoins $⟕_{pred}$ and semijoins $\ltimes_{pred}$ (although strictly speaking they are redundant to the algebra). Another set of redundant, yet useful operators, are *nested joins*, denoted $\bowtie^n_{pred}$, and *nested outerjoins*, denoted $⟕^n_{pred}$, with the following semantics:

$$r ⟕^n_{pred} s = \{(t_1, \{t_2 \in s \mid pred(t_1, t_2)\}) \mid t_1 \in r\}$$
$$r \bowtie^n_{pred} s = \{(t_1, \{t_2 \in s \mid pred(t_1, t_2)\}\ ) \mid t_1 \in r, \{t_2 \in s \mid pred(t_1, t_2)\} \neq \emptyset \ \}$$

An interesting class of logical join operators (resp. nested joins, outerjoins, nested outerjoins, or semijoins) is obtained when the predicate's comparator is $\prec$ or $\prec\!\!\prec$, and the operand attributes are identifiers from $\mathcal{I}$. Such operators are called *structural joins*. Observe that we only refer to *logical structural joins*, independently of any physical implementation algorithm; different algorithms can be devised [3, 12].

Let $A_1, A_2, \ldots, A_k$ be some atomic r attributes. A projection $\pi_{A1,A2,\ldots,Ak}(r)$ by default does not eliminate duplicates. Duplicate-eliminating projections are singled out by a superscript, as in π^0. The group-by operator $\gamma_{A_1,A_2,\ldots,A_k}$, and unnest u_B, where B is a collection attribute, have the usual semantics [1].

We use the *map* meta-operator to define algebraic operators which apply *inside* nested tuples. Let *op* be a unary operator, $r.A_1.A_2 \ldots A_{k-1}$ a collection attribute, and $r.A_1.A_2 \ldots A_k$ an atomic attribute. Then, $map(op, r, A_1.A_2 \ldots A_k)$ is a unary operator, and:

- If $k = 1$, $map(op, r, A_1.A_2.\ldots.A_k) = op(r)$.
- If $k > 1$, for every tuple $t \in r$:

– If for every collection $r' \in t.A_1$, $map(op, r', A_2.\ldots.A_k) = \emptyset$, t is eliminated.

– Otherwise, a tuple t' is returned, obtained from t by replacing every collection $r' \in r.A_1$ with $map(op, r', A_2.\ldots.A_k)$.

For instance, let $r(A_1(A_{11}, A_{12}), A_2)$ be a nested relation. Then, $map(\sigma_{=5}, r, A_1.A_{11})$ only returns those r tuples t for which *some* value in $t.A_1.A_{11}$ is 5 (existential semantics), and reduces these tuples accordingly. *Map* applies similarly to π, γ and u. By a slight abuse of notation, we will refer to $map(op, r, A_1.A_2.\ldots.A_k)$ as $op_{A_1.A_2.\ldots.A_k}(r)$. For instance, the sample selection above will be denoted $\sigma_{A_1.A_{11}=5}(r)$.

Binary operators are similarly extended, via *map*, to nested tuples (details omitted). The algebra also features an operator for new node construction, of less interest here [23].

3. OPTIMIZING WITH PLANS AND PATTERNS

This section describes our approach for building algebraic plans together with equivalent tree patterns throughout the optimization process. Section 3.1 outlines the overall process, while Section 3.2 details the most technically involved step in this process: synthesizing tree patterns equivalent to a given complex query plan.

3.1 Outline

We consider the following setting. An XQuery query q is evaluated against a database containing several persistent data collections, such as persistent trees, relational tables, tag, value, or structural indices etc. A set of *query tree patterns* $\{qt_1, qt_2, \ldots, qt_k\}$ has been extracted from the query q, applying the extraction algorithms of [7, 13]. Each data collection stored in the database is described by a *view tree pattern*, yielding the view tree pattern set $\{v_1, v_2, \ldots, v_n\}$. Without loss of generality, we will consider q concerns a single document D, characterized by the constraints encapsulated in a summary S.

