Materialized View Selection for XQuery Workloads

Asterios Katsifodimos
Inria Saclay &
Université Paris-Sud
Île-de-France, France
asterios.katsifodimos@inria.fr

Ioana Manolescu
Inria Saclay &
Université Paris-Sud
Île-de-France, France
ioana.manolescu@inria.fr

Vasilis Vassalos
Athens University of
Economics and Business
Athens, Greece
vassalos@aueb.gr

ABSTRACT

The efficient processing of XQuery still poses significant challenges. A particularly effective technique to improve XQuery processing performance consists of using materialized views to answer queries. In this work, we consider the problem of choosing the best views to materialize within a given space budget in order to improve the performance of a query workload. The paper is the first to address the view selection problem for queries and views with value joins and multiple return nodes. The challenges we face stem from the expressive power and features of both the query and view languages and from the size of the search space of candidate views to materialize. While the general problem has prohibitive complexity, we propose and study a heuristic algorithm and demonstrate its superior performance compared to the state of the art.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

XML, Materialized Views, View Selection

1. INTRODUCTION

The efficient processing of XML queries raises many challenges, due to the complex and heterogeneous XML structure, and on to the complexity of the W3C XQuery language. XQuery is Turing-complete, thus many performance-enhancing works focused on accelerating the processing of a central language subset, typically consisting of tree patterns. Performance enhancing techniques include efficient tree pattern evaluation algorithms [1, 2], new physical operators such as the Holistic twig join [3] and its improved variants, query simplification and minimization [4], algebraic optimization etc. To speed up XML data access, previous research has focused on building efficient stores, exploiting XML node identifiers encapsulating useful structural information, as well as building XML summaries and indices, with DataGuides [5] and $D(K)$ indices [6] being among the best-known proposals.

Materialized views have improved performance by orders of magnitude in relational databases [7, 8, 9], and they raised interest also in the context of XML databases [10]. The problem of rewriting an XML query using one view has been extensively studied e.g., in [11, 12, 13], and using several views, e.g., in [14, 15, 16, 17]. A dual problem to view-based rewriting is the automated selection of materialized views to improve the performance of a given XQuery workload. Well-studied for relational databases [7, 8], it has also attracted attention for XML queries and views [11, 13, 18, 19].

In this work, we consider the problem of selecting a set of views to be materialized within a given space budget S, in order to minimize the processing costs associated to a given query workload Q. We consider queries and views expressed in a large subset of XQuery, consisting of conjunctive tree patterns (using the child and descendant axis and existential branches) return data from several nodes, connected with value joins. Following [14, 16, 17, 20, 21], our views (and queries) are also allowed to store XML node identifiers, which enable interesting view joins and potentially more efficient rewritings. We picked this language since it is among the most expressive for which multiple-views equivalent query rewriting algorithms are known. Specifically, we rely on the rewriting algorithm of [16] which, given a set of views $V = \{v_1, v_2, \ldots, v_n\}$ materialized over a database D and a query q, returns the equivalent rewritings of q using the views in V. Each such rewriting is *complete*, in the sense that the database is no longer needed in order to evaluate the queries. A rewriting is expressed in a tuple-based XML algebra, to be detailed further on.

The space budget may not suffice to store views based on which *all* workload queries can be completely rewritten. In this the case, our focus is to identify a subset of the workload, and recommend views based on which this query subset can be rewritten, in a way that minimizes the overall workload cost (knowing that the remaining queries will still have to be evaluated in the database).

Assuming that each query $q_i \in Q$ is associated a weight $w_i \geq 0$ (for instance, reflecting the query frequency), the view selection problem we consider is: find the set of materialized views V_{best} such that (i) the V_{best} views, together, fit in the space budget, and (ii) the weighted sum of the costs for processing all workload queries, either through rewritings based on V_{best} views or directly from the documents, is the smallest that can be attained by any view set satisfying (i), that is, fitting in the space budget.

In this work, we make the following contributions:

- We are the first to formalize the problem of materialized view selection for the expressive tree pattern query with value joins dialect we consider. We show that the space of potential candidate views makes complete exploration unfeasible and present several effective candidate pruning criteria.
- We leverage an existing query rewriting algorithm [16] to propose two view selection algorithms: a benefit-oriented greedy algorithm named UDG, reminiscent of previous algorithms [11, 19], and a state search-based algorithm named

1	q :=	for $absVar$ (, $(absVar \mid relVar)$)* (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)) $\} \langle /l_i \rangle$

Figure 1: Grammar for views and queries.

ROA, exploring many view set transformations and including a randomized component.

- We compare UDG and ROA with their closest competitors from the literature [11, 19]. Our experiments show that ROA scales well beyond the algorithms of [19] and our UDG. While ROA is slower than the algorithm of [11], we show that it consistently recommends view sets leading to lower processing costs. This is because ROA considers many-views rewritings, and exploits the full spectrum of rewriting possibilities our query and view language enable.

The remainder of the paper is organized as follows. Section 2 outlines our views, queries, and rewritings, while Section 3 discusses candidate view sets. Section 4 presents our view selection algorithms, while Section 5 details the closest competitors we compare with. Section 6 describes our view selection experiments. We discuss other related works in Section 7, then we conclude.

2. VIEWS, QUERIES AND REWRITINGS

We characterize the XQuery dialect we consider in Section 2.1. Section 2.2 presents a joined tree pattern formalism, conveniently representing queries, while Section 2.3 describes our rewritings based on views and the database.

2.1 XQuery dialect

Let $\mathcal{L}$ be a set of XML node names, and $\mathcal{XP}$ be the XPath$^{\{/,//,[]\}}$ language [22]. We consider views and queries expressed in the XQuery dialect described in Figure 1. 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 [23]. 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. *We assume structural IDs*, i.e., comparing the IDs $id(n_1)$ and $id(n_2)$ allows determining if n_1 is a parent (or ancestor) of n_2. Our XQuery dialect distinguishes structural IDs, value

```
q   for $p in doc("confs")//confs//SIGMOD/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='ACM' and $y1=$y2 and $c1=$c2
    return ⟨res⟩ ⟨tval⟩{string($t)}⟨/tval⟩ ⟨/res⟩

v1  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⟩

v2  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")//SIGMOD/paper,
        $y2 in $p/year, $a in $p//author[email]
    where $e='ACM' 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⟩ ⟨/res⟩
```

Figure 2: XQuery query, views, and rewriting.

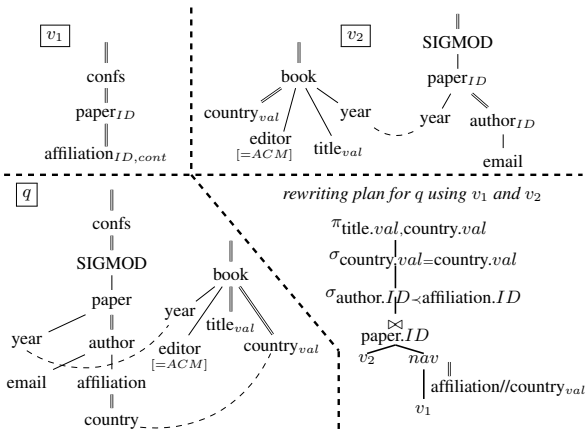

Figure 3: Pattern query and views, and algebraic rewriting.

and contents, and allows any subset of the three to be returned for any of the variables, resulting in significant flexibility.

Figure 2 shows a query q in our XQuery dialect, as well as two views v_1 and v_2. The parent custom function returns true if its 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 where clauses imply the string values of these bindings (e.g. $e='ACM' is implicitly converted to string($e)='ACM').

2.2 Joined tree patterns

We use a dialect of joined tree patterns to represent views and queries. 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. For illustration, Figure 3 depicts the (joined) tree pattern representations of the query and views shown in XQuery syntax in Figure 2. In short, the semantics of an annotated tree pattern against a database is a list of tuples storing the ID, val and $cont$ from the tuples of database nodes in which the tree pattern embeds. The tuple order follows the order of the embedding target nodes in the database. The detailed semantics feature some duplicate elimination and projection operators (from the algebra we will detail next), in order to be as close to the W3C's XPath 2.0 semantics as possible. The only remaining difference is that tree patterns return tuples, whereas standard XPath/XQuery semantics uses node

op

$author_{ID}$	$author_{cont}$
$id_{author,1}$	⟨author⟩⟨bk⟩bk1⟨/bk⟩⟨/author⟩
$id_{author,2}$	⟨author/⟩
$id_{author,3}$	⟨author⟩⟨bk⟩bk2⟨/bk⟩⟨bk⟩bk3⟨/bk⟩⟨/author⟩

$\Rightarrow$

$nav_{author_{cont},//bk_{cont}}(op)$

$author_{ID}$	$author_{cont}$	bk_{cont}
$id_{author,1}$	⟨author⟩⟨bk⟩bk1⟨/bk⟩⟨/author⟩	⟨bk⟩bk1⟨/bk⟩
$id_{author,3}$	⟨author⟩⟨bk⟩bk2⟨/bk⟩⟨bk⟩bk3⟨/bk⟩⟨/author⟩	⟨bk⟩bk2⟨/bk⟩
$id_{author,3}$	⟨author⟩⟨bk⟩bk2⟨/bk⟩⟨bk⟩bk3⟨/bk⟩⟨/author⟩	⟨bk⟩bk3⟨/bk⟩

Figure 4: Sample input and output to a logical nav operator.

lists. Algebraic operators for translating between the two are by now well understood [24]. The semantics of a joined tree pattern is the join of the semantics of its component tree patterns.

