Efficient XQuery Rewriting using Multiple Views

Ioana Manolescu [#1], Konstantinos Karanasos [#1], Vasilis Vassalos [*2], Spyros Zoupanos [#†1]

[#]*INRIA Saclay, France* [*]*AUEB, Greece* [†]*U. Paris IX, France*

[1]firstname.lastname@inria.fr [2]lastname@aueb.gr

***Abstract*— We consider the problem of rewriting XQuery queries using multiple materialized XQuery views. The XQuery dialect we use to express views and queries corresponds to tree patterns (returning data from several nodes, at different granularities, ranging from node identifiers to full XML subtrees) with value joins. We provide correct and complete algorithms for finding minimal rewritings, in which no view is redundant. Our work extends the state of the art by considering more flexible views than the mostly XPath 1.0 dialects previously considered, and more powerful rewritings. We implemented our algorithms and assess their performance through a set of experiments.**

I. Introduction

Query rewriting based on materialized views is a well-known performance enhancement technique in databases. While it was mostly used in relational databases [1], [2], query rewriting has recently received attention also in the context of XML databases, e.g. [3], [4], [5], [6], [7].

In this paper, we study the problem of rewriting queries from an expressive XQuery dialect, using *multiple views* from the same dialect. As in [3], [4], [5], [7], [8], [9], [10], we consider *equivalent rewritings*, which compute the same results as the original query, but *rely exclusively on the materialized views* (without accessing the original XML documents). In this context, given a set of views and a query, the task of evaluating the query can be split into three successive steps: (i) filter the view set to eliminate (as much as possible) those views which cannot be part of an equivalent query rewriting; (ii) find one or several rewritings of the query; (iii) through a process of logical and physical optimization, pick the rewriting which looks most promising, enumerate physical plans which may compute this rewriting, pick the best one and evaluate it.

Task (ii) above, i.e., query rewriting, has generally high computational complexity even for simple view and query languages. This is why step (i) is important: any reduction in the number of views to use during rewriting is likely to significantly impact the time and memory needs of the rewriting. In the literature, view filtering is typically performed by testing some form of embedding from each view into the query [3], [11]. More recently, [5] has proposed a highly efficient view filtering method, when the query and the views are expressed in XPath$^{\{/,//,*,[]\}}$. The filtering method is shown to perform well in practice, however, when it comes to step (ii), to avoid the high complexity of embedding tests in the presence of $*$ [12], the authors adopt some heuristics which in some cases make their proposed rewriting algorithm incomplete. Restricted to XPath$^{\{/,//,[]\}}$, the approach in [5] is complete and very efficient. Optimizing and executing a rewriting (task (iii)) is a problem in itself, whose solution needs to take into account parameters such as the available storage structure and access methods, including possible indices on the materialized views, available physical operators, data distribution etc. Parameters characterizing a solution to this problem may be a cost model or an optimization strategy. Some recent works [9], [10] focus on task (iii), providing efficient physical operators for view joins.

The focus in this work is on the core task (ii) above, that is: given a query and a set of views (which we assume already filtered), find all the possible equivalent rewritings of the query based on the views. Three main dimensions set our work apart from previous related works [3], [4], [5], [8], [9], [10].

First, we are only interested in *minimal* rewritings, i.e., those from which no view can be removed while still preserving the equivalence between the rewriting and the original query. We focus on minimal rewritings since regardless of the particular rewriting evaluation engine, a non-minimal rewriting will always entail more processing than a corresponding minimal one. Indeed, as the experiments of Section VII (Figure 9) show, the processing time difference between executing minimal and non-minimal rewritings is often very significant. To develop minimal rewritings only, we introduce a novel *bottom-up rewriting approach*, which builds partial rewritings by combining at every step, a smaller rewriting with an extra view. The combination either produces a rewriting over a bigger view set, or fails if it is non-minimal, i.e., if the new view was redundant with respect to the smaller rewriting.

Second, we consider a rich dialect of XQuery, where a single query (respectively, view) can return (respectively, store) information from *several variables*, and including *value joins*. In contrast, views in [3], [4], [5] have a single return node, and each view must store: the subtree rooted at its return node, the node identifier (or ID, in short), and (in [3]) the full label path to the node and other information. In our work, one can specify at various granularity levels what a view stores from each node, which again leads to more flexibility. Views and queries in [8] support group-by but not node IDs, which removes some rewriting opportunities while creating new ones.

Finally, our rewritings are expressed in a *generic logical XML algebra*, compatible with many well-established XQuery processing platforms (see, e.g., [13]). If the views are materialized as XML documents, our rewritings can directly be translated to XQuery statements, to be evaluated over the views by an off-the-shelf XQuery engine.

In this paper, we make the following contributions:
(1) We identify the problem of minimally rewriting XML queries of a rich, meaningful XQuery subset using multiple

views, characterize the size of the rewriting search space, and provide a complete algorithm for solving this problem. Our language includes FLWR expressions with value joins, XPath navigation and return of multiple values and subtrees, and extends in various ways languages considered in previous works [3], [4], [5], [6], as we detail in Section VIII. To make search as efficient as possible, we identify *left-deep query tree-organized rewritings* (LDQTs, in short), a tight subset of all the possible minimal rewritings, and focus only on finding LDQTs. We show that any other minimal equivalent rewriting can be obtained from a corresponding LDQT by the optimizer.
(2) As a sub-problem of query rewriting, we solve the question of computing the tree pattern obtained by joining two smaller tree patterns (if it exists). This extends the problem of XPath intersection [4] to more complex patterns (and corresponding XQuery queries). We explain the extra difficulties encountered in this context, and show how to solve them.
(3) We have fully implemented our algorithms and demonstrate their efficiency and large benefits for query execution in the presence of views through a series of experiments.

Our rewriting algorithm has exponential complexity in the total size of the query and views, which is expected since our work extends the language of [4] (which established co-NP hardness for XPath$^{\{/,//,[]\}}$ rewriting using views with IDs) to a richer query and view language. This is not a problem in practice, especially in view of the benefits for total query execution time, as shown in Section VII.

The paper is organized as follows. Section II outlines the rewriting problem we consider, and our solution to this problem, based on an example. The next Sections then detail our approach. Section III specifies the XQuery dialect we use for views, queries and rewritings, as well as the internal models used by our algorithm: tree patterns and algebra. For readability, we first describe our complete algorithm for building minimal rewritings in the restricted case when both the query and views consist of single tree patterns (no value joins across patterns) in Section IV. Section V focuses on a crucial problem at the core of this algorithm: computing the join of two expressive tree patterns with several return nodes. Section VI generalizes the algorithm from Section IV to handle our general language (with value joins). Section VII validates our approach through a set of experiments. We then provide more details on the related works, and we conclude.

II. Motivating Example

Our query rewriting approach is illustrated by the example depicted in Figure 1. The Figure shows two views v_1 and v_2, a query q, and three increasingly larger logical plans p_1, p_2, and p_3, such that p_3 is an equivalent rewriting of q using v_1 and v_2. The Figure shows the query and views as tree patterns, and the plans in an algebraic formalism. The details on their corresponding XQuery syntax and on the algebra will be provided in Section III.

Each solid single (double) edge in Figure 1 denotes parent-child (ancestor-descendant) relationships. The view v_1 stores the identifiers (or IDs, in short) and the contents (the full

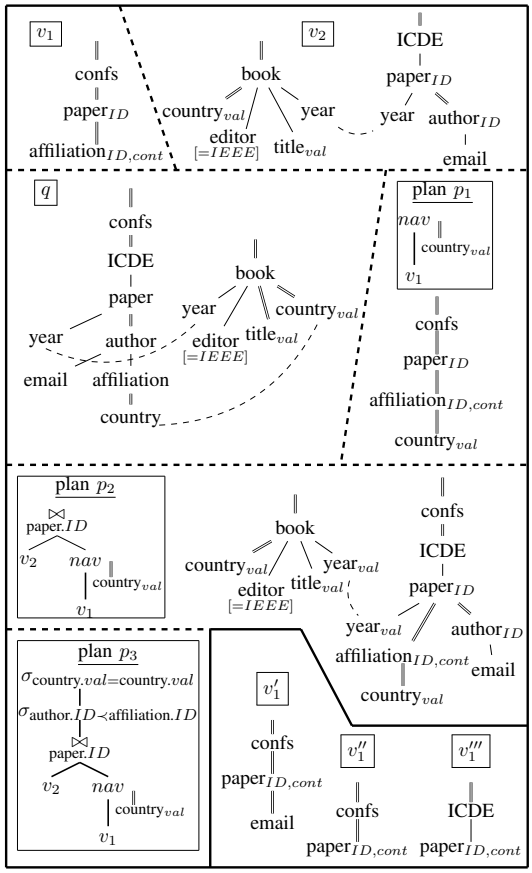

Fig. 1. Sample views, query, and algebraic rewritings.

XML subtrees, without including the IDs of the corresponding nodes, denoted *cont*) of the affiliations of all conference papers, along with the IDs of these papers. View v_2 stores the title and the publishing country of the books edited by IEEE and published the same year as an ICDE paper whose author must have an email, as well as the IDs of these papers and of their authors. We use the notation val to denote the string value of an element, obtained by concatenating all the text descendants of that element [14]. The predicate $[=IEEE]$ imposes that the val of the respective element should be equal to 'IEEE'. The dashed line connecting the two nodes labeled year in v_2 joins the two tree patterns, on the condition that the value of the year elements be the same on both sides. The query q asks for the title and country of books edited by IEEE and published the same year and in the same country as some author of an ICDE paper, for which an e-mail is specified.