The query optimization problem thus becomes: for each query pattern $qt \in \{qt_1, qt_2, \ldots, qt_k\}$, find all algebraic plans p such that $p \equiv_S qt$. Such plans are algebraic trees whose leaves are labeled with views from $v_1, v_2, \ldots, v_n$, and whose internal nodes are labeled with algebraic operators described in Section 2.5.

This optimization (or rewriting) problem requires exploring a space of algebraic plans, using some search strategy. For simplicity, we consider a simple bottom-up dynamic programming approach, as outlined in [22]:

1. For every view v_i, construct a *(plan, pattern)* pair, such that the plan is $Scan(v_i)$ (denoted simply v_i), and the pattern is v_i *endowed with path annotations against the summary* S, as explained in Section 2.4. Observe that in these initial pairs, the plan and the pattern are naturally S-equivalent.
2. From the existing plan-pattern pairs, build increasingly larger ones as follows: combine two pairs (p_i, t_i) and (p_j, t_j) into $(p_i \bowtie p_j, t_z)$, where $\bowtie$ denotes some join operation (either on an ID attribute or a structural join), and t_z is a pattern *computed such that* $p_i \bowtie p_j \equiv_S t_z$.
   For every (p_i, t_i) plan-pattern pair thus obtained, test whether $t_i \subseteq_S qt$, where qt is the target pattern.[1]
   - If $t_i \subseteq_S qt$, then (p_i, t_i) is added to a set PR of partial rewritings.
   - If $t_i \not\subseteq_S qt$, then (p_i, t_i) is put back in the pool of partial plans as an ingredient for larger plans.
3. When plan enumeration stops,[2] for every subset $\{(p_1, t_1), (p_2, t_2), \ldots, (p_m, t_m)\}$ of the partial rewritings set PR, test whether $t_1 \cup t_2 \cup \ldots \cup t_m \equiv_S qt$. If yes, then $p_1 \cup p_2 \cup \ldots p_m$ is an S-equivalent rewriting of qt.

Plan combination via (structural) joins is straightforward.

The difficulty left to handle is the construction of a tree pattern S-equivalent to a given query plan, such as the pattern t_z in the above algorithm, which must be S-equivalent to the plan $p_i \bowtie p_z$. Before discussing the detailed algorithm, let us consider some examples.

Figure 3 depicts a summary S, whose b nodes are numbered for convenience, and a set of patterns. Next to each node labeled b in these patterns, we show its path annotation against S as a set of integers, corresponding to the respective b nodes in S. We assume patterns p_1, p_2, p_4 and p_7 describe the data stored in the materialized views v_1, v_2, v_4 and v_7, thus the rewriting algorithm starts with the (plan, pattern) pairs (v_1, p_1), (v_2, p_2), (v_4, p_4) and (v_7, p_7).

Let us consider combining (v_1, p_1) and (v_2, p_2). The pattern equivalent to $v_1 \bowtie_{bID=bID} v_2$ would return the IDs of b elements with an a ancestor, together with their x and y children's ID. The resulting equivalent pattern turns out to be p_3 in Figure 3.

Now consider combining (v_1, p_1) with (v_4, p_4). In the pattern S-equivalent to $v_1 \bowtie_{bID=bID} v_4$, the b node should have both an a and a c ancestor. Moreover, in this pattern, the b node's annotation should be the intersection of the b nodes' annotations in p_1 and p_4, that is, the path set $\{1, 3, 7, 8\}$. From the structure of S, it follows that a is an ancestor of c on paths 1 and 7, while c is an ancestor of a on paths 3 and 8. Thus, there is no way to place a and c as ancestors of b in a hypothetical pattern S-equivalent to $v_1 \bowtie_{bID=bID} v_4$. However, if we relax the problem by also considering *pat-*

[1]Some simple transformations may be opportunistically applied on p_i and t_i prior to the containment test. Examples of such transformations are: adding a selection on the name or value of some t_i node (and correspondingly on p_i), or a projection, to align t_i to the needs of qt; adding a γ or an *unnest* operator on p_i to nest/unnest a t_i edge to align it with a qt edge etc.

[2]Under S constraints, the size of rewriting plans is bounded by a factor related to $|S|$ [22], thus rewriting can stop after enumerating such plans. More generally, the document's tree depth can be used as a bound on the depth of a structural join tree, or rewriting can be stopped as soon as some solutions are found etc.