Translating from our XQuery dialect to the joined tree patterns is quite straightforward. The only part of the XQuery syntax *not* reflected in the joined tree patterns is the names of the elements created by the return clause. These names are not needed when rewriting queries based on views. Once a rewriting has been found, the query execution engine creates new elements out of the returned tuples of XML elements, values and/or identifiers, using the names specified by the original query, as explained in [25]; we will not discuss this further. From now on, for readability, we will only use the tree pattern query representations of views and queries.

2.3 Rewritings, algebra and costs

A rewriting is an XQuery query expressed in the same dialect as our views and queries, but formulated against XML documents corresponding to materialized views. For instance, the rewriting XQuery expression r in Figure 2 is an equivalent rewriting of the query q using the views v_1 and v_2 in the same Figure.

An alternative more convenient way to view rewritings is under the form of *logical algebraic plans*. Before presenting plans, we introduce some useful logical operators. 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).

We consider an algebra on tuple collections (such as described in the previous Section) whose main operators are: (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 tuple attributes, $\underline{c}$ 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) navigation, denoted $nav_{a,np}$. nav 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 in the input of nav. 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 $\{t \underset{a}{\bowtie} np(t.a)\}$.

Figure 4 illustrates the functioning of nav on a sample input operator op. The parameters to this nav are $author_{cont}$ (the name of the column containing ⟨author⟩ elements), and the tree pattern $//bk_{cont}$. The first tuple output by nav is obtained by augmenting the corresponding input tuple with a bk_{cont} attribute containing the single bk-labeled child of the element found in its $author_{cont}$ attribute. The second and third nav output tuples are similarly obtained from the last tuple produced by op. Observe that the second tuple in op's output has been eliminated by the nav since it had no ⟨bk⟩ element in its $author_{cont}$ attribute.

The algebra also includes the join operator, defined as usual, sort and duplicate elimination. For illustration, in the bottom of Figure 3, we depict the algebraic representation of the rewriting r shown in XQuery syntax at the bottom of Figure 2.

Coupling with a cost-based optimizer As stated in the Introduction, as part of the input to our problem, we assume available a function $size(v)$ returning the space occupancy of a view v, and a function $cost(q|_V)$ returning the cost to evaluate a query q based on the view set V. The latter is the cost of evaluating q's rewriting based on V if such a rewriting exists, otherwise, the cost of evaluating q directly on the database.

The question arising next is: which rewriting of q using V to consider in the $cost$ function? Indeed, there may be several. For example, consider the query q: $/a/b_{val}$ and the views v_1: $/a_{id,cont}$ and v_2: $//b_{id,val}$. Query q can be rewritten by (i) navigating in a_{cont} of v_1 to extract b_{val} or; (ii) performing a structural join of views v_1 and v_2 on the IDs of a and b and finally projecting b_{val} of v_2. The evaluation costs of these rewritings may be different.

To get a well-defined notion of cost, we assume available an *algebraic cost-based optimizer*, whose cost function is the same as the one used by our view selection framework. The optimizer is pipelined at the output of the query rewriter: for each algebraic rewriting r of a query q found by the rewriter, the optimizer applies logical and physical plan transformations, looking for the most efficient way to evaluate r, such that the $cost(q|_V)$ function returns the *minimum physical evaluation cost* among all the minimal equivalent rewritings of q using V. The cost is an aggregated measure of I/O and CPU costs. We term the q having the minimum physical cost the *best* rewriting of q using V, and by "rewriting of q using V" we will always designate the best rewriting.

Since computing actual view sizes or query processing costs is too costly, we (and the optimizer) rely as usual on corresponding *estimation functions*. Thus, $size^\epsilon(v, D)$ estimates the size of view v on a database D, while $cost^\epsilon(q|_V)$ and $cost^\epsilon(Q|_V)$ estimate the costs of processing a query (resp. a workload) assuming a set of materialized views V. Details on the actual estimation functions used in our implementation will be given in Section 6.1.

Due to the estimation error of $size^\epsilon$, a recommended view set whose size was estimated under S may in fact occupy more than S. To guard against this, one could run the view selection with a budget of $(1 - f) \times S$, for some small f between 0 and 1. In our experiments, this problem did not occur, but in some cases it might.

Concerning $cost^\epsilon$, we make a few assumptions that are well-supported by the existing XML processing literature. First, each physical operator has a positive cost. Second, the cost of each physical operator (such as view scan, hash join, holistic twig join etc.) is monotonous in the size of each of its inputs, that is: if we fix all but one inputs to the operator and *add* tuples to the last input, the cost of evaluating the operator increases. Summing this up over a fixed physical plan, the more data is added in the views on which the plan is computed, the higher the physical plan evaluation cost.

For simplicity of notation, we will use $cost$, $size$ instead of $cost^\epsilon$ and $size^\epsilon$. Our problem statement can thus be formalized as:

> Find the view set V_{best} such that $\sum_{v \in V_{best}}(size(v)) < S$ and $\sum_{q \in Q}(cost(q|_{V_{best}})) \leq \sum_{q \in Q}(cost(q|_V))$ for all view sets V fitting in S.

Minimal rewriting Each equivalent rewriting r of a query q produced by [16] is *minimal*, that is: one cannot obtain an equivalent rewriting of q using only *a strict subset of the view occurrences*

appearing in r. For instance, the rewriting r in Figure 2, of the form $\sigma(v_1 \bowtie_{paper.ID} v_2)$ is minimal. In contrast, a rewriting r' of the form $\pi(\sigma(v_1 \bowtie_{paper.ID} v_1 \bowtie_{paper.ID} v_2))$, using the v_1 view twice, in a self-join on paper.ID, is not minimal. Considering only minimal rewriting allows keeping view storage space *and* rewriting evaluation costs low. Indeed, with our assumptions on $cost^{\epsilon}$, a non-minimal rewriting is likely to incur a higher evaluation cost than the non-minimal one – if only for scanning the extra view occurrences.

Rewriting complexity The rewriting algorithm we build on [16] is computationally expensive: it extends the simpler XPath 1.0 case when views and query have a single returning node, for which the problem is coNP-hard [14]. As we will show, frequent calls to the rewriting algorithm negatively affect view selection performance.

3. CANDIDATE VIEW SETS

In this Section, we first explain (Section 3.1) which views can be considered candidates. Section 3.2 presents a set of techniques for pruning candidate views. Finally, Section 3.3 quantifies the interest of a materialized view using the notion of benefit.

3.1 Candidate views for a workload

DEFINITION 3.1 (CANDIDATE VIEW). *A view v is a candidate view for a query q* iff *there exists an equivalent rewriting of q using v (and possibly other views).*

We denote by $\mathcal{CS}_0(q)$ the set all candidate views for the query q, and by $\mathcal{CS}_0(Q)$ the candidates for all queries in a workload Q. We start by considering candidates for tree pattern queries:

Candidate views for a tree pattern query It has been shown [16, 17] that a tree pattern view v may participate in an equivalent rewriting of a tree pattern query q only if there exists a tree embedding $\phi : v \to q$. This embedding must preserve node labels, i.e., for any $n \in v$, $label(n) = label(\phi(n))$. The embedding must also respect structural relationships between nodes:

- for any node $n \in v$ and m a /-child of n, $\phi(m)$ must be a /-child of $\phi(n)$;
- for any node $n \in v$ and m a //-child of n, $\phi(m)$ must be a descendant of $\phi(n)$.

Finally, ϕ must not contradict value predicates from the query, i.e.: for any node $n \in v$, such that $m = \phi(n) \in q$, if m is labeled with a predicate of the form $[val = c_1]$ for some constant c_1, then n must not be labeled with a predicate of the form $[val = c_2]$ for some constant $c_2 \neq c_1$.

We now turn to the task of enumerating $\mathcal{CS}_0$ candidates. We denote by *unlabeled* tree patterns those patterns whose nodes carry an attribute or element name, but no annotation of the form val, $cont$, ID, or $[val = c]$. One can enumerate all candidate views for a workload Q by (i) enumerating all *unlabeled* tree patterns that can be embedded in some query $q \in Q$ and (ii) creating from each unlabeled tree pattern thus obtained, all possible tree patterns that differ in their ID, val and $cont$ annotations. For example, consider the workload Q_1 consisting of the single query q_1: $/a/b/c_{val}$. Sample unlabeled tree patterns which can be embedded in q_1 are: $/a$, $//a$, $//b$, $//c$, $/a/b$, $/a//b$, $\ldots$, $/a/b/c$, $//a/b/c$ etc. In turn, from the unlabeled tree pattern $/a$, one can derive e.g., the labeled patterns $/a_{ID}$, $/a_{ID,val}$, $/a_{ID,val,cont}$, $/a_{cont}$ etc.

How to estimate the size of $\mathcal{CS}_0(q)$? We denote the number of nodes of q by $|q|$ and start by counting the unlabeled tree patterns which can be extracted out of q. For a given $k < |q|$, there are $\binom{|q|}{k}$ subsets of q nodes, and in the worst case, each subset determines a sub-pattern of q, having k edges (counting also the edge above the sub-pattern root, e.g., in Figure 3, the // edge above the $confs$ node). Each such edge could be labeled / or //. Thus, the number of unlabeled tree patterns of k nodes that can be constructed from a query q is, in the worst case, $\binom{|q|}{k} \times 2^k$.