We now show how we build an equivalent algebraic rewriting of q using v_1 and v_2. The query asks for the country in author/affiliation. Plan p_1 applies a *navigation* operator, denoted nav, on top of the view v_1, searching for descendants labeled country within the affiliation subtrees, and retaining their string values. Here, the small tree pattern //country$_{val}$ is a parameter of the algebraic nav operator. Underneath p_1, we show the tree pattern equivalent to this plan. Navigation has added the bottom country$_{val}$ node.

Plan p_2 joins the view v_2 with the plan p_1, on the paper ID. At its right, we show the pattern equivalent to this plan. Note

that the paper node has now three children (year, author, and affiliation). *As a side effect of the join* on paper ID, the parent of the paper node is labeled ICDE (as in v_2), whereas the confs node is pushed one level up, as an ancestor of the ICDE node. This is because the paper node can only have one parent, and v_2 specifies its label is ICDE. The confs node must then be an ancestor of the ICDE node in the join result. Such reasoning has first been made in [4], for a strict subset of our language (with only one return node and no value joins). In [5], when joining two XPath views on their target node ID, the order among the ancestors of the join node can be established by a *physical join operator* over the views, exploiting the expressive IDs they use. The join may return an empty result if the nodes belong to different paths, and this would only be detected at runtime. In our work, as in [4], we do such reasoning *statically* on the views, without accessing the view data.

We now discuss how to go from p_2 to the equivalent rewriting p_3. A first important remark is that in the pattern produced by the plan p_2, affiliation is a sibling of author, instead of being its child. Therefore, an *adaptation* is needed, materialized by the lower selection in p_3, namely $\sigma_{\text{author.ID}\prec\text{affiliation.ID}}$, where the $\prec$ symbol stands for a binary "isParentOf" predicate. We assume in this example that the IDs of author and affiliation are *structural*, that is, one can determine the structural relationships (parent or ancestor) between two nodes just by comparing their IDs. If the IDs were not structural, no equivalent rewriting of q using v_1, v_2 exists, since one cannot ensure the affiliation nodes from v_1 are children of the author nodes from v_2. (In cases not requiring structural predicates, we may obtain rewritings even based on simple IDs.)

Our rewriting algorithm then realizes that the query-specified join on year is already applied by v_2, whereas equality of the two country nodes still needs to be enforced. The rewriting is completed by applying the top selection of plan p_3 (and a final projection, not shown in Figure 1).

We now illustrate the differences between our work and the closest related ones [4], [5] using XPath views. Since these works do not handle value joins, for the purpose of the comparison, we restrict q to its leftmost tree pattern only. Moreover, as the other approaches do not support queries and views with multiple return nodes, we have to further simplify the query so that only one node is returned, e.g. the val of country nodes. Among the views which [4] and [5] may use to rewrite this restricted query are v_1', v_1'' and v_1''' shown at the bottom of Figure 1 (one could also copy under the paper nodes of v_1', v_1'' and v_1''' the subtrees of the paper query node, as existential branches). Observe that v_1', v_1'' and v_1''' store whole paper contents, which may be much larger than what the query needs. This drawback is due to the single return node in XPath 1.0, forcing the view to store the least common ancestor of all nodes from which some information is returned by the query. In contrast, v_1 and v_2 only store what the query needs. Moreover, [4] and [5] would produce a rewriting using all of v_1', v_1'' and v_1''', whereas our algorithm understands that v_1'' and v_1''' suffice for a minimal rewriting. A procedure to minimize non-minimal rewritings is sketched in [5], however, it relies on the special properties of the IDs they use.

1	$q :=$	for $absVar$ (, $(absVar\|relVar)$)* (where $pred$ (and $pred$)*)? return ret
2	$absVar :=$	x_i in doc(uri) p
3	$relVar :=$	x_i in x_j p // x_j *introduced before* x_i
4	$pred :=$	string(x_i) = (string(x_j) | c)
5	$ret :=$	$\langle l\rangle$ $elem$* $\langle/l\rangle$
6	$elem :=$	$\langle l_i\rangle\{$ (x_k | id(x_k) | string(x_k)) $\}\langle/l_i\rangle$

Fig. 2. Grammar for views and queries.

III. Views, Queries and Rewritings

We characterize the XQuery dialect we consider in Section III-A. We then present a joined tree pattern formalism used by our algorithm in Section III-B, and formalize the minimal query rewriting problem we address in Section III-C.

A. XQuery Dialect

Let $\mathcal{L}$ be a finite set of XML node names, and $\mathcal{XP}$ be the XPath$^{\{/,//,[]\}}$ language [12]. We consider views and queries expressed in the XQuery dialect described in Figure 2. In the for clause, $absVar$ corresponds to an absolute variable declaration, which binds a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the root of some document available at the URI uri. The non-terminal $relVar$ allows binding a variable named x_i to a path expression $p \in \mathcal{XP}$ to be evaluated starting from the bindings of a previously-introduced variable x_j. The optional where clause is a conjunction over a number of predicates, each of which compares the string value of a variable x_i, either with the string value of another variable x_j, or with a constant c.

The return clause builds, for each tuple of bindings of the for variables, a new element labeled l, having some children labeled l_i ($l, l_i \in \mathcal{L}$). Within each such child, we allow one out of three possible information items related to the current binding of a variable x_k, declared in the for clause: (1) x_k denotes the full subtree rooted at the binding of x_k; (2) string(x_k) is the string value of the binding; (3) id(x_k) denotes the ID of the node to which x_k is bound.

There are important differences between the *subtree* rooted at an element (or, equivalently, its *content*), its *string value* and its *ID*. The content of x_i includes all (element, attribute, or text) descendants of x_i, whereas the string value is only a concatenation of n's text descendants [14]. Therefore, string(x_i) is very likely smaller than x_i's content, but it holds less information. Second, an XML ID does not encapsulate the content of the corresponding node. However, XML IDs enable joins which may stitch together tree patterns into larger ones. Our XQuery dialect distinguishes IDs, value and contents, and allows any subset of the three to be returned for any of the variables, resulting in significant flexibility.

For illustration, Figure 3 shows the query q and the views v_1 and v_2 from Figure 1, in our XQuery dialect. The XQuery expression r corresponds to the algebraic plan p_3 of Figure 1, dictating how q can be answered using exclusively v_1 and v_2. The parent custom function returns true iff inputs are node IDs, such that the first identifies the parent of the second. Moreover, as usual in XQuery, the variable bindings that appear in the

q	for $p in doc("confs")//confs//ICDE/paper, $y1 in $p/year,
	$a in $p//author[email], $c1 in $a/affiliation//country,
	$b in doc("books")//book, $y2 in $b/year, $e in $b/editor,
	$t in $b//title, $c2 in $b//country
	where $e='IEEE' and $y1=$y2 and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{string($t)}⟨/tval⟩ ⟨/res⟩

v_1	for $p in doc("confs")//confs//paper, $a in $p/affiliation return ⟨v1⟩ ⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨acont⟩{$a}⟨/acont⟩ ⟨/v1⟩

v_2	for $b in doc("books")//book, $c in $b//country, $e in $b/editor,
	$t in $b/title, $y1 in $b/year, $p in doc("confs")//ICDE/paper,
	$y2 in $p/year, $a in $p//author[email]
	where $e='IEEE' and $y1=$y2
	return ⟨v2⟩ ⟨cval⟩{string($c)}⟨/cval⟩ ⟨tval⟩{string($t)}⟨/tval⟩
	⟨pid⟩{id($p)}⟨/pid⟩ ⟨aid⟩{id($a)}⟨/aid⟩ ⟨/v2⟩

r	for $v1 in doc("v1.xml")//v1, $p1 in $v1/pid, $af1 in $v1/aid,
	$c1 in $v1//acont//country, $v2 in doc("v2.xml")//v2,
	$c2 in $v2/cval, $t2 in $v2/tval, $p2 in $v2/pid, $a2 in $v2/aid
	where $p1=$p2 and parent($a2,$af1) and $c1=$c2
	return ⟨res⟩ ⟨tval⟩{$v2/tval}⟨/tval⟩ ⟨/res⟩

Fig. 3. Sample query, views, and rewriting.

where clauses imply the string values of these bindings (e.g. $e='IEEE' is implicitly converted to string($e)='IEEE').

Duplicates and order. The result of a view (or query) may include duplicates, e.g., the query for $a in doc("bib")//author, $l in $a/lastname return ⟨last⟩{$l}⟨/last⟩ in a database where an author has several publications. As per XQuery semantics, results follow the order of the bindings of the for variables, and are thus obtained in the order they appear in the query. For instance, in Figure 3, results of v_1 are ordered first by the $p bindings and then by $a bindings etc. Our rewriting preserves query semantics including such duplicates, and result order.

B. Joined Tree Patterns and Algebra

We use a dialect of joined tree patterns to internally represent views and queries, some examples of which appeared in Figure 1. Formally, a tree pattern is a tree whose nodes carry labels from $\mathcal{L}$ and may be annotated with zero or more among: ID, val and $cont$. A pattern node may also be annotated with a value equality predicate of the form $[=c]$ where c is some constant. The pattern edges are either simple for parent-child or double for ancestor-descendant relationships. A joined tree pattern is a set of tree patterns, connected through value joins, which are denoted by dashed edges.

