DOI: 10.2298/CSIS100804037L

An XML-algebra for Efficient Set-at-a-time Execution

Maxim Lukichev[1], Boris Novikov[1], and Pankaj Mehra[2]

[1] Saint-Petersburg University
Saint-Petersburg, Russia
maxim.lukichev@gmail.com, borisnov@acm.org
[2] Whodini,Inc.
CA, USA
pankaj@whodini.com

Abstract. The importance of XML query optimization is growing due to the rising number of XML-intensive data mining tasks. Earlier work on algebras for XML query focused mostly on rule-based optimization and used node-at-a-time execution model. Heavy query workloads in modern applications require cost-based optimization which is naturally supported by the set-at-a-time execution model. This paper introduces an algebra with only set-at-a-time operations, and discusses expression reduction methods and lazy evaluation techniques based on the algebra. Our experiments demonstrate that, for queries with complex conditional and quantified expressions, the proposed algebra results in plans with much better performance than those produced by the state-of-the-art algebras. For relatively simple queries, the proposed methods are expected to yield plans with comparable performance.

Keywords: native XML databases, XML query optimization, query algebras.

1. Introduction

High-level declarative query languages are one of the most important tools offered by database management systems. These languages have great expressive power and are easier to use than conventional programming languages. Modern query execution engines contain sophisticated query optimizers that transform a declarative query into an efficient sequence of low-level operations. The overall optimization process can be presented as a two-phase process, where the first phase is *logical optimization* and the second one, *cost-based optimization* [7]. Logical optimization focuses on rewriting the query into logically equivalent forms that are hopefully more efficient. Cost-based optimizations first creates detailed plans for each rewritten query and then selects the best one based on available statistics of the data.

An *algebra* of operations specifies equivalent operator reorderings. Relational database management systems use common relational algebra, however there is no widely accepted algebra for XML databases. The reasons for this are

partly historical, and partly to do with the complexity of the XQuery language. Known XQuery algebras can be classified according to the optimization phase that they target, as either *rule-based* (RB) or *cost-based* (CB).

RB algebras target the logical optimization phase. The operations of such algebras support both node-at-a-time (NT) and set-at-a-time (ST) execution models [10]. The former type sequentially pass execution for each input item to subsequent operations. Operations of the latter type take as arguments sets of tuples, just as relational operators do.

CB algebras require their operations to be cost-estimation friendly. Traditional cost estimation techniques [8] work only for ST operations for which the cardinality of the result can be estimated from the cardinalities of the arguments. For these techniques to apply, conditional predicates (if any) in the ST operations should test for simple value-based conditions, such as value equality or a structural relationship in case of XML (*e.g.* parent-child).

The XQuery language includes quantified and conditional expressions and *where*-clauses, each of which could involve potentially expensive predicates. Lazy evaluation of the query, *i.e.* performing only those operation evaluations that are necessary for producing the result, hence turns out to be an important technique. While NT-strategy naturally supports lazy evaluation, ST requires entire sets of input to be explicitly pre-evaluated, that in turn can lead to unnecessary computations.

In this paper, we propose algebraic transformations that generate more efficient ST plans by pushing selective operations closer to the leafs of the query plan. Unlike in the relational algebra, these algebraic transformations for XQuery are non-trivial, as they require additional logical-plan transformations in order to preserve correctness. An algebra that facilitates such transformations is the focus of this paper. Our algebra, XAnswer, covers all the XQuery constructs except recursive functions.

We do not discuss cost estimation techniques in detail, instead focusing on logical transformations of the inherently more efficient ST plans. We experimentally demonstrate performance gains for queries with complex conditional and quantified expressions by executing ST-operations in the order discovered using the proposed transformations.

The remainder of the paper has following structure. In the section 3 we define the XAnswer algebra and its main operations. In the section 4 we describe our rules for expression construction and their normalization. The expression construction rules map XQuery to XAnswer. Since the proposed rules produce redundant operations, in the section 5 we discuss transformations that reduce redundancy and enable lazy evaluation. The section 6 presents our experimental validation.

2. Related work

XML-query optimization has been the subject of recent intensive research. Early approaches [1, 2] are based on the algebra for XML Query introduced

in [18] and utilize rewriting rules over expressions of XQuery Core [6]. Such approaches, while are feasible for relatively simple queries, yield poor performance on queries with nested expressions because of use of the use of NT-style operations.

Later approaches fall into two groups: *tree-based* and *tuple-based*. These approaches differ in the kind of elements their operations are defined over. The former use tree-based algebras [4, 9] which operate on ordered sets of trees. The latter use relational-like structures, sets of tuples.

The operations used in tree-based approach generate trees for intermediate results and use pattern matching over those trees.