We now consider building candidate views out of an unlabeled tree pattern t_{ul}. Let n be a node of t_{ul} such that there is an embedding from $\phi_{ul} : t_{ul} \to q$ for some q in the workload. To obtain a labeled tree pattern t (candidate view) out of t_{ul}, we need to decide on the annotations of each node $n' \in t$ corresponding to $n \in t_{ul}$. We can annotate n' with any of the four subsets of the attribute set $\{ID, cont\}$, to indicate whether t stores an ID and/or the full serialized XML image of the node. With respect to the val attribute, two cases occur: (i) if the query node $\phi_{ul}(n)$ is annotated with a predicate of the form $[val = c]$, one may label n' with either val, $[val = c]$ or no val label; (ii) if $\phi(n)$ has no such predicate, we can annotate n' with val, or omit the val label, but we cannot annotate with $[val = c]$ for any constant c, since this would prevent the existence of an embedding $\phi : t \to q$ and thus prevent t from being a candidate view for q. Thus, in the worst case, there are 3 val annotation possibilities for n', which, multiplied by the 4 possibilities of $ID, cont$ annotation, lead to 12 possible node annotations for n'. Assuming the size of t_{ul} (and t) is k, the node annotation possibilities alone lead to 12^k possible t trees out of a given t_{ul}.

Based on this, out of a query q, the number of candidate views of size k is: $\binom{|q|}{k} \times 2^k \times 12^k$, where the 2^k factor is due to the edge labeling possibilities and the 12^k factor is due to node annotations. It follows that $|\mathcal{CS}_0(q)|$ is:

$$\sum_{k=1}^{|q|} \binom{|q|}{k} \times 2^k \times 12^k = \sum_{k=0}^{|q|} \binom{|q|}{k} \times 24^k - 1 = 25^{|q|} - 1$$

We end the discussion of candidate views for tree pattern queries with an interesting remark. Given a workload Q and view set V, we say a view $v \in V$ is *useful* if v is used in the best rewriting of some query $q \in Q$ using V. The set of useful candidate views for a query is guaranteed to be quite small: it turns out that a minimal rewriting of a tree pattern query q uses no more than $2 \times |q|$ views [16]. However, we do not know which are the useful views before rewriting all the queries; moreover, our aim is to select views that are *globally* best for the whole workload. Thus, one cannot use this known small bound to prune out candidates.

Candidate views for a query with value joins We now turn to the case of a tree pattern query q with value joins. One can show that a view v may participate to an equivalent rewriting of q only if there exists a set of tree embeddings $\phi_1 : t_1^v \to t_1^q$, $\phi_2 : t_2^v \to t_2^q$ etc. embedding each view tree pattern to some query tree pattern, and satisfying the following condition. For each value join in v of the form $n_i^v.val = n_j^v.val$, where n_i^v, n_j^v are nodes in the view tree patterns t_i^v, respectively t_j^v, the query must feature a value join edge between the nodes $\phi_i(n_i^v)$ and $\phi_j(n_j^v)$.

For example, consider the view v_2, with a value join between the year nodes of its tree patterns, and the query q in Figure 3. Let ϕ_l the embedding from v_2's left subtree into the right subtree of q, and ϕ_r the embedding from v_2's right subtree into the left subtree of q. Observe that ϕ_r and ϕ_l map the year nodes of v_2 into the two year nodes of the query, thus the condition for v_2 to participate in some rewriting of q is satisfied. The intuition is that a view with "more join predicates" than the query cannot be used to rewrite it, following the similar property of relational containment mappings.

More generally, let q be a query q consisting of k tree patterns t_i^q, $1 \leq i \leq k$, and assume q has m value joins. We enumerate the candidate views as follows. (i) Build the candidate view sets $\mathcal{CS}_0(t_i^q)$ for $1 \leq i \leq k$; (ii) For each subset $\{i_1, i_2, \ldots, i_n\}$ of $\{1, 2, \ldots, k\}$, and each set of candidate tree patterns $t_1 \in \mathcal{CS}_0(t_{i_1}^q)$,

$t_2 \in \mathcal{CS}_0(t^q_{i_2})$ etc., create the candidate view $t_1 \times t_2 \times \ldots \times t_n$. Then, let JE be the set of query value join edges, such that embeddings from $t_1, t_2, \ldots, t_n$ into the query reach both ends of the value join edge. For each subset of $J \subseteq JE$, we generate a distinct candidate view by pushing on top of $t_1 \times t_2 \times \ldots t_n$ the join conditions of J. Overall, the number of candidate views for q is bound by $2^m \times |\mathcal{CS}_0(t^q_1)| \times |\mathcal{CS}_0(t^q_2)| \times \ldots \times |\mathcal{CS}_0(t^q_k)|$.

For example, in Figure 3, to obtain candidate views for q, one can first chose $\{i_1 = 1\}$ and generate the set $\mathcal{CS}^1_0$ of all candidate tree patterns for the left subtree of q; then, chose $\{i_1 = 2\}$ and generate the set $\mathcal{CS}^2_0$ of all candidate tree patterns for q's right subtree; finally, choosing $\{i_1 = 1, i_2 = 2\}$ leads to enumerating all combinations of the form $\{t_1, t_2 \,|\, t_1 \in \mathcal{CS}^1_0, t_2 \in \mathcal{CS}^2_0\}$ and, for each such t_1 and t_2: (*a*) add the view $t_1 \times t_2$ to the candidate set of q; (*b*) if t_1 and t_2 both have a year node, also add the view $t_1 \bowtie_{year.val} t_2$ to the candidate set.

The number of candidate views for all queries in a workload may be prohibitively high. Of course, the more commonality the queries exhibit, the more common candidates they may have, but this still leaves a large number of candidate views.

3.2 Pruning candidate views

We now describe several methods for pruning candidate views. We start by introducing two important notions. Let Q be a workload and V_1, V_2 be two candidate view sets.

DEFINITION 3.2 (REWRITING POWER PRESERVATION). *If, for every query $q \in Q$ and rewriting r of q using views in V_1, there exists a rewriting r' of q using views from V_2, we say that replacing V_1 with V_2* preserves rewriting power.

Rewriting power preservation ensures that V_2 enables to rewrite at least the queries V_1 did. However, it says nothing about the cost of the rewritings using V_2. The following notion is more restrictive:

DEFINITION 3.3 (REWRITING COST PRESERVATION). *If (i) replacing V_1 with V_2 preserves rewriting power and (ii) for any query $q \in Q$ and rewriting r of Q using V_1, there exists a rewriting r' of q using the views V_2 such that $cost^\epsilon(r') \leq cost^\epsilon(r)$, we say that replacing V_1 with V_2* preserves rewriting costs.

Now several candidate view pruning techniques are described. Based on a set of candidate views V, our first techniques each focus on a tree pattern query. Then we discuss pruning methods targeting queries with value joins.

ALLID For any view $v \in V$, let v_{ID} be the views obtained by copying v and then adding the ID annotation to all nodes (v_{ID} may or may not be identical to v). The transformation ALLID consists of replacing V with the set $V' = \{v_{ID} | v \in V\}$. In other words, we only keep the views where all nodes are annotated with ID. For example, consider the query q and the view cv_7 of Figure 5. The transformation ALLID removes the view cv_7 from the set of candidate views and replaces it with cv_4 that has the ID annotation in all of its nodes.

It is easy to see that ALLID preserves rewriting power: storing more IDs enables *more* joins among the views (and thus, more rewritings). However, it may not preserve rewriting costs, since storing IDs for all nodes may increase the view size. To compensate, our view selection algorithms aggressively prune out IDs which turn out not to be used by the best rewritings, as we will explain in Section 4.

TRIMAXIS We have mentioned earlier in this Section that by labeling candidate view edges either / or //, one gets 2^k possible edge labelings for a k-node view. Our TRIMAXIS transformation eliminates some of the options brought by the edge labeling possibilities. For example, consider a workload consisting of the query q of Figure 5 and a view set including the candidate views cv_5 and cv_6 in the same Figure. In this case, TRIMAXIS will remove cv_5, since it has an ancestor-descendant edge // mapped only to a parent-child query edge, and will preserve cv_6, identical to cv_5 except for the label of the edge between the a and c nodes.

Formally, we define this pruning as follows: let v_1, v_2 be two views in V, identical except for one edge: e_1 in v_1 is of the form n^1_1/n^2_1, and e_2 in v_2 is of the form $n^1_2//n^2_2$, n^1_1 and n^1_2 have the same label, while n^2_1 and n^2_2 have the same label. Assume that for every $q \in Q$ and every embedding $\phi_2 : v_2 \rightarrow q$, there is an embedding $\phi_1 : v_1 \rightarrow q$ such that $\phi_1(n^1_1) = \phi_2(n^1_2)$, $\phi_1(n^1_2) = \phi_2(n^2_2)$ and ϕ_1, ϕ_2 coincide on all the other nodes of v_1 and v_2. Then, TRIMAXIS transforms V into the view set $V' = V \setminus \{v_2\}$, in other words it removes v_2.

It can be shown that TRIMAXIS preserves rewriting power, because for every rewriting r based on v_2 one can build a rewriting r' based on v_1 computing the same results. TRIMAXIS also preserves rewriting costs, since v_1 stores at most as much data as v_2.

One may wonder whether TRIMAXIS should not also work the other way around, that is, prune views with / edges if they only match // edges in workload queries. However, such views are not candidates, because they do not embed into the query. For instance, if the query is $//a//b_{cont}$, the view $//a/b_{cont}$ cannot be used to rewrite it, thus it is not a candidate view.

TRIMVAL Let $v \in V$ be a view and n be a view node. Assume that the set of all embeddings of v into workload queries is $\{\phi_1 : v \rightarrow q_1, \phi_2 : v \rightarrow q_2, \ldots, \phi_k : v \rightarrow q_k\}$. Assume that for any $1 \leq i \leq k$, the query node $\phi_i(n)$ is neither annotated val nor with a predicate of the form $[val = c]$ neither takes part in any value-join. Transformation TRIMVAL replaces v in V with a copy v', in which the copy of n is not annotated val.

For instance, in Figure 5, TRIMVAL replaces the view cv_3 with a copy of cv_3 whose a node is not annotated with val. The replacement takes place since the a query node q is not labeled val, nor $[val = c]$, and does not take part in a value join.