As can be seen comparing Figures 1 and 3, the translation from our XQuery dialect to the joined tree patterns is quite straightforward. The only XQuery syntax aspect not reflected in the joined tree patterns is the name assigned to elements created by the return clause. This information is irrelevant to rewriting, therefore it is directly transmitted to the optimizer which will insert the appropriate element constructor operators on top of the rewritings we produce.

Tree pattern semantics. We start by considering the semantics of a single tree pattern. This can be defined in a standard way using tree embeddings as in, e.g., [15]. Instead, to better highlight the connection between views and rewritings, we equivalently define it using an algebra introduced in [7], and briefly recalled below.

Given a document d and label $a \in \mathcal{L}$, we denote by R_a^d and call *virtual canonical relation of a in d*, the list of tuples

$$s_{\text{paper}.ID,\text{affiliation}.ID}$$
$$\delta$$
$$\pi_{\text{paper}.ID,\text{affiliation}.ID,\text{affiliation}.cont}$$
$$\sigma_{\text{confs}.ID \prec\!\!\prec \text{paper}.ID \wedge \text{paper}.ID \prec\!\!\prec \text{affiliation}.ID}$$
$$R^d_{confs} \times R^d_{paper} \times R^d_{affiliation}$$

Fig. 4. Algebraic semantics of the tree pattern view v_1 from Figure 1.

of the form $(n.ID, n.val, n.cont)$ obtained from all the a-labeled nodes n in d. The tuples in R_a^d follow the order of appearance of the corresponding nodes in d. We denote by $\prec$ the *parent comparison operator*, which returns true if its left-hand argument is the ID of the parent of the node whose ID is the right-hand argument. Similarly, $\prec\!\!\prec$ is the *ancestor comparison operator*. Observe that $\prec$ and $\prec\!\!\prec$ are only abstract operators here (we do not make any assumption on how they are evaluated).

Algebra. Let $\mathcal{A}_0$ be the algebra consisting of the following operators: (1) scan of all tuples from a view v, denoted $scan(v)$ (or simply v for brevity, whenever possible), (2) cartesian product, denoted $\times$; (3) selection, denoted σ_{pred}, where $pred$ is a conjunction of predicates of the form $a \odot \underline{c}$ or $a \odot b$, a and b are attribute names, $\underline{c}$ is some constant, and $\odot$ is a binary operator among $\{=, \prec, \prec\!\!\prec\}$; (4) projection, denoted π_{cols}, where $cols$ is the attributes list that will be projected; (5) duplicate elimination (denoted δ); (6) sort, denoted s_{cols}, where $cols$ is the list of attributes defining the ordering. We also use joins, defined, as usual, as selections over $\times$.

The semantics of a tree pattern having k nodes, over a document d, is an $\mathcal{A}_0$ expression whose leaves are k virtual canonical relations, one for each tree pattern node. For illustration, Figure 4 depicts the semantics of the view v_1 from Figure 1, over a document d. The σ operator enforces the structural relationships between the nodes. The projection retains the attributes projected by query nodes, e.g. paper.ID, affiliation.ID and affiliation.$cont$. After duplicate elimination (δ), we sort the tuples in the order dictated by the IDs of the bindings of all nodes.

Joined tree pattern semantics. Let jt be a joined tree pattern over the tree patterns $t_1, t_2, \ldots, t_m$ and $vj_1, vj_2, \ldots, vj_k$ be its value joins. Each value join vj_i can be denoted as $t_{i_1}.n_1.val = t_{i_2}.n_2.val$, where vj_i connects node n_1 from the tree pattern t_{i_1} to the node n_2 from the tree pattern t_{i_2}, $1 \leq i_1, i_2 \leq m$. For each tree pattern t_i, $1 \leq i \leq m$, let t'_i be the tree pattern obtained from t_i by annotating with val each t_i node involved in a value join. Then, the semantics of jt is defined by the expression $\pi_{jt}(\sigma_{\bigwedge_{i=1,\ldots,k} vj_i}(t'_1 \times t'_2 \times \ldots \times t'_m))$, where π_{jt} is a projection retaining all the attributes of the original patterns $t_1, t_2, \ldots, t_m$.

C. Formal Problem Statement

We define an algebra $\mathcal{A} = \mathcal{A}_0 \cup \{nav\}$, where $\mathcal{A}_0$ is the algebra introduced in the previous Section, while navigation, denoted $nav_{a,np}$, is a unary algebraic operator, parameterized by one of its input columns' name a, and a tree pattern np. The name a must correspond to a $cont$ attribute. Let t be a tuple in the input of nav, and $np(t.a)$ be the result of

evaluating the pattern np on the XML fragment stored in $t.a$. Then, $nav_{a,np}$ outputs the tuples $\{t \times np(t.a)\}$, obtained by successively appending to t each of the tuples in $np(t.a)$. An example of navigation appeared in the plan p_1 of Figure 1. The plan p_1 outputs 4-attribute tuples: paper IDs, affiliation IDs and contents (coming from v_1), as well as country values (by virtue of the navigation).

Definition 3.1 (Equivalent rewriting): Given a set of views $\mathcal{V}$ and a tree pattern query q, an equivalent rewriting (or rewriting, in short) of q using $\mathcal{V}$ is an $\mathcal{A}$ expression e whose leaves are views from $\mathcal{V}$, such that for any document d, $e(d) = q(d)$, that is, evaluating e over d yields the same results as evaluating q over d.

Definition 3.2 (Minimal rewriting): A rewriting e of the query q using $\mathcal{V}$ is minimal if no other rewriting of q uses a proper subset of the view instances used in e.

Minimal rewritings should be distinguished from *min-size* rewritings, i.e., the (minimal) rewritings using the smallest possible number of views. Min-size rewritings do not always lead to the most efficient plans. For instance, the single view ($//\text{conf}_{ID,cont} \times //\text{book}_{ID,cont}$) suffices to answer the query q in Figure 1, yet it is very large and the rewriting based on v_1 and v_2 is likely to be much more efficient.

Therefore, the problem we consider is: *given a query q and a view set $\mathcal{V}$, find minimal $\mathcal{A}$ rewritings of q using $\mathcal{V}$*. The best minimal rewriting should be chosen by the optimizer.

IV. Rewriting Tree Pattern Queries

This Section presents our query rewriting algorithm in the case where the query and the views each correspond to a single tree pattern (Section VI will address the full language). From the XQuery syntax viewpoint, this restriction amounts to replacing rules 1 and 4 in the grammar of Figure 2 with:

$1'$	q :=	for $absVar$ (, $relVar$)* (where $pred$ (and $pred$)*)? return ret
$4'$	$pred$:=	string(x_i) =c

Rewriting algorithm overview The algorithm consists of two stages. First, it identifies all possible ways in which a view can be used to provide part of the query results. It may also apply some algebraic operations on each view, to better adapt it to the query. Possible rewritings using just one view are also identified in the first stage, and output. Moreover, this stage creates a set of *partial rewritings*: algebraic expressions resulting from view adaptation, which are not equivalent rewritings, but may serve as building blocks for producing them. The second stage combines partial rewritings by node ID equality joins (and, if IDs are structural, by structural joins as well). When building such a join, the algorithm may detect several reasons why the join result cannot be part of any equivalent rewriting; these join results are discarded immediately. Our algorithm ensures that throughout the search, *only minimal (partial or equivalent) rewritings are built*. This has two benefits:
(1) among all equivalent rewritings, the minimal ones are sure to have the lowest processing costs;
(2) enumerating non-minimal partial rewritings would lengthen the search.

Clearly, the search space of all join expressions built on top of the views is infinite. Section IV-A identifies a set of important classes of rewritings and shows that it suffices for our approach to be complete to consider the finite (and compact) set of LDQT rewritings. Section IV-B outlines our main rewriting algorithm, while Section IV-C discusses search strategies which it may use to determine which partial rewritings to combine. Finally, Section IV-D characterizes the size of the search space.

A. Left-Deep Query Tree Organized Rewritings (LDQT)

This Section makes two crucial observations on the space of partial or equivalent rewritings. Each observation leads to identifying a subset of all possible rewritings, and shows that it suffices to search within this class, while still preserving completeness.

We first observe that algebraic expressions (and in particular, rewritings of a query based on the same set of views) may differ in the details of their algebraic syntax, yet represent fundamentally the same (equivalent) rewriting. For instance, if $e = \sigma_{cond}(v_1 \bowtie (v_2 \bowtie v_3))$ is a rewriting, where p_1 is a predicate on the view v_1, then so is $e' = ((\sigma_{cond}(v_1) \bowtie v_2) \bowtie v_3)$. Our search for rewritings should not spend time enumerating candidate rewritings that can be obtained from one another by pushing σ and π operators, exploiting the transitivity of the $=$, $\prec$ and $\prec\!\!\prec$ comparison operators, reordering joins etc. Instead, such transformations should be left to the subsequent optimization stage, and rewriting should focus on finding fundamentally different alternatives.

Left-deep rewritings formalize this intuition:

Definition 4.1 (Left-deep rewriting): A rewriting e of the query q is *left-deep* iff: (i) all $\times$ operators in e are binary, and their right-hand children contain no $\times$ operator (e is a left-deep binary tree); (ii) all σ, π and δ are pushed as low as possible in e; (iii) all the nav operators are applied below any $\times$ operator.