While most of the tuple-based approaches require the sets of tuples to be ordered, others such as the query algebra [14] avoid this limitation. While some algebras for tuple-based approach assume that tuples contain only atomic values [14], others permit values to be sequences [13, 15, 19], and some others [11, 16] allow values to be sets of tuples.

Tuple-based algebras enable traditional optimization techniques known from relational databases. Significant performance gains can be obtained by using *query unnesting* [11, 13], which is special transformation to substitute NT-operations with combinations of ST-operations. This gain is due to the ability to exploit hash join for implementing ST-operations, whereas for NT-operations a nested-loop join is likely the only possible implementation. One of the most aggressive unnesting schemes is used in Galax [13]. For query translation it uses special NT-operations *MapConcat*, *MapFromItem* and *Cond* for conditional expressions (*if-then-else*). It also defines an ST-operation *GroupBy*. Query unnesting is based on the idea of introducing the *GroupBy* operation where possible (actually, inside operations corresponding to *for*-clauses). Then, using other rules, *GroupBy* and *MapConcat* are swapped to replace *MapConcat* with cartesian product (or join) when possible.

Recent algebras [13, 15] are unable to unnest certain classes of nested expressions, instead being forced to utilize NT-operations. These operations appear in selections with complex predicates, as well as in quantified (*some* or *every*) or conditional expressions.

Let's consider a FLWOR with *if-then-else* clause:

```
for \$i in A/B
let \$b := \$i/D
return if \$i/c then \$b/T else \i/F
```

Translation of this query with NT-style condition, like in earlier work, will lead to the use of inefficient nested loop join and the lack of support for lazy evaluation will cause redundant execution of $\$i/D$ even if $\$i/c$ is *false*.

3. Algebra overview

In this section we introduce *XAnswer*, our tuple-based algebra, in which all operations can be *set-at-a-time (ST)*. Our algebra, has some similar components to XAT [19] and Galax. Just like XAT and Galax, its operations are defined over ordered sets of tuples. Each tuple is a set of items that can be either a single value (XML atomic value, XML-node [17]) or a sequence of single values. We call this structure an $envelope$ and define it formally:

Definition 1 *Envelope ($< he|be|re >$) is a triple of header* he*, body* be *and result attribute* re*. Header* he *is an unordered set of unique-whitin-this-header names called attributes (A). Result attribute* re *is an attribute from* he*. Body* be *is an ordered set of tuples (τ), where each tuple is a set of pairs (A, v) for each attribute* A *of the header. v is either a single value or a sequence of single values.*

In what follows, we will denote tuples as $\tau(e)$, where e is a sequence of values of corresponding tuple pairs. Envelopes are denoted as $< h|\tau(e_1)\ldots|r >$, where h is the header, r is the result attribute, $\tau(e_i)$ forms envelope body, and e_i denotes the sequence of values corresponding to the $i-th$ attribute within h. The signature $< || >$ is used for the empty envelope.

Envelopes are *similar* ($\simeq$) if they have identical headers and their bodies differ only in order of tuples. An envelope *includes* ($\succeq$) another envelope if they have identical headers and if the the body of the first contains all the tuples of the latter.

We continue with the description of the operations that form an algebraic basis. Next, we briefly introduce additional operations that simplify notations or support more efficient implementation.

3.1. Basic operations

Basic operations are presented in the Table 1. To uniquely identify new attributes, we use the function $nextId()$ to generate the attribute's names. An operation for attribute renaming is trivial, and we do not list it among the operations in the table.

Along with relational-like operations that appear in other tuple-based algebras [13, 14, 19], XAnswer adds the *union* operation. Unlike its relational counterpart, XAnswer's *union* does not remove duplicates.

In XAnswer, we introduced a left-outer-join operation instead of expressing it using selection, cross product, and union operators. This is so because envelopes' bodies are ordered and may contain duplicates. Using the left-outer-join along with selection and union, it is possible to express other set operations, namely intersection and subtraction.

To enable query unnesting, previous tuple-based algebras [13, 19] use tuples grouping. They evaluate a nested query in ST-fashion instead of NT, group