TRIMVAL preserves rewriting power, since it removes only val annotations that are useless in any rewriting. It also preserves rewriting cost, since eliminating val reduces view space occupancy without breaking any useful rewritings.

TRIMCONT It seems natural to remove unused $cont$ annotations just like val ones. One must take into account, however, that $cont$ attributes can be used by rewriting in a way that val does not support: as explained in Section 2.3, one may navigate by applying an XPath expression within a $cont$ attribute, to extract a subset of the data stored in that node (recall the example in Figure 4). Thus, before removing a $cont$, one must ensure this does not prevent some interesting navigation.

Formally, let $n \in v$ be a node in a candidate view v and let $\{\phi_1 : v \rightarrow q_1, \phi_2 : v \rightarrow q_2, \ldots, \phi_k : v \rightarrow q_k\}$ be the set of all embeddings of v into Q queries. If for any $1 \leq i \leq k$, the query node $\phi_i(n)$ is (*i*) not annotated $cont$ and (*ii*) a leaf in q_i, transformation TRIMCONT replaces v with a copy v', in which the copy of n is not annotated $cont$. For instance, in Figure 5, TRIMCONT removes cv_1 and replaces it with a copy thereof, where the b node is not annotated $cont$. It is easy to show that TRIMCONT preserves rewriting power and cost as it removes only useless annotations.

We now present a candidate view pruning technique specific to workloads with value joins.

NOCART Eliminating views with cartesian products, i.e., those having a tree pattern unconnected (by a value join) to any other tree pattern, can significantly reduce the number of candidates. For instance, for a query of 2 tree patterns t_1, t_2 combined by a value

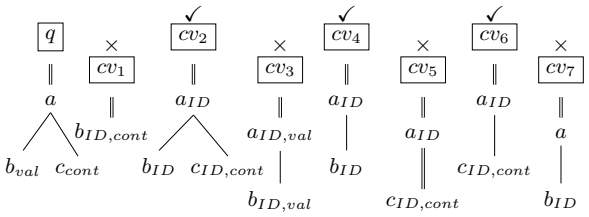

Figure 5: Sample query and some of its candidate views.

join, avoiding cartesian products leads to an upper bound reduction from $2 \times |\mathcal{CS}(t_1)| \times |\mathcal{CS}(t_2)|$ to $|\mathcal{CS}(t_1)| + |\mathcal{CS}(t_2)|$. However, in the (unlikely) case that the query features a cartesian product, and that the space budget allows it, it may be a good idea to materialize it. We define transformation NOCART as pruning away views with a cartesian product such that in the query, the corresponding tree patterns are connected by a value join (we omit the formalization for brevity). In a nutshell, NOCART only allows query-inspired cartesian products; it preserves rewriting power and cost.

Candidate view set $\mathcal{CS}_1$ For a tree pattern query q, we denote by $\mathcal{CS}_1(q)$ the set obtained from $\mathcal{CS}_0(q)$ by applying ALLID, TRIMAXIS, TRIMVAL, TRIMCONT and NOCART exhaustively, until none of them applies any more. ALLID is guaranteed to remove some views from $\mathcal{CS}_0(q)$, leading to a bound for $|\mathcal{CS}_1(q)|$ of $1/2 \times (25^{|q|} - 1)$. The impact of TRIMVAL and TRIMCONT on the number of candidate views depends on the workload. For example, in Figure 5, we mark the views that are included in $\mathcal{CS}_1(q)$ with "✓" and the ones pruned out with "×".

Since the size of $\mathcal{CS}_1(q)$ may still be quite high for large queries, we introduce a last pruning:

LIN This technique removes all non-*linear* views, i.e., having a node with more than one child. (Only when) applied after ALLID, which ensures IDs on all nodes, LIN preserves rewriting power, however, it may not preserve rewriting costs.

Based on LIN, we identify a smaller candidate set:

Candidate view set $\mathcal{CS}_2$ We denote by $\mathcal{CS}_2(q)$ the candidate view set obtained by applying LIN to $\mathcal{CS}_1(q)$. For a workload Q, we set $\mathcal{CS}_2(Q) = \cup_{q \in Q} \mathcal{CS}_2(q)$. For instance, in Figure 5, the candidate view cv_2 is part of $\mathcal{CS}_1(q)$, but not part of $\mathcal{CS}_2(q)$.

3.3 Benefit of a view set

Let V be candidate view set for the workload Q. To quantify the positive impact of materializing the views in V, we define:

DEFINITION 3.4 (BENEFIT OF A VIEW SET). *The benefit of a view set V is defined as the savings created by V in the processing costs of Q:*

$$b(V, Q) = \sum_{q \in Q} (w_i \times (cost(q|_\emptyset) - cost(q|_V)))$$

where w_i is the weight of query q_i in Q.

Notice that adding v_1 *alone* or v_2 *alone* to an existing view set V may not increase its benefit, if $V \cup \{v_1\}$ and $V \cup \{v_2\}$ do not enable any new interesting rewritings, while adding v_1 and v_2 *simultaneously* to V would increase it. This is the case when a more efficient rewriting can be found based on $V \cup \{v_1, v_2\}$.

4. VIEW SELECTION ALGORITHMS

In this Section we describe algorithms for solving the view selection problem. Section 4.1 presents a simple exhaustive algorithm. Section 4.2 presents an algorithm inspired from the classic Knapsack problem. In Section 4.3 we present a state and transition-based algorithms moving from one view set to another, trying to find an optimum under the space budget constraint.

4.1 Exhaustive search

A simple algorithm for finding the best candidate view set is: (i) enumerate all candidate views in $\mathcal{CS}_0(Q)$; (ii) generate the powerset $\mathcal{P}(\mathcal{CS}_0(Q))$; (iii) compute the benefit of each set $V \in \mathcal{P}(\mathcal{CS}_0(Q))$; (iv) choose the set $V_{best} \in \mathcal{P}(\mathcal{CS}_0(Q))$ having the maximum benefit among those who fit the space budget S.

Clearly, this algorithm computes the solution to our view search problem. However, due to the large size of $\mathcal{CS}_0(Q)$ and even more of its powerset, it is unfeasible for meaningful workloads Q.

4.2 Knapsack-style view selection

Our view selection problem is closely related to the knapsack problem. The classic knapsack problem considers a set of k items having the *space occupancy* s_1, s_2, ..., s_k and the *benefits* b_1, b_2, ..., b_k and tries to fill a space budget S with items so as to maximize the sum of the benefits of the selected items. However, there is a fundamental difference: in our case, since we support multiple-view rewritings, the benefit of selecting one view depends on the presence of other views among those already recommended for materialization. Considering that at some point during the algorithm we have selected the view set V, what we need to identify next is the candidate view (among those not already in V) that would lead to the greatest benefit *together with the views in V*.

We adapt knapsack-style view selection to our setting as follows.

DEFINITION 4.1 (VIEW UTILITY). *For a given workload Q and set of materialized views V, the utility of a view v is the ratio between the benefit brought by materializing v next to V, and the space occupancy of v: $u(v) = b(V \cup \{v\}, Q)/size(v)$.*

Evaluating $b(V \cup \{v\}, Q)$ requires rewriting each query $q \in Q$ using $V \cup \{v\}$, which for large Q and V sets may be extremely expensive. One can reduce this effort by rewriting only *some* queries $q \in Q$, namely those having a tree pattern t^q into which some tree pattern t^v of v embeds. (It is easy to show that the best rewritings of the other Q queries are not affected by the addition of v).

Utility-driven greedy (UDG) Our first utility-driven view selection algorithm, based on a given candidate view set V_c, goes as follows: (i) initialize the recommended view set V to $\emptyset$; (ii) compute the utility of each view $v \in V_c$; (iii) sort the candidate views in descending order of their utility; (iv) add the candidate with the highest utility to V, if it fits the space budget, and remove it from V_c; (v) repeat (ii)-(iv) until no view $v \in V_c$ has a strictly positive utility, or the space budget S has been attained.

Observe that at step (ii), the algorithm updates the utilities after each addition to V. This is because the utility of a view v_1 with respect to a view set V can either increase or decrease when a view v_2 is added to V. The utility of v_1 may increase, if v_1 and v_2 together enable a very efficient rewriting; the utility of v_1 may decrease if v_2 is a competitor to v_1, i.e., v_2 enables a lower-cost rewriting than one enabled by v_1. The repeated recomputation of utility values brings quite some overhead, especially since it requires calling an expensive query rewriting algorithm, such as [16, 17]. An important remark allows to reduce this overhead: when considering whether or not to add a view v_1 to V, we only need to find out the new rewritings (if any) that v_1 enables. In turn, v_1 can only lead to new rewritings for those queries $q \in Q$ such that there is an embedding $\phi : v \rightarrow q$ (as explained in Section 3.1). This observation significantly reduces the benefit recomputation costs, especially for large workloads.

Algorithm UDG may miss the optimal solution. For instance, assume that query $q \in Q$ can be rewritten very efficiently based on v_1 and v_2, but neither v_1 nor v_2 suffice to rewrite q. In this case, the benefits of v_1 alone and v_2 alone may be small, leading UDG

to chose neither v_1 nor v_2 for materialization. This prevents UDG from realizing how interesting it would have been to add *both*.

4.3 State search-based view selection

One can model our view selection problem as a state search problem. Every *state* consists of a set of materialized views, and a benefit. The *initial state* corresponds to a seed view set V_0, having some benefit $b(V_0)$. By adding, modifying, or removing a view from the initial state, one can obtain another state, characterized by the view set V, having the benefit $b(V)$, which may be higher or lower than that of V_0. Recall that the benefit is based on the cost of the *best* rewritings that V supports. Thus, we represent view selection as an optimal-state search problem in a directed graph, where nodes are states, and edges are transitions from one state to another.