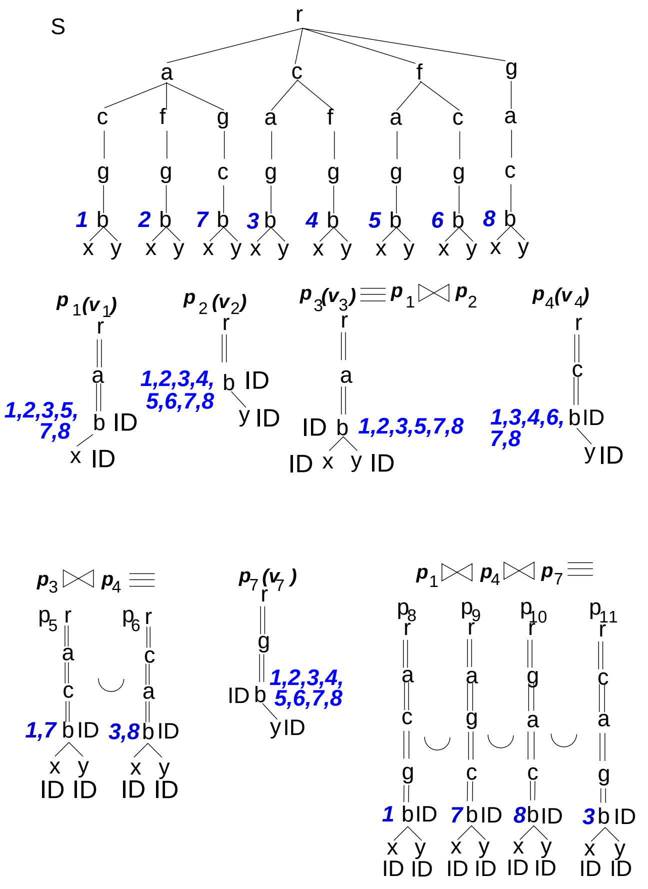


Figure 3: Sample summary S, patterns representing stored structures (p_1, p_2, p_4, p_7) and patterns obtained by successive joins ($p_3, p_5, p_6, p_8, p_9, p_{10}, p_{11}$).

tern unions, then a solution can be found. For instance, in Figure 3, $v_1 \bowtie_{bID=bID} v_4 \equiv_S p_5 \cup p_6$.

Now consider the join plan $(v_1 \bowtie_{bID=bID} v_4) \bowtie_{bID=bID} v_7$. By a similar reasoning carrying over the a, c and g ancestors of the b node, no single pattern is S-equivalent to this plan, however the union $p_8 \cup p_9 \cup p_{10} \cup p_{11}$ is equivalent to this plan.

It can be shown that any algebraic plan built with $\bowtie_=$, $\bowtie_\prec$, $\bowtie_{\prec\prec}$, and π on top of some patterns $p_1, \ldots, p_n$ is S-equivalent to a union of patterns. While the patterns in Figure 3 only feature non-optional, non-nested edges, this result carries over for general-case XAMs.

3.2 Computing the pattern equivalent to a join plan

Algorithm 1 shows how to combine two (plan,pattern) pairs via a join on the plans, into a resulting (plan,pattern) pair. Its input are two patterns p_1 and p_2, and its output is a pattern p which is S-equivalent to $p_1 \bowtie_{a_k.ID=b_l.ID} p_2$. We call this algorithm *pattern zipping*, due to the way it handles pattern nodes (see below). Observe that we may sometimes need to zip *unions of patterns*, not simple patterns; for simplicity we focus on zipping two patterns for now, and will extend this later to pattern unions.

The zipping algorithm can be traced by considering Figure 4. Here, p_1 and p_2 are shown distinguishing the ancestors of a_k in p_1 with the $a_1, a_2, \ldots, a_{k-1}$ labels, and similarly the ancestors of b_l in p_2. The edges on the path from p_1's root to a_k are labeled $e_1, e_2, \ldots, e_{k-1}$, while the remaining children of each a_i node are grouped in a forest denoted S_i. Similar e' edge labels and S' forests are outlined in p_2.