Table 1. Main algebraic operations

Operation name	Input	Output
Unary operations		
Function execution (f_f)	$< h\|\tau(e_1)\dots\tau(e_n)\|r >$	$< h, i = nextId()\|\tau(e_1, f(e_1))\dots\tau(e_n, f(e_n))\|i >$
Selection (σ_{pr})		$< h\|\tau(e'_1)\dots\|r >$, where $(e')_i \subseteq (e)_i$ and $pr(e_i) = true$
Projection ($\pi_{h'}$)		$< h'\|\tau(e'_1\ \|_{h'})\dots\|r\ \|_{h'} >$, where $h' \subseteq h$
Sort ($sort_{h'}$)		$< h'\|\tau(e'_1)\dots\|r >$, where $(e')_i = Sort_{h'}(e)_i$
Index ($index_i$)		$< h, i\|\tau(e_1, 1)\dots\tau(e_n, n)\|r >$
Nest ($nest_{h'}$)		See example table 2 and comments in section 3.1
Unnest ($unnest_{h'}$)		See example table 2 and comments in section 3.1
Duplicate (dup_{h_i})		$< h, nextId()\|\tau(e_1, e_1\ \|_{h_i})\dots\|r >$, where $h_i \in h$
Binary operations		
Union ($\bigcup$)	$< h\|\tau(e_1)\dots\|r >$, $< h\|\tau(e'_1)\dots\|r >$	$< h\|\tau(e_1)\dots\tau(e_n), \tau(e'_1)\dots\tau(e'_m)\|r >$
Cross product ($\times$)	$< h\|\tau(e_1)\dots\|r >$, $< h'\|\tau(e'_1)\dots\|r' >$	$< h, h'\|\tau(e_1, e'_1)\dots\tau(e_1, e'_n), \tau(e_2, e'_1)\dots\|r' >$
Left outer join ($\bowtie^l_{pr}$)		$< h, h'\|\tau(e_1, e''_1)\dots\tau(e_1, e''_n), \tau(e_2, e''_1)\dots\|r' >$, where if $(pr(e_i, e'_i) = true)$ then $e''_i = e'_i$ else $e''_i = ()$

the results of the query into sequences, and append the sequences to corresponding tuples. In [13], this is done using a complex operation called *group by*. We define a lower-level operation $nest$ along with an $unnest$ operation that ungroups grouped tuples. The semantics of these operations is shown in the Table 2. While $nest$ and $unnest$ appear in [19], previously proposed tuple groupings are different from ours. Our $nest$ and $unnest$ operations are not complementary (see Table 2). Note that if h is empty in $nest_h$, all tuples fall into one group, and the resulting envelope has a single tuple with sequences of values for each attribute.

Table 2. Input and output of algebraic operations

(a) $nest_{A,B}$

Input			Output		
A	B	C	A	B	C
a_1	b_1	c_1	a_1	b_1	c_1
		c_2			c_2
a_1	b_2	c_3			
a_1	b_1	c_2	a_1	b_2	c_3

(b) $unnest_{A,B}$

Input			Output		
A	B	C	A	B	C
a_1	b_1	c_1	a_1	b_1	c_1
		c_2	a_1	b_1	c_2
		c_2	a_1	b_1	c_2
a_1	b_2	c_3	a_1	b_2	c_3

(c) dup_B

Input		Output		
A	B	A	B	B'
a_1	b_1	a_1	b_1	b_1
a_2	b_2	a_2	b_2	b_2

(d) $quant^{every}_{true,B}$

Input		Output	
A	B	A	B
a_1	$true$	a_2	true
	$false$		
	$true$		
a_2	$true$		

For duplication of attribute values we use *duplicate* operation (dup_a). The input and output are presented in the Table 2. The operation is used in plan reduction described in the section 5.2 to replace relatively expensive joins.

3.2. Additional operations

In order to simplify notation, we introduce specific leaf algebraic operations, *leaf path step* ($LPS_p := unnest_{re(f)}(f_p(< || >))$) and *leaf constructor* ($LC_c := unnest_{re(f)}(f_c(< || >))$). LPS is used to extract values stored in a document. LC is used to extract XML-values (constants) that are not present in a document (e.g. they occur in the text of a query). In LC_c operation, c is a sequence constructor (e.g. `(1,2,3)`) or an element constructor (e.g. `<a>text</a>`). In LPS_p operation, p is an XPath expression (e.g. `/a/b//c`).

Structural join is the traditional join operation with a structural predicate that specifies the relation between two nodes (e.g. parent-child). If structural predicate is applied to sequences of values, it evaluates to true if there is at least one value satisfying the predicate in each sequence. Unary path step and structural join are used in path expressions mapping.

XAnswer has a basic function operation (f_f). The pattern f specifies the overall structure of the element and places where corresponding tuple values should be substituted. In our notation, we add a special operation called *element constructor*(θ_p), where p is an element pattern which describes the schema of an element.

The purpose of *quantify* ($quant^q_{c,h_i}$) operation is to efficiently implement lazy evaluation (section 5.3). The operation is a specific kind of selection used for controlling the pipeline. A sample input and output of the quantify operation are presented in Table 2.

Addition operation is a special complex operation included in the algebra to avoid unnecessary steps in physical plan. The operation appears during the plan reduction described in the section 5.2. *Addition* is a combination of projection, nest and join: $A \bigoplus_p^v B = nest_{he(A)}(\pi_{he(A) \bigcup v}(A \bowtie_p^l B))$, where p is a predicate and $v \subseteq he(B)$.