How should one pick the initial state, i.e., the seed set of views? We have decided to start with $V_0 = Q$, that is, the workload queries themselves. If Q needs more space than S, the initial state is not a solution. To solve this problem, we will introduce space-reducing transitions, allowing us to reach acceptable states.

In the following, Section 4.3.1 describes a set of view set transformations, while in Section 4.3.2 we present a search algorithm based on these transformations.

4.3.1 State transformations

The transformations we use determine the space of states that we can reach, and how fast we find interesting states. Moreover, those we present below, when applied on subset of $\mathcal{CS}_1$, can always yield another $\mathcal{CS}_1$ subset. Thus, there is a close relationship between the transformations and the pruning criteria presented in Section 3.2, which will be formalized at the end of this Section.

We start by describing a set of state transformations which can be shown not to lose rewriting power (recall Definition 3.2). Such transformations are most interesting for us since our goal is to maximize benefit, and thus to rewrite as many queries as possible.

BREAK Splitting a view in several sub-views may enable the identification of common sub-expressions across views. Transformation BREAK has three variants:

- *Structural break*: BREAK picks a view $v \in V$ and a v edge connecting the node n_1 to its child node n_2. It replaces the view set V with $V \setminus \{v\} \cup \{v_1, v_2\}$, where: v_2 is a copy of the v subtree rooted at n_2, and v_1 is copy of v from which n_2's subtree is removed. BREAK also adds ID annotations to the copy of n_1 in v_1 and to the copy of n_2 in v_2. This ensures that any rewriting using v is still possible by replacing v with: $v_1 \bowtie_{n_1 \prec n_2} v_2$ if n_2 is a /-child of n_1, respectively, $v_1 \bowtie_{n_1 \prec\prec n_2} v_2$ if n_2 is a // child of n_1.
- *Value-join break*: BREAK picks a view $v \in V$ and a j value-join edge connecting two nodes n_1, $n_2 \in v$. BREAK removes the j edge and adds val annotations to the nodes it used to connect. This may lead either to two distinct views $\{v_1, v_2\}$ (if the removed edge was the only one connecting them), or to a single view having one less value join.
- *Cartesian product break*: BREAK picks a view $v \in V$ and a cartesian product of two sub-views of v, $\{v_1, v_2\}$, such that $v \equiv v_1 \times v_2$. BREAK replaces v with $\{v_1, v_2\}$.

BREAK preserves rewriting power, since the broken join can always be reenforced, using either the ID, val attributes it introduced, or a cartesian product.

JOIN Opposite to BREAK, this transformation adds to the view set V the join of two views $v_1, v_2 \in V$. This transformation may reduce costs by pushing a join from the query into some view. For example, consider the views v_1: $//a_{ID}$ and v_2: $//b_{ID,Val}$ and the query q_1 $//a//b_{Val}$. To evaluate q_1, one has to scan both views, and join them as follows: $v_1 \bowtie_{a_{id} \prec\prec b_{id}} v_2$. Transformation JOIN adds to V the new view $v_{1,2} = //a_{ID}//b_{ID,Val}$, which is exactly the result of the join. Similarly, JOIN joins two views with a value-join or a cartesian product. Note that JOIN joins two views only if the resulting view respects the TRIMAXIS and NOCART pruning techniques of Section 3.2.

GENERALIZE GENERALIZE tries to identify commonality between workload queries by generalizing/relaxing a candidate view. Relaxing a candidate view may increase its space occupancy but it makes it more reusable. GENERALIZE has two variants:

- *Cont generalization*: GENERALIZE picks a view $v \in V$ and a non-leaf node $n \in v$, and replaces v by a view v' in which the child subtrees of n have been erased and n has been annotated $cont$. For instance, if v is $//a[//b]//c[//d_{cont}]/e_{val}$, GENERALIZE may replace it with $//a[//b]//c_{cont}$ if the c node is chosen, or $//a_{cont}$ if the a node is chosen.
- *Val generalization*: GENERALIZE picks a view $v \in V$ and a node $n \in v$ which is annotated with a equality predicate of the form $n_{[val=c]}$ and replaces v with a view v' in which: (i) node n is no longer annotated with an equality predicate and; (ii) node n is annotated val. For instance, if v is $//a//b_{[val=3]}$, GENERALIZE replaces it with $//a//b_{val}$.

GENERALIZE applies to a view only if the resulting view respects the TRIMCONT and TRIMVAL pruning techniques. GENERALIZE preserves rewriting power. However, it can either increase or decrease the storage space, and thus rewriting cost.

ADAPT A candidate view may turn out to be more general than any of the queries into which the view embeds. In this case, transformation ADAPT adds a query-adapted view, typically smaller, which may also reduce query cost by removing the need for processing the view to adapt it to the query. Two variants of adaptation exist:

- Pick a view $v \in V$, a // edge e of v and an embedding $\phi : v \rightarrow q$ for some $q \in Q$. Denote by n_1 the node above e and by n_2 the node below e. If ϕ maps e either to (a) a / path or (b) a path of length greater than one, add to V the view v' obtained by copying v and in the copy, replacing e with the path to which e maps. For example, if the view v $/a//d$ embeds in the query $/a/b/c/d$, add the view v': $/a/b/c/d$.
- Pick a view $v \in V$, a *cont*-labeled node $n \in v$ and an embedding $\phi : v \rightarrow q$ such that $\phi(n)$ has some children. Add to V a view v' obtained by copying v and adding as children to (the copy of) n in v', all the child subtrees of $\phi(n)$ in q. For example, if the view $//a_{cont}$ embeds into the query $//a[//c/d]/b_{val}$, add the view $//a_{cont}[//c/d][b]$.

Observe that the symmetric situation to our first adaptation case, i.e., a view $//a/b_{val}$ and a query $//a//b_{val}$, does not occur, since such a view is not a candidate for the query (Section 3.1).

PROJECT When a view stores attributes not needed by any of the rewritings, we can remove these stored attributes to diminish view space occupancy. Transformation PROJECT picks a view $v \in V$ and replaces it with a view v' restricted to a subset of the stored attributes (ID, val, $cont$ or $[val = c]$) of v.

RANDOM Adding a candidate view to the materialized view set increases view storage size and may also increase benefit if the new view enables some rewritings efficient enough to offset the storage costs. Transformation RANDOM picks a candidate view not already in V, and adds it to the view set.

Our last transformations may trade rewriting power for space:

REMOVE and REMOVE0 Removing a view decreases view storage size and may reduce the benefit of a state. Transformation REMOVE picks a view and removes it from the view set, while

REMOVE0 only removes zero-utility views. REMOVE0 preserves rewriting power and cost, while REMOVE is not guaranteed to do so. Both allow reducing space occupancy.

Importantly, REMOVE0 can be applied after a transition which has added a view v_1, to identify some view v_2 rendered useless by the addition of v_1. In this case, REMOVE0 eliminates v_2.

We also define some variations of our transformations. We consider the repeated exhaustive application of a transition τ, and use the shorthand $V_1 \xrightarrow{\tau*} V_k$ to state that repeated application of τ led from V_1 to V_2, from V_2 to V_3 etc. until V_k. In particular, REMOVE* repeatedly removes the least benefit view from a state V until it fits in the space budget S, REMOVE0* removes all unused views from a state V, while PROJECT* removes *all* unused attributes from *all* views $v \in V$. It can be shown that the state attained by REMOVE0*, or by PROJECT*, does not depend on the order in which the transformations are applied.

Which transition sets suffice to reach all candidate view sets? Since the answer depends on the candidate views and on the initial state, we include them in the definition:

DEFINITION 4.2 (TRANSFORMATION SET COMPLETENESS). *Let Q be a workload, $\mathcal{CS} \subseteq \mathcal{CS}_0(Q)$ be a set of candidate views and V_0 an initial state in $\mathcal{CS}$. A set of transformations $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is $\mathcal{CS}$ and V_0-complete iff for any set of views $V \subseteq \mathcal{CS}$, there exists a sequence of states $V_0 \xrightarrow{\tau_{i_1}} V_1 \xrightarrow{\tau_{i_2}} \ldots \xrightarrow{\tau_{i_k}} V$, where for $1 \leq j \leq k$, $\tau_{i_j} \in \mathcal{T}$.*

Clearly, the set {RANDOM, REMOVE} is complete for any candidate view set $\mathcal{CS} \subseteq \mathcal{CS}_0(Q)$, and initial view set $V_0 \in \mathcal{CS}$, since RANDOM allows generating all possible candidate view sets. However, the transformation path from V_0 to V may be arbitrarily long. Below we present a more practical transformation set.

PROPOSITION 4.1. *The transformation set* {BREAK, JOIN, GENERALIZE, PROJECT, REMOVE} *is $\mathcal{CS}_1(Q)$ and V_0 complete for $V_0 = Q$.*

The completeness follows from the ability of BREAK to break the initial view set all the way to one-node ID-annotated views, JOIN to glue back the pieces to build V, PROJECT out possible extra annotations, GENERALIZE to annotate view nodes with $cont$ or val. Extraneous views can be removed with REMOVE.

We end by noting that an exhaustive application of BREAK, JOIN, PROJECT, GENERALIZE and REMOVE on $V_0 = Q$ is still likely to be very costly, first, because of the large number of states reached, and second, because the calls to the rewriting algorithm (needed by *every* JOIN, PROJECT, or REMOVE0) are very expensive, motivating further interest in searching for heuristics.