Algorithm 1 is called with a focus on two specific nodes a_i and b_j from p_1, respectively p_2. Initially, $i = k$ and $j = l$; in subsequent recursive calls, a_i and a_j will move up from a_k, respectively b_l, towards the pattern roots. It is due to this gradually moving pair of nodes that we call the algorithm "zipping".

We assume a_k and b_l have compatible labels (either the same label, or one label is $*$), compatible conditions on their values, if any, and the intersection of their path annotations is non-empty (otherwise, the join result is empty).

At the first invocation, $a_i = a_k$ and $b_j = b_l$, and Algorithm 1 (lines 1-4) computes the subtree of the node which, in the resulting pattern p, corresponds to the unification of nodes a_k and b_l. We will call this node a_k also in p. The p subtree rooted in a_k is built as follows:

- The children of b_l (the S'_l forest in Figure 4) are copied as children of a_k in p. Thus, the children of a_k in p are the subtrees from forests S_k and S'_l.
- The path annotation for a_k in p is computed as the intersection of the paths annotations of a_k in p_1 and of b_l in p_2.
- The path annotations of a_k's descendants in p are modified to reflect a_k's path annotation. We call this *downward path propagation from a_k in S_k*. During downward propagation, for every descendant x of a_k in S_k, the path annotation of x is restricted to those paths which are descendants of some path now annotating a_k. If for a non-optional descendant x, the path annotation has become $\emptyset$, then p (and the join) have empty result.
  We have still not determined the shape of p *above* the node a_k; this is more delicate and may lead to pattern unions, as illustrated by $v_1 \bowtie_{bID=bID} v_4$ in Figure 3. In preparation of that step, the path annotation of node a_{k-1} in p_1 is restricted to those paths which have a descendant in the path annotation of a_k in p. Then, downward path propagation is applied from a_{k-1} into S_{k-1}, subtree of p_1 (see Figure 4). The process is repeated on a_{k-2}, …, a_1. Similarly, the annotations of b_{l-1} are restricted, then propagated into S'_{l-1}, and so on until b_1.

The part of the resulting pattern *above* node a_k is computed by lines 5-26 in Algorithm 1.

For the current a_i and b_j nodes, we compute three sets as follows: P_1 holds the paths from the annotation of a_i that

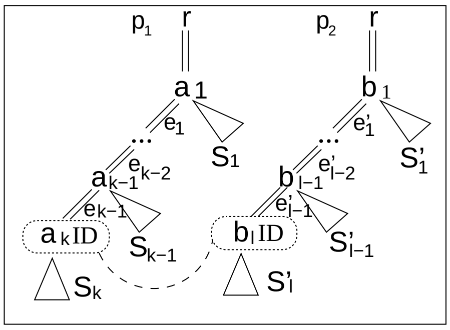


Figure 4: Zipping patterns p_1 and p_2 on $a_k.ID = b_l.ID$.

are ancestors of a path from the annotation of b_j. Symmetrically, P_2 holds the paths from $\alpha(p_2, b_j)$ that are ancestors of a path from $\alpha(p_1, a_j)$. Finally, P_3 gathers the paths belonging both to the path annotation of a_i in p_1, and to the annotation of b_j in p_2.

If P_1 is not empty, this means that for some documents conforming to S, some XML nodes matching a_i are ancestors of some nodes matching b_j. Therefore, in the pattern p equivalent to $p_1 \bowtie_{a_k.ID=b_l.ID} p_2$ (or in one of the patterns appearing in the equivalent pattern union), node a_i may appear *as an ancestor* of node b_j. In this case, we insert b_j in p *between* a_i and a_{i+1}. Of course, b_j preserves its children subtrees. Having decided on the relative order of a_i and b_j on the path from a_k to the root of the resulting pattern, we move one step upwards, and call the algorithm again, to decide the relative positions of nodes a_i and b_{j-1}. In this case, zipping advances one level up, both to a_{i-1} and to b_{j-1}. The resulting pattern(s) are gathered in res_1.

Similarly, if P_2 is not empty, then node a_i may appear underneath b_j in p (or one of the patterns in the equivalent pattern union, if a single pattern cannot be found). In a symmetric manner, a_i is inserted between b_j and b_{j+1} and the algorithm is called again to place a_{i-1} and b_j. The resulting patterns are gathered in res_2.