From a rewriting e, one can obtain by algebraic transformations several left-deep rewritings, as illustrated in Figure 5, where for readability, we write $a \prec b$ instead of $a.ID \prec b.ID$ (and similarly for other predicates). Note also that here and in the sequel, we may omit drawing the edge above a top pattern node, whenever the discussion does not require it. In Figure 5, e' and e'' are left-deep rewritings obtained from e. We call the set of left-deep rewritings obtainable from a rewriting e via algebraic transformations, the *corresponding* left-deep rewritings of e.

Our second observation exploits the inner connection between tree patterns and algebraic operators. We first introduce:

Definition 4.2 (LDT rewriting): A left-deep rewriting is tree-organized (LDT rewriting, in short), iff each sub-plan p of e, such that p is a child of a $\times$ operator, is equivalent to some tree pattern t_p.

In Figure 5, e' is an LDT rewriting, since the leaf operators corresponding to scans of the views v_1, v_2 and v_3 are equivalent to the respective view tree patterns. Moreover, the plan

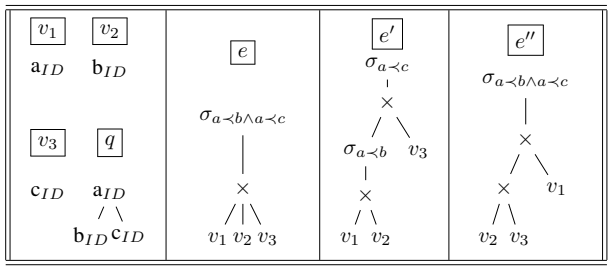

Fig. 5. Sample views, query, non-left-deep rewriting, and two corresponding left-deep rewritings.

$\sigma_{a\prec b}(v_1 \times v_2)$ is equivalent to the tree pattern //a_{ID}/b_{ID}. In contrast, e'' in Figure 5 is not an LDT rewriting: its sub-plan $v_1 \times v_2$ is a child of a $\times$ operator, and is not equivalent to any tree pattern, since it combines data from unrelated nodes.

We now state an important property:

Proposition 4.1: Let e be any rewriting of q using $\mathcal{V}$. Then, there exists an LDT rewriting e' corresponding to e.

The proof can be found in our technical report [16].

For example, in Figure 5, the rewriting e' is LDT and corresponds to e. Observe that e' does not need to be unique: swapping v_2 with v_3 and b with c in the predicates of e' in Figure 5 yields another LDT rewriting e''', in which all $\times$ children operators are equivalent to some tree pattern.

An important subset of LDT rewritings consists of:

Definition 4.3 (LDQT rewriting): An LDT rewriting e is query-tree-organized (LDQT in short) iff for each sub-plan p of e, such that p is a child of a $\times$ operator, and p is equivalent to the tree pattern t_p, t_p can be embedded into q.

For example, the LDT rewriting e' in Figure 5 is an LDQT rewriting, since the tree pattern equivalent to $\sigma_{a\prec b}(v_1 \times v_2)$ is the left branch of q. In this example, a sample LDT rewriting which is not LDQT, would be $\sigma_{b\prec a}(v_1 \times v_2)$, equivalent to the tree pattern //b_{ID}/a_{ID}, which cannot be embedded in q.

Proposition 4.2: Let e be a rewriting of q. There exists an LDQT rewriting e' corresponding to e.

The proof can be found in [16]. Observe that several LDQTs may correspond to a given rewriting, e.g., in Figure 5, e' and e''' mentioned above are LDQTs corresponding to e.

Proposition 4.2 is of crucial importance, as it allows us to *build and use only LDQT rewritings* during the rewriting. Any non-LDQT rewriting e can be derived by the optimizer from its corresponding LDQT rewriting e', by reversing the algebraic transformations which compute e' from e.

B. Main Rewriting Algorithm

Algorithm 1 (called **TPR**) outlines the two stages of our rewriting algorithm. Since it only builds LDQT rewritings, the algorithm works with (tree pattern, algebraic plan) pairs, such that in each pair, the plan and the pattern are equivalent. The algorithm starts by gathering an initial set S_0 of $(v, scan(v))$ pairs for each view $v \in \mathcal{V}$. Each view in such a pair may be embedded in multiple ways into the query. If a view node stores full subtrees (is annotated with *cont*) and the query requires evaluating sub-queries on these subtrees, we add the corresponding nav operators (line 4), following the one-view XPath rewriting approach of [6]. For example, in Figure 1,

Algorithm 1: Tree Pattern Rewriting (**TPR**)

Input : View set $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, query q
Output: A minimal algebraic rewriting of q using $\mathcal{V}$

1 $S_0 \leftarrow \{(v, scan(v))\}_{v\in\mathcal{V}}$; $S_1 \leftarrow \emptyset$; $S_{crt} \leftarrow \emptyset$
2 **for** $(v, scan(v)) \in S_0$ **do**
3 **for** ϕ *tree embedding,* $\phi : v \rightarrow q$ **do**
4 $ppe \leftarrow (v', nav(scan(v)), \phi)$
5 *//nav compensates v into v'*
6 **if** *the pattern of ppe is equivalent to q* **then**
7 output *the plan* of *ppe*; exit
8 add *ppe* to S_1
9 $S_{crt} \leftarrow S_1$
10 **while** *new triples are added to S_{crt}* **do**
11 pick $(t_1, p_1, \phi_1) \in S_{crt}$, $(t_2, p_2, \phi_2) \in S_1$
12 **for** $n_1 \in t_1, n_2 \in t_2, m \in q$, *with* $\phi_1(n_1)$=$\phi_2(n_2)$=m *and n_1, n_2 annotated with ID* **do**
13 $(t, p, \phi) \leftarrow (t_1, p_1, \phi_1) \bowtie^q_{n_1.ID=n2.ID} (t_2, p_2, \phi_2)$
14 **if** $(t, p, \phi) \neq \perp$ **then**
15 **if** $t \equiv q$ **then** output solution p; exit
16 **else** add (t, p, ϕ) to S_{crt}
17 **for** $n_1 \in t_1, n_2 \in t_2, m_1, m_2 \in q$ *such that m_1 is a /-parent (resp. //-parent) of m_2, $\phi_1(n_1) = m_1$, $\phi_2(n_2) = m_2$, n_1 and n_2 are annotated with ID* **do**
18 $(t, p, \phi) \leftarrow (t_1, p_1, \phi_1) \bowtie^q_{n_1.ID\prec n_2.ID} (t_2, p_2, \phi_2)$ (respectively, $(t, p, \phi) \leftarrow (t_1, p_1, \phi_1) \bowtie^q_{n_1.ID\prec\prec n_2.ID} (t_2, p_2, \phi_2)$)
19 **if** $(t, p, \phi) \neq \perp$ **then**
20 **if** $t \equiv q$ **then** output solution p; exit
21 **else** add (t, p, ϕ) to S_{crt}

the navigation plan p_1 compensates the view v_1 in order to rewrite the query q.

Afterwards (not shown in the Algorithm), we may add a value selection or a selection on node IDs to transform a // edge into a / edge, if the query requires it. If these transformations yield an one-view rewriting, we output it and exit; otherwise, we add the resulting *(pattern, plan, embedding)* triple to the set S_1. Note that such a triple defines a *view instance* that will be potentially used in the rewriting. The plan may contain σ and nav on top of a view scan, and the pattern is obtained from the view pattern correspondingly modified.

The **while** loop of Algorithm TPR seeks to produce an LDQT rewriting by combining a partial rewriting using one or more views, with a partial rewriting using one view. The combination is done via a join (either by ID equality, or by the structural condition $\prec$ or $\prec\prec$). Observe that we join *(pattern, plan, embedding)* triples, that is, if the join succeeds, we create: a new tree pattern t, a plan joining the plans p_1 and p_2 and producing exactly the same tuples as t, and an embedding ϕ from t to q. By construction, the plan is created only if it is still minimal, otherwise, the join fails (see Section V).

View joins at lines 13 and 18 are denoted $\bowtie^q$ (*query-driven* joins) to highlight the role of the query in deciding how to join the views. Remember that we are only interested in LDQT rewritings. Therefore, *the join of t_1 and t_2 must produce some t which can be embedded in q*, otherwise the join fails. If an LDQT rewriting is found (minimal by construction), Algorithm TPR returns it and exits. Section V provides further details on joining partial rewritings.

Notice also that pattern equivalence checks (lines 6, 15 and 20) rely on the PTIME containment algorithm of [12].

C. Rewriting Search Strategies

We define three variants of Algorithm TPR, each differing in the way it chooses the next pair of partial rewritings on which to attempt a query-driven join (line 11).

Naïve dynamic programming (NDP) first attempts to join partial rewritings over 1 view each; then, partial rewritings of 2 views with partial rewritings of 1 view etc. At step k, it joins rewritings of k views with rewritings of 1 view.

Query-driven dynamic programming (QDP) is similar to NDP. Nevertheless, to obtain candidate pairs of a k-views rewriting and a 1-view rewriting, QDP iterates *over the query nodes* and only if there is one ID-annotated node from each candidate mapped to a given query node, QDP attempts to join them. QDP does not try to join partial rewritings embedded in disjoint areas of the query, or rewritings with no ID to join on (which NDP attempts to do, and fails).