4.3.2 Reduce-optimize algorithm (ROA)

Based on the above transformations, we devised a search algorithm with a randomized component, which we term Reduce-Optimize Algorithm (or ROA, in short). ROA repeatedly executes two successive phases: first, the *reduce* phase which seeks to reduce the space occupancy of a state, by applying a chain of transformations on it; then, the *optimize* phase which attempts to increase the benefit of the target state. Both phases follow a trial-and-error approach, that is, an attempted transformation may fail to reduce space during *reduce*, or fail to increase benefit during *optimize*. If this happens, *reduce* (respectively, *optimize*) simply ignores the unsatisfactory target state and continues applying other transformations starting from the previously attained state.

One phase *may* also reach the desirable effect of the other, that is, *optimize* can reduce the space occupation of a state and *reduce* can increase the benefit of a state. However, most often, these objectives conflict, thus each phase strictly attempts to obtain one of the two improvements.

As described above, *reduce* and *optimize* explore the space in quite a linear fashion, that is, the fan-out of the search is low: if, say, the first transition attempted on V_0 during *reduce* does diminish space occupancy, we move to the resulting state V_1 and do not come back to V_0 to apply other transformations to it. A small search fan-out is desirable since exploring all possibilities would lead to too many states and unacceptably slow down the search. However, a disadvantage is that this leads to never visiting large parts of the search space and potentially missing interesting states. To cope with this, whenever *reduce* or *optimize* find a state on which the *reduce-optimize* sequence has already been applied, ROA jumps to a randomly chosen state, and continues the search from there.

As Algorithm 1 shows, the best state returned by ROA is stored

Algorithm 1: Reduce-Optimize Algorithm (**ROA**)

Input : Query workload $\mathcal{Q}$, candidate view set $\mathcal{CV}$, space budget S
Output: Best view set V_{best}

$V_{best} \leftarrow \emptyset$; $V \leftarrow Q$ // V is the current state
$V \leftarrow$ rewriteAndTrim(V, Q)
$\mathcal{S} \leftarrow \emptyset$ //the set of states on which a *reduce-then-optimize* sequence has been applied
while *!timeout* **do**
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{V\}$
  // *reduce* phase:
  foreach $\tau \in$ {BREAK, JOIN,GENERALIZE, ADAPT, REMOVE*} **do**
    $V' \leftarrow \tau(V)$ //apply τ to V
    $V' \leftarrow$ rewriteAndTrim(V', Q)
    if $size(V') < size(V)$ **then**
      $V \leftarrow V'$ //we found a smaller state, *reduce* will continue on this one
    else
      //ignore V', *reduce* will continue on V
    if $size(V) \leq S$ **then**
      break //end of *reduce* phase
  //*optimize* phase:
  foreach $\tau \in$ {ADAPT, JOIN} **do**
    $V' \leftarrow \tau(V)$ //apply τ to V
    $V' \leftarrow$ rewriteAndTrim(V', Q)
    if $b(V', Q) > b(V, Q)$ **then**
      $V \leftarrow V'$
  //seek a new state on which to apply *reduce-then-optimize*:
  while $V \in \mathcal{S}$ **do**
    //at most k attempts of adding a random view
    foreach $i \in 1, 2, \ldots k$ **do**
      $V \leftarrow V \cup \{v$ chosen at random from $\mathcal{CV}, v \notin V\}$
      $V \leftarrow$ rewriteAndTrim(V)
      if $V \notin \mathcal{S}$ **then**
        break
    if $V \in \mathcal{S}$ **then**
      $V \leftarrow$ REMOVE(V); $V \leftarrow$rewriteAndTrim(V)
return V_{best} //updated by calls to procedure rewriteAndTrim

Algorithm 2: Procedure **rewriteAndTrim**

Input : View set V, workload Q
Output: Restriction of V to the views and attributes needed by the best rewritings of Q. Side effect on V_{best}

$E \leftarrow \{$rewrite$(q, V) | q \in Q\}$
$W \leftarrow$ PROJECT*(REMOVE0*(V)) //remove views and IDs not used in the best minimal rewritings
if $cost(Q|_W) < cost(Q|_{V_{best}})$ *and* $size(W) \leq S$ **then**
  $V_{best} \leftarrow W$
return W

in its local variable V_{best}. To simplify the description, we assume V_{best} is modified by the helper procedure **rewriteAndTrim** (Algorithm 2). This procedure is the only place where the (costly) **rewrite** algorithm of [16] is called. After each call, the incoming view set V is trimmed down by exhaustively applying PROJECT and REMOVE0, and the trimmed-down version W is returned.

Algorithm's ROA exploration history is stored in the set $\mathcal{S}$ of all states on which *reduce* has been applied followed by *optimize*. During the *reduce* phase, it successively tries several transformations.

If a transformation reduces storage space, the next will follow from the reduced state, otherwise, ROA will apply it again on the start state V. The *reduce* phase ends either when 5 successive transformations have been applied, or when the size of the state found so far has decreased under the space budget S. Similarly, *optimize* attempts to apply two transformations to increase the current state benefit. Finally, after the two phases, we need to find a new state to work on. If the current state (reached at the end of *optimize*) has not yet gone through the two phases, ROA restarts *reduce* from there. Otherwise, we successively draw k random candidate views, and check if they enable new rewritings. Observe that **rewriteAndTrim** may remove these views, or views from V, when they are rendered useless by a randomly-added view. We have empirically set k to be 40 which gave good results; a large k increases the chances of finding a good view but lengthens the search. Finally, if k successive view additions did not lead to a new state, we REMOVE some views until a non-visited state is reached. ROA needs to stop on a timeline, since completing its randomized search would take unacceptably long.

Remarks on the implementation To increase ROA efficiency, we let it take hints from the query optimizer, in order to restrict the space of alternatives for its transformations. Specifically, the REMOVE0 and PROJECT transformations are only applied to remove attributes and views unused *by the best rewritings* of the workload queries (instead of "unused by any rewriting"). Similarly, JOIN only attempts to build view joins *that are part of some query's best rewriting*. This is possible thanks to the fact that rewritings are passed as algebraic expressions from the rewriter to the optimizer, and from there to the view selection.

Another concern is to be able to quickly identify (line 23) whether a state is already in $\mathcal{S}$. To efficiently support this, we index states by (string) signatures of their views, as follows. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a view set and assume first that each v_i is a (minimal) tree pattern [4]. We compute serialized signatures of the V views $\{s(v_1), s(v_2), \ldots, s(v_n)\}$, and sort them into a list $s(v_{i_1}), s(v_{i_2}), \ldots, s(v_{i_n})$ where $(i_1, i_2, \ldots, i_n)$ is some permutation of $(1, 2, \ldots, n)$. Then, $\mathcal{S}$ can be organized as a multi-level hash structure where V is first indexed by $s(v_{i_1})$, then by $s(v_{i_2})$ and so on up to $s(v_{i_n})$. This structure allows determining with certainty by n look-ups whether a given state V of n *tree pattern* views has already been visited. In the general case (tree patterns with value joins), we encode a view of the form $t_1 \bowtie t_2 \bowtie \ldots \bowtie t_k$ as if it was the set of views $\{t_1, t_2, \ldots, t_k\}$. This introduces some imprecision in the state look-up, e.g., when looking up the view set $V_1 = \{t_1 \bowtie t_2\}$, one may (also) find the different view set $V_2 = \{t_1, t_2\}$. In this case, one still needs to check whether the state found by the multi-level look-up really is the same as the one we searched for, but overall, the search remains quite efficient.

5. CLOSEST COMPETITOR ALGORITHMS

An early materialized view selection work for downward XPath (including wildcard $*$ nodes) is [11]. To select materialized views, they use an algorithm inspired from the greedy polynomial approximation of a set-cover problem [26], by defining view utility as the number of queries that it answers. Their set-cover greedy algorithm (which we denote **SCG** from now on) has an upper bound M on the number of views that can be recommended. In our implementation of SCG, to make a meaningful comparison, we dynamically set M to be the number of views selected by SCG that happen to reach our space bound $\mathcal{S}$.

More recently, [19] studied view selection for the same XPath dialect. From an XPath workload, they identify a subset of candidate views, consisting of the minimal XPath queries based on which at least one query may be answered. This set can be organized as a lattice of size $2^{|Q|}$ which two algorithms rely on in order to recommend views. First, the dynamic-programming based *s*pace-*o*ptimized *a*lgorithm (denoted **SOA** in the sequel) searches for the smallest view set that can rewrite all the workload. Their second algorithm which we denote **STT** seeks to optimize a *s*pace/*t*ime *t*rade-off. It assigns to each view a benefit computed by summing the *weights* of the queries it can answer (regardless of the costs), divided by the view size. STT then greedily selects views in the decreasing order of their utility, until the space budget is filled up or all workload queries can be rewritten using the views.

Conceptually, the biggest difference between these and our algorithms is that they only apply for *XPath queries returning single nodes*. To compensate for this, we plugged in our implementation of SCG, SOA and STT the query rewriting, embedding etc. modules relevant for our language (Section 2).

A second important difference is that [11] and [19] assume that *each query can only be re-written based on at most one view*, while we (as well as e.g., in [14, 17, 27]) consider query rewritings *based on multiple views*. This significantly complicates our setting, since for each query q and n candidate views, up to 2^n view sets may be used to rewrite q, instead of just n. Also, as our experiments will show, the algorithms [11, 19] by design do not capture the opportunities of multiple view-based rewritings, and in our setting, different algorithms exploiting these opportunities can achieve much better savings.

6. EXPERIMENTAL EVALUATION

We now describe experiments we have performed with our and previous view selection algorithms. Section 6.1 outlines our software experimental framework, we describe the data and workloads in Section 6.2 and the algorithms with their settings in Section 6.3. Section 6.4 study candidate view set sizes, Section 6.5 algorithm effectiveness, and Section 6.6 their efficiency. We end with a conclusion of the experiments.