Finally, if P_3 is not empty, the nodes a_i and b_j may be unified in a single one in p (or one of the patterns in the equivalent pattern union). The resulting patterns are gathered in res_3.

The patterns in $res_1 \cup res_2 \cup res_3$ make up the result of the algorithm.

We can follow examples of Algorithm 1's execution on the patterns in Figure 3.

Algorithm 1: Compute the equivalent pattern corresponding to the join of p_1 and p_2 on $a_k.ID = b_l.ID$

```
Input : Pattern $p_1$, pattern $p_2$, pattern node $a_i$,
        pattern node $b_j$, pattern node $a_k$, pattern
        node $b_l$
Output: A set of patterns $\{r_1, r_2, \ldots, r_m\}$ such that
        $r_1 \cup r_2 \cup \ldots \cup r_m \equiv_S p_1 \bowtie_{a_i.ID=b_j.ID} p_2$
1  if $a_i = a_k$ and $a_j = b_l$ then
2      copy $a_k$ and its subtree in $p$; add a copy of $S'_j$ as $a_k$
       children in $p$; $\alpha(a_k, p) \leftarrow \alpha(a_k, p_1) \cap \alpha(b_l, p_2)$;
3      propagatePathComputations($p, a_k$);
4      return computeEquivPattern($p_1$, $p_2$, $a_{k-1}$, $b_{l-1}$, $a_k$,
       $b_l$);
5  else
6      $P_1 \leftarrow$ all paths from $\alpha(a_i, p_1)$ that are ancestors of a
       path from $\alpha(b_j, p_2)$;
7      $P_2 \leftarrow$ all paths from $\alpha(b_j, p_2)$ that are ancestors of a
       path from $\alpha(a_i, p_1)$;
8      $P_3 \leftarrow \alpha(a_i, p_1) \cap \alpha(b_j, p_2)$;
9      if $P_1 \neq \emptyset$ then
10         copy $p_1$ into a new pattern $p$;
11         insert $p_2.b_j$ between $p.a_i$ and $p.a_{i+1}$ connected
           with $p.a_i$ by the edge $e_{i-1}$ and with $p.a_{i+1}$ by
           the edge $e'_{j-1}$;
12         copy $S_j$ trees under $p.a_i$;
13         $\alpha(a_i, p) \leftarrow P_1$;
14         propagatePathComputations($p, a_i$);
15         $res_1 \leftarrow$
           computeEquivPattern($p, p_2, a_i, b_{j-1}, a_k, b_l$);
16     if $P_2 \neq \emptyset$ then
17         copy $p_2$ into a new pattern $p$;
18         insert $p_1.a_i$ between $p.b_j$ and $p.b_{j+1}$, connected
           with $p.b_j$ by the edge $e_{j-1}$ and with $p.b_{j+1}$ by
           the edge $e'_{i-1}$;
19         $\alpha(b_j, p) \leftarrow P_2$;
20         propagatePathComputations($p, b_j$);
21         $res_2 \leftarrow$
           computeEquivPattern($p, p_2, a_{i-1}, b_j, a_k, b_l$);
22     if $P_3 \neq \emptyset$ then
23         copy $S_j$ as children of $a_i$ in $p_1$;
24         $\alpha(p, a_i) \leftarrow P_3$;
25         propagatePathComputations($p_1, a_i$);
26         $res_3 \leftarrow$
           computeEquivPattern($p, p_2, a_{i-1}, b_{j-1}, a_k, b_l$);
27     return $res_1 \cup res_2 \cup res_3$;
```