Query-driven depth-first (QDF): at any point, pick the partial rewriting whose embedding in the query reaches the largest number of query nodes, and seek to combine it with 1-view partial rewritings (those covering most query nodes first).

NDP and QDP find a rewriting of k views only after all rewritings of l views, $l < k$. To this end, they find min-size rewritings before all the others. However, if the smallest rewriting involves many views, NDP and QDP may take too long building all partial rewritings of sizes $1, 2, \ldots, k-1$. QDF uses the number of query nodes that the partial rewriting embeds to (or covers) as a hint to how many extra views must be joined, to obtain a query rewriting. This may lead QDF to finding a first minimal rewriting fast (if one exists).

D. Search Space Size

We have so far considered finding *one* minimal rewriting. To enumerate all rewritings, we remove the "exit" from lines 7, 15 and 20 of Algorithm TPR. This raises the question of whether (and when) the enumeration of minimal rewriting terminates.

Proposition 4.3: If IDs are structural, a minimal rewriting of q uses at most $|q|$ views, where $|q|$ is the number of nodes in q. If IDs are not structural, it may use at most $2 \times |q|$.

The proof can be found in [16]. Since LDQT rewritings are minimal, Proposition 4.3 entails that Algorithm TPR produces a number of rewritings bound by $O(|\mathcal{V}|^{|q|}/|q|!)$.

V. Query-driven Join of Partial Rewritings

This Section focuses on the problem of joining two (pattern, plan, embedding) triples. Formally, we denote the triples by (t_1, p_1, ϕ_1) and (t_2, p_2, ϕ_2), where t_1 is a tree pattern which ϕ_1 embeds into q (similarly for t_2 and ϕ_2), while p_1 and p_2 are LDQT partial rewritings such that t_1 and p_1 produce the same tuples (similarly for t_2 and p_2). Let $n_1 \in t_1$ and $n_2 \in t_2$ be two nodes such that $\phi_1(n_1) = \phi_2(n_2)$ (n_1 and n_2 map to the same query node) and n_1, n_2 are annotated with ID. How to build $(t, p, \phi) = (t_1, p_1, \phi_1) \bowtie^q_{n_1.ID=n_2.ID} (t_2, p_2, \phi_2)$?

Building $p = p_1 \bowtie_{n_1.ID=n_2.ID} p_2$ is straightforward. However, building the equivalent tree pattern t is not.

First, observe that the join may fail if there exists no tree pattern embeddable into q, and equivalent to p. For instance, let $v_1 = //a//b_{ID}$ and $v_2 = //c//b_{ID}$, $t_1 = v_1$ and $t_2 = v_2$, $p_1 = scan(v_1)$ and $p_2 = scan(v_2)$, $q = //a//c//b_{ID}$, and assume we join on the b nodes. No q subtree is equivalent to $p_1 \bowtie_{b.ID=b.ID} p_2$, because the a and c nodes have no common ancestor. In this case, the query-driven join of (t_1, p_1, ϕ_1) and (t_2, p_2, ϕ_2) is undefined.

Second, in some cases when no tree pattern embeddable into q is equivalent to p, one such pattern may still be obtained by an *adaptation* of p. For instance, let $v'_1 = //a_{ID}//b_{ID}$ and $v'_2 = //c_{ID}//b_{ID}$, and let $q = //a//c//b_{ID}$. Joining on the b nodes yields a plan p with no equivalent tree pattern. However, if IDs are structural, we can build $p' = \sigma_{a.ID \prec\!\!\prec c.ID}(p)$, which is an LDQT rewriting for q. The selection is inspired by the query, which specifies that a nodes are ancestors of c nodes. Thus, restricting p to p' does not lose results, and is getting closer to a rewriting. We formalize adaptation next.

Definition 5.1 (Query-driven join): Let (t_1, p_1, ϕ_1) and (t_2, p_2, ϕ_2) be two (pattern, plan, embedding) triples. Let $pred$ be a predicate of the form $n_1.ID = n_2.ID$, $n_1.ID \prec n_2.ID$ or $n_1.ID \prec\!\!\prec n_2.ID$, where $n_1 \in t_1$, $n_2 \in t_2$. The query-driven join $(t_1, p_1, \phi_1) \bowtie^q_{pred} (t_2, p_2, \phi_2)$, if it exists, is a triple (t, p, ϕ) such that:

1) t is a tree pattern which can be embedded in q, p is a plan using exclusively the plans p_1 and p_2, and $t \equiv p$
2) $p \subseteq p_1 \bowtie_{pred} p_2$, i.e., for any document d, $p(d) \subseteq p_1(d) \bowtie_{pred} p_2(d)$.
3) The embedding $\phi : t \to q$ agrees with both ϕ_1 and ϕ_2.
4) For any other triple (t', p', ϕ') satisfying conditions 1-3 above, $p' \subseteq p$ (p is *maximally contained* in $p_1 \bowtie_{pred} p_2$).

Condition 4 states that the algebraic component of the query-driven join should be restricted (by selections) only as much as needed to get an equivalent tree pattern embeddable in q. Further selections would constrain the partial rewriting more than the original query, disallowing the construction of an equivalent rewriting on top of this partial rewriting.

The next Sections describe our *zipping* algorithm for computing query-driven joins. For ease of explanation, we describe it for increasingly large subsets of the tree pattern language.

A. Zipping Linear Patterns

We start by considering linear path patterns with / and // edges, where the return node is annotated with ID and has no children. Algorithm 2 (**LZip**) computes the tree pattern component t of the query-driven join of the form $n_1.ID$=$n_2.ID$, for this case. LZip climbs up in parallel on the paths above n_1,

Algorithm 2: Linear path zipping (**LZip**)

Input : Linear paths t_1, t_2, return nodes $n_1 \in t_1$, $n_2 \in t_2$, such that $n_1.label$=$n_2.label$=l

Output: Tree pattern join of t_1 and t_2 on $n_1 = n_2$

1 $n \leftarrow node(l)$; $crt_1 \leftarrow n_1$; $crt_2 \leftarrow n_2$;
2 $par_1 \leftarrow$ the edge above crt_1 is $/$; $crt_1.\uparrow$;
3 $par_2 \leftarrow$ the edge above crt_2 is $/$; $crt_2.\uparrow$;
4 **while** *($crt_1 \neq \top$ or $crt_2 \neq \top$)* **do**
5 **if** $crt_1 = \top$ *(resp.* $crt_2 = \top$*)* **then**
6 copy the tree above crt_2 (resp. crt_1) on top of n
7 return the resulting pattern
8 **else if** par_1 *and* par_2 *(crt_1, crt_2 have same labels)* **then**
9 $n \leftarrow node(crt_1.label).addChild(n, /)$
10 $par_1 \leftarrow$ the edge above crt_1 is $/$; $crt_1.\uparrow$
11 $par_2 \leftarrow$ the edge above crt_2 is $/$; $crt_2.\uparrow$
12 **else if** par_1 *and* $!par_2$ **then**
13 **if** crt_1 *and* crt_2 *have the same label* **then**
14 **if** *the path above crt_2, including crt_2, embeds into the path above crt_1* **then**
15 copy the path above crt_1 on top of n
16 return the resulting pattern
17 **else**
18 $n \leftarrow node(crt_1.label).addChild(n, /)$
19 $par_1 \leftarrow$ the edge above crt_1 is $/$; $crt_1.\uparrow$
20 *// $!par_1$ and par_2 symmetrical to lines 12-19 above ...*
21 **else if** $!par_1$ *and* $!par_2$ **then**
22 **if** crt_1 *and* crt_2 *have the same label* **then**
23 $n \leftarrow node(crt_1.label).addChild(n, //)$
24 $par_1 \leftarrow$ the edge above crt_1 is $/$; $crt_1.\uparrow$
25 $par_2 \leftarrow$ the edge above crt_2 is $/$; $crt_2.\uparrow$
26 **else if** *the path above crt_2, including crt_2, (resp. crt_1, including crt_1) embeds into the path above crt_1 (resp. crt_2)* **then**
27 copy the path above crt_1 (resp. crt_2) on top of n; return the resulting pattern
28 return failure

respectively, above n_2 (thus the *zipping* name), and attempts to build the output pattern node by node, from the bottom up. The algorithm uses some notations: $node(a)$ builds a new pattern node of label a; $n.addChild(m, axis)$, where $axis$ is $/$ or $//$, adds the pattern m as a child of node n, connected to n by $axis$; $n.\uparrow$, when n is a variable bound to a pattern node, moves n to the node immediately above it in the pattern.

In the algorithm, $\top$ denotes a pattern root symbol. Initially, n is a fresh node with the label of n_1 and n_2. We attempt to build the path above n in the result tree pattern. There are four cases, depending on whether the edge above n_1 in t_1 and the one above n_2 in t_2, are labeled $/$ or $//$.

(1) Parent-parent: We unify the upper nodes of the two edges (guaranteed to have the same label by ϕ_1 and ϕ_2), since n can only have one parent.

(2) Parent-ancestor: (lines 12-19), and the symmetric **(3) ancestor-parent**: If crt_1 and crt_2 have the same label, say k, then we output only one k node, connected with a $/$ edge.[1] We now need to ensure that the ancestor conditions satisfied *both* by crt_1 in p_1 and crt_2 in p_2 are respected in the result. This is guaranteed only if the path above crt_2 in p_2 embeds in (is implied by) the path above crt_1 in p_1.