6.1 Framework

We have implemented our view selection algorithms within a Java-based XML data management platform that we developed. The platform supports the materialization of the complex XML views, featuring tree patterns with multiple returning nodes and value joins, described in Section 2.1. View tuples are stored into a native store that we built using the Berkeley DB library v3.3.75. The platform also provides a view-based rewriter module which, given a query q and a set of materialized views V, returns the best rewriting of q using views in V, as discussed in Section 2. Its optimizer takes as input the rewritings (logical plans over the views), pushes selections and projections, re-orders joins, identifies groups of binary structural joins to be transformed in an n-ary holistic twig join etc. Logical plans are then translated into physical plans including operators to: scan materialized views, apply selections, projections, value-based or structural joins (e.g., holistic joins [3]), add Sort operators when needed etc. To evaluate tree patterns directly on the data, as well as nav operators, we implemented an efficient XML stream-based tree pattern matching algorithm [28].

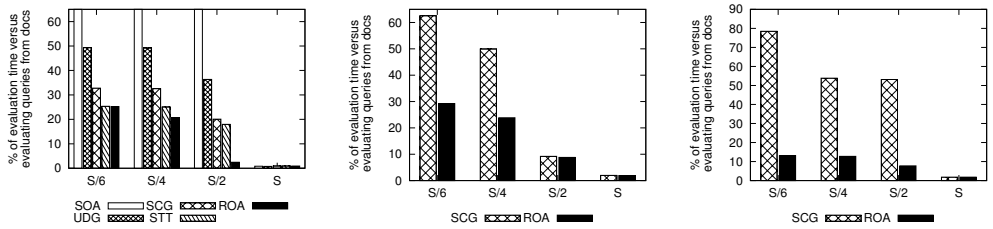

Figure 6: Workload execution time for the workloads Q_1, Q_2 and Q_3 based on views selected by various algorithms.

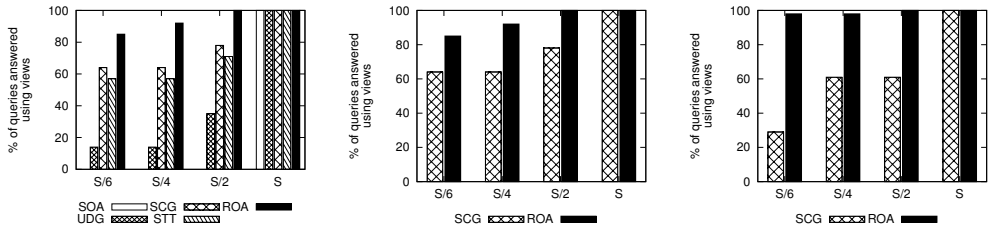

Figure 7: Fraction of queries from the workloads Q_1, Q_2 and Q_3, rewritten using the recommended views.

Workload	$\lvert\mathcal{CS}_0\rvert^{max}$	$\lvert\mathcal{CS}_0\rvert^{max}$	$\lvert\mathcal{CS}_1\rvert$	$\lvert\mathcal{CS}_2\rvert$
Q_1	10^{12}	10^{13}	5944	992
Q_2	10^{12}	10^{13}	7582	1054
Q_3	10^{12}	10^{13}	10014	1454
Q_4	10^{16}	10^{16}	8570	1250

Table 1: Size of candidate view sets $\mathcal{CS}_0$, $\mathcal{CS}_1$ and $\mathcal{CS}_2$.

The *physical plan cost estimation function cost*$^\epsilon$ takes into account the I/O cost of scanning views, as well as the CPU costs of selections, projections, joins, navigation, sort etc.

We have implemented our *view size estimation* function $size^\epsilon(v)$ based on a DataGuide [5] augmented with detailed statistics for each parent-child path p starting from the root of a document d: number of nodes on path p, minimum, average and maximum number of children of a node on path p on each child path, minimum and maximum string value of the nodes, number of distinct string values, average size of *cont* etc. Our $size^\epsilon(v)$ function also makes simple independent-distribution and uniform-distribution assumptions. More elaborate estimations such as e.g., TreeSketch [29] could easily plugged instead in our view selection approach.

6.2 Inputs: data, queries and space budget

We used 10 synthetic XMark benchmark (xml-benchmark.org) documents of 100 MB each resulting in a total of 1 GB.

Four our tests, we generated four random query workloads based on the XMark document structure and content. First, we use three tree pattern query workloads Q_1 of 14 queries, Q_2 having 50 queries and Q_3 of 100 queries. We picked the size of Q_1 as the largest that all algorithms could handle, and Q_2, Q_3 to study further scale-up. Q_1, Q_2 and Q_3 only have tree pattern queries; each query has between 3 and 8 nodes. We added a fourth workload Q_4 of 50 queries, 10 of which have value joins; Q_4 queries have between 6 and 15 nodes. All queries have 2 to 4 returning nodes.

Within each workload, we varied query selectivity as follows. From each document (each 1/10 of the data), 30% of the queries return just 1 result, 30% return a few hundred results, while 30% return a few thousand results. The remaining 10% of the queries return hundreds of thousands of results.

Clearly, materializing the workload is the best solution if the space budget allows it, but the interesting area is when this is not possible due to space constraints. For that purpose, we have taken $S = \sum_{q \in Q} size(q)$, and tested with the space budgets: $S/6$, $S/4$, $S/2$ and S. The main interest of the S value is to show the minimum possible query processing cost.

6.3 Algorithms and settings

We have implemented our algorithms UDG (Section 4.2) and ROA (Section 4.3.2), as well as SCG [11], SOA and STT [19] discussed in Section 5. Our UDG algorithm is quite similar to STT [19]: beyond their rewriting differences, their benefits are different (our includes processing costs and query weights, theirs only query weights), but the greedy approach is the same.

The SCG, SOA and STT algorithms start with the workload itself. Concerning our own algorithms, we used $V_0 = \emptyset$ for UDG since it proceeds by adding views (thus we start it with all the allowed space free), and $V_0 = Q$ for ROA which is more powerful and can change (break, join, adapt etc.) views in many ways. We experimented with UDG and ROA both on the $\mathcal{CS}_1$ and $\mathcal{CS}_2$ candidate view sets. For the workloads we tested, they lead to similar results, thus we report on our UDG and ROA experiments using $\mathcal{CS}_1$ as a candidate view set.

We used a desktop having an Intel Xeon CPU 5140 @2.33 Ghz, 4 GB of RAM and a 60 GB SCSI hard disk at 10.000 RPM.

6.4 Candidate view set size

Our first experiment studies the size of the candidate view sets $\mathcal{CS}_0$, $\mathcal{CS}_1$ and $\mathcal{CS}_2$ (discussed in Section 3.2) for the four workloads. Exhaustive enumeration of $\mathcal{CS}_1$ views is not possible even for medium-size queries, e.g., for a tree pattern of 8 nodes, the $\lvert\mathcal{CS}_0\rvert$ bound is $25^8 \approx 152$ billions. Instead, we use a lower bound $\mathcal{CS}_0^{min}$ assuming that all workload queries are the same (thus, their candidate views overlap) and an upper bound $\mathcal{CS}_0^{max}$ assuming the queries have nothing in common (thus all candidate views are different). We built and counted the actual sets $\mathcal{CS}_1$ and $\mathcal{CS}_2$. Table 1 shows the candidate view counts. Candidate views are reduced by many orders of magnitude using the pruning techniques presented in Section 3.2. This makes candidate view set search, feasible.

6.5 View selection algorithm effectiveness

We ran the existing algorithms SCG, SOA, STT and our algorithms UDG and ROA on tree pattern queries, which they were all built for (modulo the many returning nodes, for which we adapted the competitors as explained in Section 5); we the workloads Q_1, Q_2 and Q_3. We then materialized their recommended views, evaluated the rewritings within our execution framework three times, measured the average time, and show it in Figure 6 as a percentage of the time to evaluate the queries directly on the database.

The first observation is that SOA, STT and UDG do not scale be-

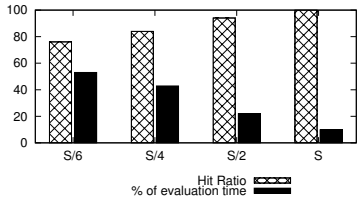

Figure 8: ROA performance on the workload Q_4.

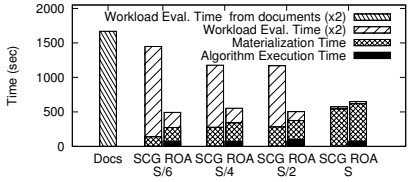

Figure 9: Overall performance of SCG and ROA.

yond Q_1. For SOA and STT, this is because they develop a candidate view set whose size is exponential in the size of the query, and this does not fit in the memory for larger workloads (similarly, [19] tests them on a 16-queries workload). For the 50-queries workload Q_2, our UDG algorithm had not finished running after two hours, when we stopped it. This is because UDG needs to update the benefit of every candidate view after each view addition, which in turn requires rewriting all the workload queries for every candidate view. Thus, for Q_2 and Q_3 we only measured SCG and ROA.

We also noticed that for Q_1, SOA failed to recommend any view (thus the evaluation time is plotted as 100% of the time to evaluate on the database directly) when given a space budget of at most $S/2$. This is because SOA only seeks view sets that can rewrite the whole workload, which for Q_1 could not be found at an $S/2$ space budget. In contrast, as expected, the greedy STT finds interesting view sets saving 50% of the execution costs or more.