- When computing $p_1 \bowtie_{b.ID=b.ID} p_2$ (recall the patterns in Figure 3), the zip algorithm will first gather the x and y children of the b node in the result. Then, the zip algorithm is called again on the a node from p_1 and the r node from p_2. We have $\alpha(p_1, a) = \{/r/a, /r/c/a, /r/g/a, /r/f/a\}$ and $\alpha(p_2, r) = \{/r\}$, thus $P_1 = \emptyset$, $P_2 = \alpha(p_2, b)$ and $P_3 = \emptyset$. Thus, a is inserted under r in the resulting pattern, and the zip algorithm is reinvoked on the r node in p_1 and the r node in p_2, which unifies them (lines 22-26) and the resulting pattern is p_3.
- When computing $p_1 \bowtie_{b.ID=b.ID} p_4$, after unifying the b nodes and cumulating their children, the zip algorithm is called on the a node from p_1 and the c node from p_2. Here, $\alpha(p_1, a) = \{/r/a, /r/c/a, /r/g/a, /r/f/a\}$, $\alpha(p_2, c) = \{/r/a/c, /r/c, /r/f/c, /r/g/c\}$, leading to: $P_1 = \{/r/a\}$, $P_2 = \{/r/c\}$, $P_3 = \emptyset$. This leads to the creation of two patterns, one (the future p_6) with the a node above the c node due to P_1, the other (the future p_7) with the c node above the a node. Each of these

Algorithm 2: ZipUnion

```
Input  : Pattern union $p_1^1 \cup p_1^2 \cup \ldots \cup p_1^m$, pattern
         union $p_2^1 \cup p_2^2 \cup \ldots \cup p_2^n$, pattern node $a_k$,
         pattern node $b_l$
Output: Pattern union $p^1 \cup p^2 \cup \ldots \cup p^r$ such that
         $p^1 \cup p^2 \cup \ldots \cup p^r \equiv_S (p_1^1 \cup p_1^2 \cup \ldots \cup$
         $p_1^m) \bowtie_{a_k.ID=b_l.ID} (p_2^1 \cup p_2^2 \cup \ldots \cup p_2^n)$
1 $res \leftarrow \emptyset$
2 foreach pattern $p_1 \in \{p_1^1, p_1^2, \ldots, p_1^m\}$ do
3     foreach pattern $p_2 \in \{p_2^1 \cup p_2^2 \cup \ldots \cup p_2^n\}$ do
4         $res \leftarrow res \cup$ computeEquivPattern($p_1$,$p_2$,$a_k$,
          $b_l$,$a_k$, $b_l$)
5 return $res$
```

patterns is completed by an extra invocation of the zip algorithm.

Proposition 3.1. The pattern union computed by Algorithm 2 is equivalent, under the path summary constraints, to the plan $p_1 \bowtie_{(a_k.ID=b_l.ID)} p_2$. □

The proof of proposition 3.1 is traced based on tree pattern containment under summary constraints [22].

Several simple extensions increase the generality of the zipping algorithm and we are going to briefly present them now:

First, ID joins can be directly extended to ID-based semijoins, nested joins, (nested) outerjoins etc. We only need to use the proper join operator in the algebraic plan, and assign to the edges connecting the b_j's children to their parents in the resulting pattern the correct annotation. Thus, optional edges reflect outerjoins, while nested edges reflect nested joins.

Second, the ID-based join can be replaced in a straightforward manner with a structural join, requiring that node a_k from p_1 be an ancestor/parent of node b_l from p_2. In this case, a_k and b_j are not unified as in lines 1-5, rather, the algorithm proceeds directly to zip a_k and b_{j-1}, since we know a_k must appear above b_j in the resulting pattern(s). Similar extensions deal with structural semijoins and outerjoins, and nested structural joins.

Finally, the zipping algorithm as defined above may only zip one pattern with another, however it can produce a union of patterns. The simple extension presented in Algorithm 2 handles the general case, when we need to zip one pattern union with another such union.

4. IMPLEMENTATION AND PERFORMANCE

We implemented XAMs and the algebraic optimization framework outlined in this paper as part of our ULoad [8] Java-based prototype. For structural constraints ULoad relies on XSum [9], a path summary library available at gemo.futurs.inria.fr/software/SUMMARY. Our measures were performed on a DELL Precision M70 laptop with a 2 MHz CPU and 1 GB RAM, running Linux Gentoo, and using JDK 1.5.0 from SUN.