If crt_1 and crt_2 do not have the same label, we copy crt_1 as a $/$ parent of the output pattern, and move crt_1 a level up.

(4) Ancestor-ancestor: If the nodes have the same tag, we unify them and step up on both paths. Otherwise, we search for a path embedding in order to unify the patterns to the more specific one; if no embedding is found, zipping fails.

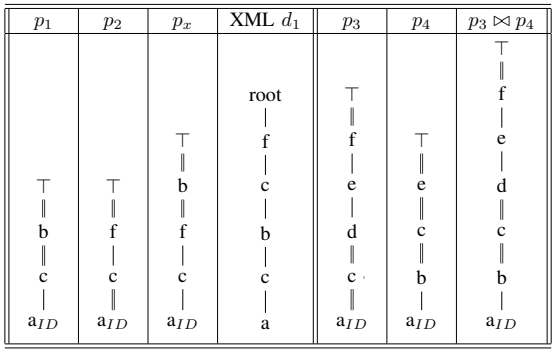

Fig. 6. Linear path zipping examples.

Figure 6 shows two linear zipping examples. Consider first the patterns p_1 and p_2. Algorithm LZip outputs an a node, then moves one edge up: c is a parent of a in p_1 but an ancestor in p_2. The path above c in p_2 does not embed into the path above c in p_1, therefore zipping fails, and indeed the tree pattern join is undefined. (Observe that the pattern p_x shown in the Figure is *not* the join of p_1 and p_2 on their c nodes. This is illustrated by the XML document d_1 at its right, which matches both p_1 and p_2, but does not match p_x.)

Now consider the patterns p_3 and p_4 in Figure 6. The algorithm produces an a node, then climbs to the b node in p_4 and adds a b on top of a to the output, then again climbs the c node in p_3 and adds a c in the output, then succeeds in embedding the path above the c in p_4 in the path above the c in p_3. Thus, it copies the path above c in p_3 on top of the nodes output so far, and exits having produced the pattern component of the query-driven join of p_3 and p_4.

Extension: structural joins. We now consider the case when node IDs, besides reflecting document order, are also *structural*, i.e., comparing $\mathsf{id}(n_1)$ and $\mathsf{id}(n_2)$ allows determining if n_1 is a parent (or ancestor) of n_2. Algorithm LZip can be easily extended to support *structural joins*, i.e. the tree pattern join of t_1 and t_2 on $n_1 \prec n_2$ or $n_1 \prec\!\!\prec n_2$. It suffices to start climbing up (lines 1-3) from $crt_1 = crt_2 = n_2$ in parallel:

[1] To see why we do not output *both* a k-labeled parent and a k-labeled ancestor above it, observe that such a pattern would be strictly included in the pattern obtained if we only output a k parent. The latter pattern would be included in, or equal to, $tp_1 \bowtie_{n_1.ID=n_2.ID} tp_2$. Therefore, the pattern with one k parent and one k ancestor does not satisfy the definition of the query-driven tree pattern join.

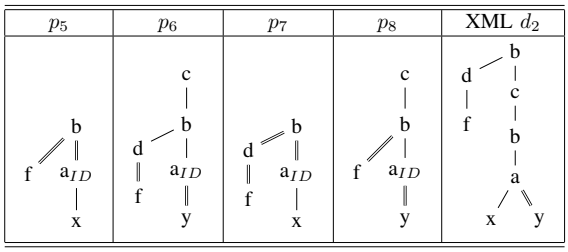

Fig. 7. Zipping XPath queries.

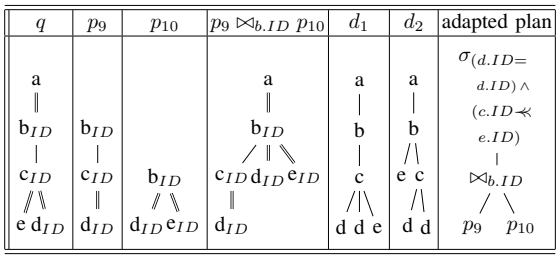

Fig. 8. Join adaptation example.

on the edge connecting n_2 to its original parent in t_2, and the new edge adding n_2 as a child of n_1.

At the XQuery level, rewritings with structural joins use two special user-defined functions parent($id1, $id2) and ancestor($id1, $id2), which return $true$ when the respective relationship holds. It is important to notice that structural identifiers enable some rewritings which would not be possible without. An example consists of the query $//a_{ID}//b_{ID}$ and the views $//a_{ID}$ and $//b_{ID}$. *In the sequel, we assume by default structural IDs.* If this is not the case, then whenever the success of an algorithm depends on the ability to apply a $\prec$ or $\prec\!\!\prec$ comparison, the algorithm fails.

Lemma 5.1: Algorithm LZip is correct and complete: it computes the query-driven join of p_1 and p_2, if one exists.

The proof is given in [16].

B. XPath Zipping

We now extend Algorithm LZip to XPath$^{\{/,//,[]\}}$ patterns.
(1) When producing the very first output node (line 1 in Algorithm LZip), add to this output node all the children of n_1 and all the children of n_2. For instance, in Figure 7, when joining p_5 with p_6, the children of both a nodes are added to the a node in the result.
(2) When Algorithm LZip looked for embeddings *between paths* above the crt nodes (lines 14 and 26), for XPath, we consider embeddings *from the sub-pattern of* p_1 *above* crt_1 into p_2. The sub-pattern of p_1 above crt_1 includes the path from t_1's root to crt_1 and the full sub-trees rooted in all nodes but crt_1, which appear on that path. For instance, in Figure 7, when joining p_5 and p_6, after producing an a node in the resulting pattern, we move up to the b nodes. The sub-pattern of p_5 above b consists of the p_5 nodes f and b. It can be embedded in p_6. Thus, the join of p_5 and p_6 on a is obtained by adding an x child to the a node of p_6.

Now consider p_7 and p_8 in Figure 7. Compared to p_5 and p_6, we swapped the left children of the b nodes. When joining p_7 and p_8 on a, after producing the a node, we cannot embed the subpattern above b in p_7 into p_8, thus the join fails.

We call the resulting algorithm **XZip (XPath zipping)**.

Lemma 5.2: Algorithm XZip is correct and complete. It computes the query-driven join of two XPath patterns p_1 and p_2, when one exists.

The proof is similar in spirit to the proof for LZip. It splits over the possible cases and shows that XZip moves from an initial graph pattern to a graph closer to a tree, and fails if and only if such a move is not possible.

For the simple languages considered so far, no join adaptation is needed: the algebraic plan of the query-driven join is exactly $p_1 \bowtie_{n_1.ID=n_2.ID} p_2$.

C. General Tree Patterns Zipping

We now extend Algorithm XZip to the general case, when any tree pattern node may be annotated with any subset of $\{ID, val, cont\}$. The changes required rely on two main observations. First, joining two general tree pattern views on n_1 and n_2 may lead to adding more join predicates on other IDs stored in the views. Second, the presence of stored attributes on several nodes may cause the join to fail, when the join introduces some subtle semantic "bugs", which cannot be "corrected". Examples will shortly illustrate this.

A first simple modification required by the presence of many attributes is that a node n annotated with ID (respectively, with val, or $cont$) can only be embedded to a node $\phi(n)$ (at lines 14 and 26 in Algorithm LZip) annotated in the same way, or, if a predicate of the form [=c] holds on n, it should also hold on $\phi(n)$. (Recall that Stage 1 of the overall rewriting algorithm may have added value selections on views).

Second, we may need to add *adaptation* predicates on top of the original join $p_1 \bowtie_{n_1.ID=n_2.ID} p_2$. An adaptation predicate of the form $n_x.ID$=$n_y.ID$ reflects the fact that the pattern produced by the query-driven join, must have only one node corresponding to both n_x and n_y, in order for the corresponding algebraic plan to be a partial rewriting. Similarly, predicates of the form $n_x.ID \prec n_y.ID$ (respectively, $n_x.ID \prec\!\!\prec n_y.ID$) must be added when the pattern produced by the query-driven join requires the node corresponding to n_x to be a parent (respectively, ancestor) of n_y.

Figure 8 illustrates adaptation. There are two "semantic bugs" in the simple join $p_9 \bowtie_{b.ID} p_{10}$. The first one can be seen on the document d_1 in Figure 8. Here, $q(d_1)$ returns two tuples, but $p_9 \bowtie_{b.ID} p_{10}$ returns 4 tuples, due to the two d children of the c element in d_1. The second is exemplified by the document d_2 in Figure 8: $q(d_2)$ should be empty since e is not a descendant of c in d_2, whereas $p_9 \bowtie_{b.ID} p_{10}$ returns again 4 tuples. The adapted plan at right in Figure 8 gives the solution. It adds two selection conditions over the join: the first unifies the two d nodes in the resulting pattern, while the second enforces that e is a descendant of c. This plan is the query-driven join of p_9 and p_{10}, and an LDQT rewriting of q.