Overall, for any workload and space budget, the ROA algorithm achieved the largest query processing cost reductions. This is because it exploits all rewriting possibilities: it tries to use both navigation (GENERALIZE transformation) and joins of several views (BREAK and JOIN transformations), aggressively prunes needless views and attributes (**rewriteAndTrim**), opportunistically modifies candidates to suit the queries (ADAPT transformation), and finally manages to visit sufficient parts of the search space through random jumps (RANDOM transformation). The greedy SCG does not perform as well because it does not consider multiple-view rewritings. We confirmed this intuition by inspecting the ROA-selected views: queries were typically rewritten using 2-3 views.

Figure 7 gives a different perspective on the same experiment: which fraction of the workload queries are rewritten based on the recommended views. ROA also is best in this respect.

For the workload Q_4 with value joins, we only ran ROA since it is the only one handling them. Figure 8 shows (in %) the workload evaluation cost reduction *wrt* evaluation on the database, and the ratio of queries rewritten by the ROA-selected views. This confirms the good properties of ROA also for queries with value joins.

6.6 View selection algorithm efficiency

We now study the performance of the view selection algorithms themselves. Based on the findings of the previous Section, we study only SCG and ROA, on the 100-query workload Q_3. Figure 9 depicts the time needed by SCG and ROA to (i) select the views to materialize (algorithm execution time). We let ROA run for 2 hours, and plot the time it needed to achieve 90% of its maximum benefit; (ii) materialize the views recommended by SCG and ROA (iii) evaluate all queries, *twice* based on the views. As a reference, we also show the time to evaluate all queries *twice* directly on the data. We timed two executions since views are typically materialized to support repeated query execution; in this case, two executions already enable the views to pay off, that is, the time to select, materialize, and exploit the views to evaluate the queries is smaller than the time to evaluate the queries directly on the database.

In this experiment, selecting, materializing and exploiting views paid off even for a single execution (except, of course, when materializing the workload itself). Moreover, SCG is much faster than ROA: SCG execution time is invisible in the Figure. This is because the greedy SCG never comes back on its decisions, whereas ROA investigates more complex view configuration settings, and may search for a long time due to its randomized component.

Showing the ROA time only up to attaining 90% of its biggest benefit may seem to give it an unfair advantage, since in practice we only stopped it after 2 hours. However: (i) increasing the view-based evaluation time in Figure 9 by a factor of $100/90$ does not change the overall picture and (ii) the robustness of the relatively quick cost reductions of ROA is confirmed by our next experiment.

	$S/6$			$S/4$			$S/2$		
	60%	80%	100%	60%	80%	100%	60%	80%	100%
Q_1	1	1	5	1	1	15	1	1	1
Q_2	1	1	25	1	1	25	1	2	2
Q_3	1	2	15	2	2	3	2	2	20
Q_4	1	1	13	1	1	1	1	1	1

Table 2: ROA time to attain increasing benefits (minutes).

Table 2 depicts the evolution of benefit through ROA execution. For each workload and space budget, we show the time (in minutes) it has taken ROA to attain 60%, 80% and 100% of the biggest benefit found in two hours. In all cases, 80% of the benefits were attained in just 2 minutes, while the maximum benefit was always attained within 25 minutes. While such times are still much longer than e.g. the greedy SCG, ROA recommends much better views. Moreover, view selection is typically an off-line process, thus we view the running times as acceptable.

6.7 Experiment conclusion

Our experiments have shown that the candidate sets $\mathcal{CS}_1$ and $\mathcal{CS}_2$ are of manageable size for workloads of up to 100 queries. Working on $\mathcal{CS}_1$, we have demonstrated that ROA and SCG scale up to 100 queries, whereas SOA and STT outgrow the available memory for 50 queries. Moreover, ROA achieved better savings (up to a factor of 8) and finds rewritings for more workload queries than its competitors. ROA (and SCG-) recommended materialized views lead to efficient execution; in our experiments materializing their views paid off starting from 2 workload runs. ROA's disadvantage is that being randomized, it needs to be stopped by a time-out, and is significantly slower than SCG. However, in practice, ROA achieves significant cost reductions (bigger than SCG) after relatively short times, of the order of minutes in our experiments. This confirms its interest for recommending views on complex XQuery workloads featuring many return nodes and value joins.

7. RELATED WORK

Our view selection approach bears similarities to those used in the relational databases [7, 8, 9]: breaking and joining views to find common sub-expressions, and especially heavily relying on the (rewriter and) optimizer's recommended best plans, since a materialized view is only useful if the rewriting-optimization pipeline identifies recognizes it as such.

The complexity of XML data has lead to several index proposals, such as the DataGuide [5], indexes for navigation in a tree [30], adaptive path indexes of fixed length [6] etc. Indexes can be seen as a specialized class of materialized views, based on which one

only retrieves the identifiers of nodes that need to be retrieved from the store in order to return the query results. In contrast, we focus on materialized views that can help to completely answer complex queries, featuring multiple returning nodes and value joins. In the space of XML view-based query rewriting, closest to our work are the equivalent rewriting algorithms: for an XPath query using one view [11, 12, 13, 31], and for XPath/XQuery using several views [14, 15, 16, 17, 20, 21, 27, 32]. In this work, we built a view selection framework that exploits the recent multiple-view equivalent rewriting algorithm of [16], capable of handling tree patterns with multiple return nodes and value joins. We are the first to study the automated selection of materialized views in this context.

Among the XML view recommender systems, the closest works consider one-view XML query rewriting [11, 19]. We discussed them in Section 5, implemented adapted versions of their algorithms for our problem and show that our ROA algorithm scales better than [19] which requires materializing an exponential-size lattice, and is more effective than [11] since it exploits multi-view, more sophisticated rewritings.

In [33] the authors study view selection to support the reconstruction an XML subtree out of shredded data in relational tables. They show this is NP-hard, and present a PTime approximative solution. The focus in [18] is on recommending relational DB2 XMLTable[1] materialized views for XQuery workloads. Their candidate views are inspired from the XPath snippets appearing in the for, where, let and return XQuery clauses; a transformation close to our GENERALIZE is applied to obtain more generic views. XQuery rewriting consists of translating XQuery into SQL queries over the XMLTable views and taking advantage of the XPath rewriting (based on one XPath views) supported by DB2. Their selection algorithm is a knapsack-style greedy, and experimented on a small workload of 10 queries. They explore a more limited space of alternatives, since they do not split and re-compose tree patterns through ID and structural joins, which ROA does extensively. Moreover, as we have shown, greedy algorithms (e.g., UDG) become impractical for complex view and query languages and multiple-view rewritings, since the repeated re-computation of benefit (through rewriting) takes prohibitive time.

8. CONCLUSIONS AND PERSPECTIVES

In this work, we considered the selection of materialized views for an XQuery dialect consisting of joined tree patterns, and assuming a rich algebraic rewriting framework capable of value, ID-based and structural joins, XPath navigation etc. We formalized the space of candidates which is extremely large, showed how to prune it, and provided a transformation-based algorithm which is efficient and effective for this problem. In the future, we plan to experiment with *query template* as in [11, 13], grouping together similar queries, to support larger workloads. We also plan to investigate the extension of the query template mining approach of [13], focused on single-view XPath rewritings, to our more complex language and query rewriting framework.

Acknowledgements The authors thank Konstantinos Karanasos for his valuable help, comments and suggestions, Federico Ulliana for his input and advice and Julien Leblay for his proofreading and comments. This work was partially supported by the CODEX (ANR 08-DEFIS-004) and DataBridges (ANR 11-EITS-003) projects.

9. REFERENCES

[1] M. Gotz, C. Koch, and W. Martens. Efficient algorithms for descendant-only tree pattern queries. *Information Systems*, 2009.

[2] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: Scalable XML document filtering by sequencing twig patterns. In *VLDB*, 2005.

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

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

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

[6] C. Qun, A. Lim, and K. W. Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *SIGMOD*, 2003.

[7] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, 2000.

[8] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, 1999.

[9] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

[10] S. Abiteboul, J. Mchugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[11] B. Mandhani and D. Suciu. Query caching and view selection for XML databases. In *VLDB*, 2005.

[12] W. Xu and M. Ozsoyoglu. Rewriting XPath Queries Using Materialized Views. In *VLDB*, 2005.

[13] L. H. Yang, M. L. Lee, and W. Hsu. Efficient mining of XML query patterns for caching. In *VLDB*, 2003.

[14] B. Cautis, A. Deutsch, and N. Onose. XPath Rewriting Using Multiple Views: Achieving Completeness and Efficiency. In *WebDB*, 2008.

[15] D. Chen and C.-Y. Chan. ViewJoin: Efficient view-based evaluation of tree pattern queries. In *ICDE*, 2010.

[16] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.

[17] 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.

[18] I. Elghandour, A. Aboulnaga, D. Zilio, and C. Zuzarte. Recommending XMLTable Views for XQuery Workloads. In *XSym workshop*, 2009.

[19] N. Tang, J. X. Yu, H. Tang, M. T. Özsu, and P. A. Boncz. Materialized View Selection in XML Databases. In *DASFAA*, 2009.

[20] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

[21] 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.

[22] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1), 2004.

[23] XPath Functions and Operators. w3.org/TR/xpath-functions, 2007.

[24] I. Manolescu, Y. Papakonstantinou, and V. Vassalos. XML tuple algebra. In *Encyclopedia of Database Systems*. Springer, 2009.

[25] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *VLDB*, 2000.

[26] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[27] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting nested XML queries using nested views. In *SIGMOD*, 2006.

[28] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. In *ICDE*, 2006.

[29] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate XML query answers. In *SIGMOD*, 2004.

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

[31] K. Lillis and E. Pitoura. Cooperative XPath caching. In *SIGMOD*, 2008.

[32] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[33] A. Chebotko and B. Fu. XML reconstruction view selection in XML databases: Complexity analysis and approximation scheme. *LNCS*, 6509, 2010.

[1] http://publib.boulder.ibm.com/infocenter/db2luw/v9/