Doc.	Shakespeare	Nasa	SwissProt	XMark11	XMark111	XMark233	DBLP02	DBLP05
Size	7.5MB	24MB	109MB	11MB	111MB	233MB	133	280MB
N	179690	476645	2977030	206130	1666310	4103208	3736406	7123198
$\|S\|$	58	24	117	536	548	548	145	159
$\|S\|/N$	$3.2{*}10^{-4}$	$5.0{*}10^{-5}$	$3.9{*}10^{-5}$	$2.4{*}10^{-3}$	$3{*}10^{-4}$	$1.3{*}10^{-4}$	$3.8{*}10^{-5}$	$2.2{*}10^{-5}$

Figure 5: Sample XML documents and their path summaries. N is the number of nodes in the original document and S is the number of nodes in the path summary.

All documents, patterns and summaries used are available at [29].

4.1 Path summaries and path annotations

A path summary is efficiently built during a single traversal of the document, in linear time, using $O(|PS|)$ memory [2, 17] where $|PS|$ is the summary size. Our implementation gathers 1 and + labels during summary construction, leading to $O(N + |PS|)$ time and $O(|PS|)$ memory complexity [2, 9].

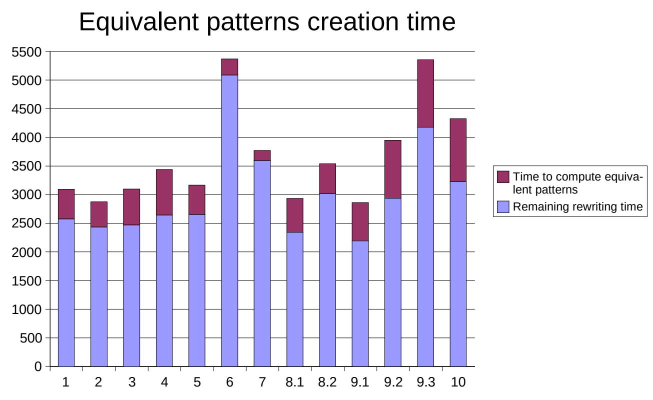
Summaries are very practical artifacts for query optimization and rewriting, and they are remarkably compact. Figure 5, borrowed from [9], illustrates this. We used a set of well-known XML documents, including various-sized XMark documents and two snapshots of DBLP, one from 2002, the other from 2005. For each document, N is its number of nodes, and $|PS|$ is its summary size. Summary sizes are very modest, and stabilize remarkably from one XMark/DBLP document to another, hinting to the fact that optimizers can easily cope with summary changes.

4.2 Plan combination

In this section, we show that computing equivalent patterns has a very moderate impact on the total effort spent optimizing a query by constructing algebraic plans *only.*

To that effect, we used the largest summary we could find for an XMark document, namely one with 548 nodes. We used the query patterns extracted from the XMark queries 1 to 10 as the target patterns. The view pattern set is initialized with 2-nodes views of the form $/site//X$, for each distinct tag X appearing in an XMark document. Each such view stores the ID and V (value) of the X-labeled nodes; the purpose of these initial views is to ensure that *some* rewritings will be found for every query.

Experimenting with various synthetic views, we noticed that large synthetic view patterns did not significantly increase the number of rewritings found, because the risk that the view has little, if any, in common with the query increases with the view size. Therefore, we added to the initial set of views 100 randomly-generated view patterns, based on the XMark summary. Each such view has 3 nodes (plus the root). 50% of the edges in these views are optional. 75% of the view nodes store IDs and values. We assume all IDs enable structural joins. No value predicates were added.



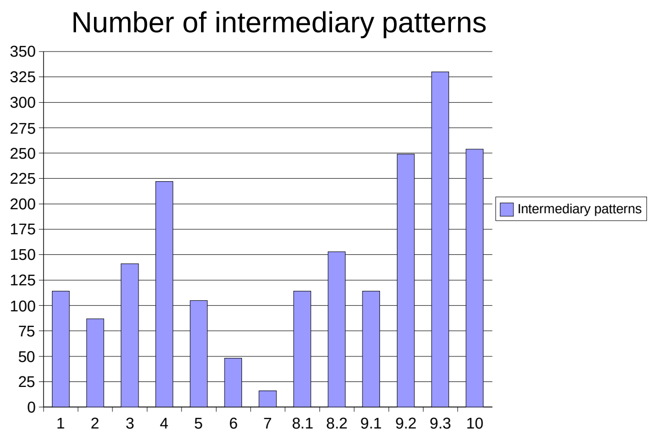


Figure 6: Query rewriting and the impact of equivalent pattern computation for XMark queries.