Adaptation modifies Algorithm XZip as follows. When fusing nodes n_1 and n_2 (lines 9 and 23 in Algorithm LZip, on which XZip is built), *merge* their descendant forests by comparing their children pairwise and possibly fuse their

Algorithm 3: Joined tree pattern rewriting (**JTPR**)

Input : View set $\mathcal{JV} = \{v_1, v_2, \ldots, v_n\}$, query jq
Output: All minimal algebraic rewritings of jq based on $\mathcal{JV}$

1 $\mathcal{V} \leftarrow \cup_{v \in \mathcal{JV}} (\cup_{t \in v} v.t')$ //$v.t' \leftarrow extend(v.t)$
2 $L_{rw} \leftarrow \emptyset$, $L_{rw_t} \leftarrow \emptyset$
3 **for** $t \in jq$ **do**
4 $\quad L_{rw_t}(t) \leftarrow TPR(\mathcal{V}, t)$
5 $\quad$ **if** $L_{rw_t}(t) = \emptyset$ **then** exit
6 **for** $rw = (rw_{t_1} \times rw_{t_2} \times \ldots \times rw_{t_{n_q}})$, *such that* $rw_{t_i} \in L_{rw_t}(jq.t_i), 1 \leq i \leq n_q$ **do**
7 $\quad earlyPrune(rw, \mathcal{JV})$
8 $\quad$ **if** !$earlyPrune$ *and* $vjCheck(rw, jq)$ **then**
9 $\quad\quad adapt(rw)$; add rw to L_{rw}
10 $\quad$ **else** discard rw
11 return L_{rw}

respective subtrees by adding $=$, $\prec$ or $\prec\!\!\prec$ joins on node IDs. The merging process is recursive (traverses the children subtrees going down) and may fail for lack of (structural) IDs. For instance, if the e node in p_{10} was not annotated with ID, or if the ID is available but is not structural, it would have been impossible to ensure that e is a descendant of a c node. In this case, forest merging (and thus adaptation) fails, signaling that the query-driven join of p_9 and p_{10} cannot be built.

We call the algorithm obtained by these modifications to XZip, **Algorithm TPZip (tree pattern zipping)**.

Proposition 5.1: Algorithm TPZip is correct and complete: it produces the query-driven join of (t_1, p_1, ϕ_1) and (t_2, p_2, ϕ_2), whenever this exists.

The completeness of Algorithm TPZip implies that of Algorithm TPR:

Proposition 5.2: If a (minimal) rewriting exists, Algorithm TPR (calling TPZip at lines 13 and 18) will find one.

Proposition 5.3: Algorithm TPR is correct: it only outputs minimal equivalent rewritings (in the sense of Definition 3.2).

The proofs of the above propositions are presented in [16].

VI. Rewriting Joined Tree Pattern Queries

We now turn to the problem of rewriting a query in our full language, consisting of several tree patterns (or an XQuery) with value joins, using a set of views $\mathcal{JV}$ of the same kind. Let $jq = \pi_{jq}(\sigma_{jq}(t'_1 \times t'_2 \times \ldots \times t'_{n_q}))$ be a joined query with $\{t'_i\}_{1 \leq i \leq n_q}$ being the extended versions (annotating with val all t_i nodes involved in value joins) of its tree patterns, and $\mathcal{JV}$ comprise the views $\{v_i\}_{1 \leq i \leq n}$, where each v_i is of the form $\pi_{v_i}(\sigma_{v_i}(t_1^{v_i} \times t_2^{v_i} \times \ldots t_{n_{v_i}}^{v_i}))$. Let $e(v_{j_1}, v_{j_2}, \ldots, v_{j_k})$ be an equivalent algebraic rewriting of jq using the views v_{j_i}, where $1 \leq j_i \leq n$ for all $1 \leq i \leq k$. We have:

Proposition 6.1 (Joined-views rewriting): For each tree pattern t_i of jq, let t'_i be its extended version, annotating with val all t_i nodes involved in value joins. For every t'_i, there exists a (LDQT) rewriting, based only on the (extended versions of the) tree patterns of the views $v_{j_1}, v_{j_2}, \ldots, v_{j_k}$ involved in the rewriting e of jq.

The proof of this proposition can be found in [16].

Proposition 6.1 states that a rewriting for the joined query "encapsulates" rewritings for each individual query tree pattern. It is important to notice that *rewriting tree patterns with tree patterns only combines view data vertically, i.e. by equality or structural joins, whereas value joins connect data horizontally, across potentially different documents*.

This leads us to Algorithm 3 (called **JTPR**) for rewriting value-joined tree pattern queries. Due to space limitations, we present a simplified version of it here; an extended version, as well as more detailed comments, can be found in [16]. Section II has presented an example. The basic steps of the algorithm can be summarized as follows:

(1) Extend all tree patterns in jq and $\mathcal{JV}$ (line 1).

(2) Call Algorithm TPR to rewrite each extended query tree pattern with all the extended tree patterns from the $\mathcal{JV}$ views (lines 3-4).

(3) For each combination of rewritings output by TPR, two pruning steps are performed. The first guarantees that all the tree patterns of a specific joined view are present in the combination and the second that the value joins of the combination are less restrictive than those imposed by jq (lines 7-8).

(4) Transform the combination of rewritings to a valid equivalent rewriting of jq, after possibly adding some additional selections, projections etc. (line 9).

Algorithm JTPR treats each tree pattern like a relation: it rewrites them each in isolation, and then applies a second (separate) rewriting on top of the tree pattern rewritings. It uses a simple bucket-like [1] strategy for enumerating combinations of individual tree pattern rewritings. A more efficient strategy as in [2] could also be applied.

VII. Experimental Evaluation

Our algorithms have been implemented in Java, within our ViP2P platform (http://vip2p.saclay.inria.fr). ViP2P evaluates and stores materialized views in a store we built based on BerkeleyDB v3.3.75. To implement navigation, ViP2P relies on Saxon-B v9.1, which we also use as a baseline for query evaluation, as it has been shown to be an efficient in-memory processor [17]. ViP2P has its own algebraic optimizer and an execution engine including standard structural joins, holistic joins etc. The experiments were conducted on a 2.53GHz Intel Core 2 Duo machine with 4GB of RAM (2GB available for the JVM), running Mac OS X 10.6. All results are averaged over three runs.

A. Performance of View-Based Rewritings

We study the benefits of view-based rewriting, and of minimal rewritings in particular. Figure 9 plots query evaluation performance on an XMark [18] document of 100 MB. We use a set of queries derived from the XMark benchmark: four tree pattern (labeled *TPQ*) and three joined queries (*JTPQ*). We

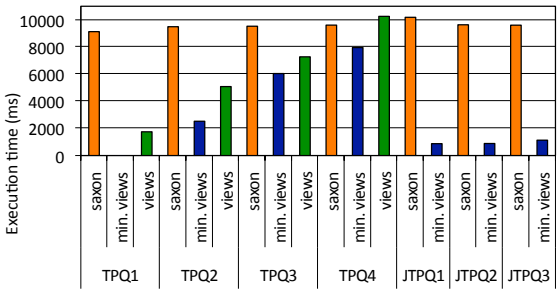

Fig. 9. View impact on query evaluation performance

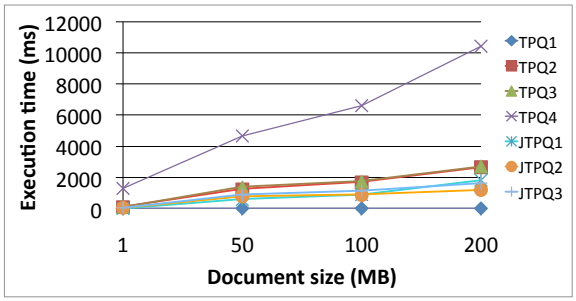

Fig. 10. Scalability of view-based query evaluation.

consider three scenarios: queries evaluated directly by Saxon (labeled *saxon*) on the document, minimally rewritten based on two to four views (*min. views*) and (for the tree pattern queries) non-minimally rewritten (*views*) using one or two more views than the minimal rewritings. The non-minimal rewritings are built as in [4] except for the usage of joins (instead of intersections) required by our more complex view language. We do not compare with the rewritings of [5] since they require special identifiers which we do not support.

Our view-based query evaluation outperforms Saxon by at least one order of magnitude in many cases. In particular, in the case of joined pattern queries, the benefits of using views are even greater, as these queries are by nature more demanding in evaluation time. In the case of TPQ4, our performance is close to Saxon's performance because one of the views used in our rewriting has a *content* node in which we navigate using Saxon's implementation. The evaluation of minimal rewritings is always faster than the evaluation of non-minimal rewritings.

Figure 10 shows the running times of the minimal rewritings found by Algorithms TPR and JTPR for the above queries and views and for XMark documents of increasing size (1 MB, 50MB, 100 MB and 200 MB). The Figure shows that the evaluation times of the rewriting plans grow linearly with the size of the data. This scalability is fundamentally due to the known good properties of the physical operators (hash join, structural join etc.) which ViP2P uses to translate the logical joins produced by the rewriting algorithms.

B. Rewriting Search Strategies

We now study the performance of query rewriting.

First, we study the impact of the three rewriting search strategies described in Section IV-C. We use a synthetic query q_{32} which is a full binary tree of 32 nodes, and four view sets. $\mathcal{V}_{31}$ has 31 views, one two-nodes view for each edge in q_{32}. $\mathcal{V}_6$ is a decomposition of q_{32} in six views having between 4 and 7 nodes. $\mathcal{V}_{8-4}$ has 8 views of 9 nodes each, but only 4 can be embedded to the query (line 3 in Algorithm TPR). Finally $\mathcal{V}_4$ is a decomposition of q_{32} in four fragments. For all but $\mathcal{V}_{8-4}$ (where only 4 views are needed for a rewriting), all the views are needed to form a rewriting.