The graph at the left in Figure 6 shows the times (in **milliseconds**) to rewrite the XMark query patterns using all the 183 views described above. Multiple query patterns correspond to query 8 and 9, since the query performs a value join over several independent tree patterns [7]. For each query, we show the time needed to compute the equivalent patterns, as a part of the total rewriting time. At the right, we show the number of intermediary (plan,pattern) pairs that had to be constructed in the rewriting process for each query.

We remark that the time needeed to construct an equivalent pattern is moderate (at most 20% of the total time). The optimization process overall performs reasonably fast when considering the number of (plan, pattern) pairs explored. As previously mentioned, practical rewriting or optimization algorithms typically stop before exploring the full search space, while our tests explore it all.

We conclude that our algorithm for developing patterns in parallel with the development of algebraic plans is practically feasible and affordable in real-life XML optimizers.

5. CONCLUSIONS

We have described a method for producing, during algebraic XQuery optimization, tree patterns equivalent to increasingly larger algebraic plans. The benefit of producing such patterns is to bridge the gap between the seemingly disparate formalisms, and to enable translation of results on containment and equivalence under structural constraints, as well as twig cardinality estimation, directly to algebraic plans. Our approach is fully implemented in ULoad [8] and we have demonstrated its low overhead.

6. REFERENCES

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

[2] A. Aboulnaga, A. R. Alamendeen, and J.F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, 2001.

[3] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[4] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDBJ*, 11(4), 2002.

[5] S. Amer-Yahia and Y. Kotidis. Web-services architectures for efficient XML data exchange. In *ICDE*, 2004.

[6] A. Arion, V. Benzaken, and I. Manolescu. XML Access Modules: Towards Physical Data Independence in XML Databases. In *XIMEP Workshop*, 2005.

[7] A. Arion, V. Benzaken, I. Manolescu, Y. Papakonstantinou, and R. Vijay. Algebra-based identification of tree patterns in XQuery. In *Flexible Query Answering Systems (FQAS)*, 2006.

[8] A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: Choosing the right storage for your XML application (demo). In *VLDB*, 2005.

[9] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern XML databases. INRIA HAL report no. 1105, available at hal.inria.fr, 2005.

[10] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, 2004.

[11] M. Brantner, S. Helmer, C.-C. Kanne, and G.Moerkotte. Full-Fledged Algebraic XPath Processing in Natix. In *ICDE*, 2005.

[12] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[13] Z. Chen, H.V. Jagadish, L. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of XQuery. In *VLDB*, 2003.

[14] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, 2004.

[15] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. In *KRDB Workshop*, 2001.

[16] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *VLDB*, 2003.

[17] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, Athens, Greece, 1997.

[18] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *SIGMOD Conference*, pages 377–388, 1989.

[19] H.V. Jagadish, L.V. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *DBPL*, 2001.

[20] R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[21] M. Lee, H. Li, W. Hsu, and B. Ooi. A statistical approach for XML query size estimation. In *DataX workshop*, 2004.

[22] I. Manolescu, V. Benzaken, A. Arion, and Y. Papakonstantinou. Structured materialized views for XML queries. INRIA HAL report no. 1233, available at hal.inria.fr, 2006.

[23] I. Manolescu and Y. Papakonstantinou. XQuery in Midflight: Emerging database-oriented paradigms and a classification of research advances(tutorial). In *ICDE*, 2005.

[24] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[25] T. Milo and D. Suciu. Index structures for path expressions. *LNCS*, 1540, 1999.

[26] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.

[27] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity estimation for XML twigs. In *ICDE*, 2004.

[28] C. Ré, J. Siméon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, 2006.

[29] ULoad web site. gemo.futurs.inria.fr/projects/XAM.

[30] P. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.

[31] The XMark benchmark. www.xml-benchmark.org, 2002.

[32] W. Xu and M. Ozsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, 2005.

[33] The XQuery 1.0 language. www.w3.org/XML/Query.