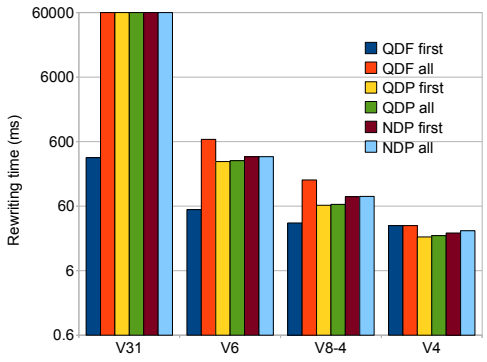

Fig. 11. Performance of rewriting strategies.

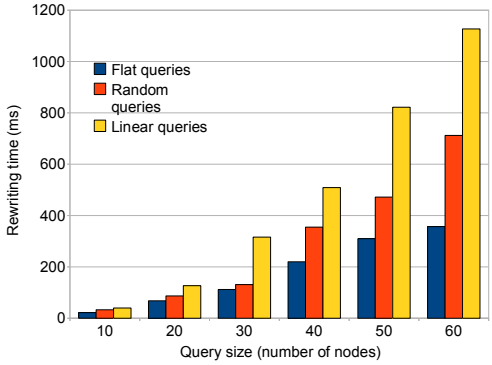

Fig. 12. Rewriting scaleup with the query size.

Figure 11 shows the time it takes to get the first, respectively, all minimal rewritings using the three search strategies. We stopped after 1 minute the runs which had not finished. Clearly, $\mathcal{V}_{31}$ is the toughest case, since any view subset can be joined together. The only reasonable time is the one of QDF to reach the first solution (approx. 300 ms). This shows that for large rewriting problems, finding all rewritings has prohibitive cost. NDP and QDP do as bad for finding all or just the first rewriting, whereas QDF remains of practical interest.

The results for V_6, $\mathcal{V}_{8-4}$ and $\mathcal{V}_4$ show the impact of the number of useful views on the running time. Observe that $\mathcal{V}_{8-4}$ takes less than $\mathcal{V}_6$ (as only 4 views are used), but longer than $\mathcal{V}_4$ because the views are slightly larger and query-driven joins are more complex. For small problems, NDP and QDP are competitive, since QDF needs to maintain more complex search structures. In general, however, QDF is more robust.

Second, we study the impact of the size and shape of the query on the total rewriting time. In Figure 12, we used a set of 60 views of one node each, and measured queries of n nodes, $n = 10, 20, \ldots, 60$. Each query required exactly n views to be rewritten. We used three families of queries: flat queries (one root with $n-1$ children), random queries (randomly generated trees of maximum fan-out 3), and linear queries (with only one leaf). We used the QDF strategy. Each query has only one rewriting. As expected, rewriting time increases with the

query size. However, the query shape also has an impact. Flat queries take less time to rewrite, since *there are less possible ways to join the views* (all joins must contain the root node). Rewriting linear queries takes longer since *any two views can be joined*, leading to an increased number of partial rewritings, whereas randomly generated queries are in-between. For 60 views the total time is quite high, however the first rewriting is found faster, and we do not expect practical rewritings to cover so many views.

VIII. Related Works

Several works have addressed XML query rewriting using views. *Maximally contained rewriting* is studied in [19]. *Equivalent rewriting using a single view* is considered in [6], [11], [20]. The works directly comparable to ours study *equivalent XML query rewriting using multiple materialized views* [3], [4], [5], [7], [8]. Unlike our work, [8] does not allow views to store IDs, which limits view join possibilities. Node IDs appear in the views in [3], [4], [5], [7]. The algorithms in [4], [7] do not require any special ID property, whereas [7] exploits also structural IDs if available. The rewriting algorithm in [3] requires that views store, next to each node ID, the complete label path from the document root to the node. Similarly, [5] requires an expressive class of IDs [21], encapsulating the labels (and IDs) of all the ancestors of the node. However, such expressive IDs are not available in all cases. Our algorithm can work with any type of ID, and exploits (but does not require) structural IDs when available. From this viewpoint, it most directly compares with [4], which shows that the rewriting problem for the XPath subset we consider in our Algorithm XZip is coNP-hard, and identifies restricted settings for which the problem is polynomial. The algorithm in [4] does not exploit structural IDs, does not guarantee minimality of the rewriting expressions and no experiments are provided. We note that the algorithms in [3], [5] do not provide completeness guarantees.

Concerning the view language, XPath 1.0 dialects in which a view may only store *ID and/or content for one node* are used in [3], [4], [5], [6], [19], [11], [20], while [9], [10] assume *only IDs are stored, and from all view nodes*. In contrast, views in [7] and in this work store IDs and/or values and/or contents for an arbitrary subset of view nodes, leading to more flexibility, but also making rewriting more complex. The rewriting algorithm of [7] requires structural knowledge about the database under the form of a Dataguide, which is not needed in this work.

Works complementary to ours [9], [10] describe efficient physical storage models for XML views, and efficient holistic twig join algorithms for joining views on IDs. Our focus here is on identifying logical rewriting plans, on which the physical optimization techniques of [9], [10] could also apply.

Closely related problems studied in XML databases are materialized view selection [22], query containment [12], [23] and minimization [15].

Early elements of this work were informally presented in [24], whereas our algorithms were demonstrated in [25].

IX. Conclusion

XQuery performance problems can be addressed by exploiting pre-computed results under the form of materialized views. This work has considered equivalent view-based rewriting of queries expressed in a rich XQuery dialect using multiple views, extending the languages previously considered in several respects. Moreover, we focused on finding only minimal rewritings, since they are the ones of interest in practice. We have presented a correct and complete query rewriting algorithm, compared several search strategies, and found QDF to be most efficient, especially when using many views. Finally, we have demonstrated through experiments the practical interest of our views and rewriting plans.

We plan to extend our algorithm to nested patterns, representing several nested XQuery FLWR blocks [7]. We are also extending our optimizer with tree pattern cardinality estimations, re-using the equivalent patterns developed for rewriting plans during the rewriting process.

References

[1] A. Y. Levy, A. Rajaraman, and J. J. Ordille, "Querying heterogeneous information sources using source descriptions," in *VLDB*, 1996.
[2] R. Pottinger and A. Y. Levy, "A scalable algorithm for answering queries using views," in *VLDB*, 2000.
[3] 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.
[4] B. Cautis, A. Deutsch, and N. Onose, "XPath rewriting using multiple views: Achieving completeness and efficiency," in *WebDB*, 2008.
[5] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong, "Multiple materialized view selection for XPath query rewriting," in *ICDE*, 2008.
[6] W. Xu and M. Ozsoyoglu, "Rewriting XPath queries using materialized views," in *VLDB*, 2005.
[7] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou, "Structured materialized views for XML queries," in *VLDB*, 2007.
[8] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola, "Rewriting nested XML queries using nested views," in *SIGMOD*, 2006.
[9] D. Chen and C.-Y. Chan, "ViewJoin: Efficient view-based evaluation of tree pattern queries," in *ICDE*, 2010, pp. 816–827.
[10] D. Phillips, N. Zhang, I. F. Ilyas, and M. T. Özsu, "InterJoin: Exploiting indexes and materialized views in XPath evaluation," in *SSDBM*, 2006, pp. 13–22.
[11] B. Mandhani and D. Suciu, "Query caching and view selection for xml databases," in *VLDB*, 2005.
[12] G. Miklau and D. Suciu, "Containment and equivalence for a fragment of XPath," *J. ACM*, vol. 51, no. 1, 2004.
[13] T. Grust, H. V. Jagadish, F. Özcan, and C. Yu, "XQuery processors," in *Encyclopedia of Database Systems*, 2009.
[14] "XPath Functions and Operators," www.w3.org/TR/xpath-functions, 2007.
[15] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava, "Tree pattern query minimization," *VLDB J.*, vol. 11, no. 4, 2002.
[16] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos, "Efficient XQuery Rewriting using Multiple Views," http://vip2p.saclay.inria.fr/papers/xq-rw.pdf, 2010, Technical Report.
[17] P. Michiels, I. Manolescu, and C. Miachon, "Toward microbenchmarking xquery," *Inf. Syst.*, vol. 33, no. 2, 2008.
[18] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A benchmark for XML data management," in *VLDB*, 2002.
[19] L. V. S. Lakshmanan, H. Wang, and Z. J. Zhao, "Answering tree pattern queries using views," in *VLDB*, 2006.
[20] L. H. Yang, M.-L. Lee, W. Hsu, and S. Acharya, "Mining frequent query patterns from XML queries," in *DASFAA*, 2003.
[21] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen, "From region encoding to extended Dewey: On efficient processing of XML twig pattern matching," in *VLDB*, 2005.
[22] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, and P. A. Boncz, "Materialized view selection in XML databases," in *DASFAA*, 2009.
[23] I. Tatarinov and A. Y. Halevy, "Efficient query reformulation in peer-data management systems," in *SIGMOD*, 2004.
[24] I. Manolescu and S. Zoupanos, "XML materialized views in P2P," DataX workshop (not in the proceedings), 2009.
[25] K. Karanasos and S. Zoupanos, "Viewing a world of annotations through AnnoVIP (demo)," in *ICDE*, 2010.