4. Plan construction

The plan construction process consists of two steps, normalization and translation. During the process FLWOR expressions are broken into single for- and let-clauses [6], predicates are moved from xpath to where-clauses [5, 11, 13] and complex expressions are removed by introducing new variables [11].

4.1. Normalization

The proposed normalization differs from previous work mostly in the normalization of nested, quantified and conditional expressions.

Below we outline some of our logical transformations. We break up complex expressions via introduction of new variables in such a way that only let-clauses contain them. Any nested FLWOR expression starts with a for-clause. Positional predicates are

```
let $k := for $i in expr
                (for $j in ...)
          return cond
(: if "some" :)
let $r := $k = true
(: if "every" :)
let $r := not($k = false)
return $r
```

Fig. 1. Quantifiers normalization

moved to where-clauses. Quantified expressions are replaced with FLWORs (see Figure 1).

4.2. Translation

An algebraic expression is constructed in a top-down fashion. The Table 3 illustrates some of the expression construction rules. We assume that P is a subexpression that was constructed on a previous step and E is an expression corresponding to the bound expression ($expr$) of a let- or a for-clause, $he(E)$ or $re(E)$ means header or return attribute extraction from the resulting envelope of the expression E.

Table 3. Construction rules

	XQuery	XAnswer
1	`for $i in` $expr$	$P \times \pi_{re(E)}(E)$.
2	`let $i :=` $expr$	$(P \times nest_{()}(\pi_{re(E)}(E))$.
3	`for $i in` $expr(v)$	$\pi_{he(P)\cup re(E)}(index_i(P) \bowtie_{i=j} E(unnest_v(index_j(P))))$.
4	`let $i :=` $expr(v)$. $Expr$ is xpath expression that has a reference to previously defined variable	$\pi_{he(P)\cup re(E)}(nest_{he(P)\cup i}(index_i(P) \bowtie^l_{i=j} \pi_{re(E)\cup j}(E(unnest_v(index_j(P))))))$.
5	`let $i :=` $expr(v_1 \dots v_n)$. $Expr$ is not xpath and has references to previously defined variables	$\pi_{he(P)\cup re(E)}(nest_{he(P)\cup i}(index_i(P) \bowtie^l_{i=j} E(index_j(P))))$.
6	$expr(v_1 \dots v_n)$. $Expr$ is logical or algebraic expression operating with variables $v_1 \dots v_n$.	$f_{ep,v_1 \dots v_n}(E)$.
7	`where` $expr$.	$\sigma_{re(E)=true}(E(P))$.
8	`order by` $v_1 \dots v_n$.	$sort_{v_1 \dots v_n}(P)$.
9	`/axis::node-test`.	$P \bowtie_{::axis} LPS_{node-test}$.
10	$E_1 op E_2$. Op is one of *intersect, union, except*.	$P_1 op P_2$, $op \in (\cup, \cap, \setminus)$.
11	`if` $expr_c$ `then` $expr_t$ `else` $expr_f$.	See section 5.3.

Note that, if $P =< \,||\, >$, in the rules *1,2,9* only right operand of $\bowtie$ and $\times$ should be considered as a resulting subplan.

Example 1

Below we demonstrate sample plan construction process. Let's denote $h_A = re(LPS_A)$ and $h_B = re(LPS_B)$.

Step	Rule	Input	Output
0		for $v in A let $m := $v/B	$P =< \| \| >$
1	1	`for $v in A` ; $P =< \| \| >$	$P = \pi_{re(E)}(E)$
2	9	`A` ; $P =< \| \| >$	$E = LPS_A$
3	1	`for $v in A` ; $P =< \| \| >$; $E = LPS_A$	$P = \pi_{re(LPS_A)}(LPS_A)$
4	5	`let $m := $v/B` ; $P = \pi_{re(LPS_A)}(LPS_A)$	$\pi_{he(P)\cup re(E\|_{P'})}(nest_{he(P)\cup i}(index_i(P) \bowtie^l_{i=j} \pi_{re(E\|_{P'})\cup j}(E\|_{P'}))), P' = unnest_v(index_j(P))$
5	9	`$v/B`; $P' = unnest_v(index_j(P))$	$E\|_{P'} = P' \bowtie_{::child} LPS_B$
6	5	`let $m := $v/B`; $P = \pi_{re(LPS_A)}(LPS_A)$; $E\|_{P'} = P' \bowtie_{::child} LPS_B$	$P = \pi_{h_A\cup h_B}(nest_{h_A\cup i}(index_i(T) \bowtie^l_{i=j} \pi_{h_B\cup j}(unnest_v(index_j(T)) \bowtie_{::child} LPS_B))), T = \pi_{h_A}(LPS_A)$
Optimized: $\pi_{h_A \bigcup h_B}(nest_{h_A\cup i}((index_i(LPS_A)) \bowtie^l_{child} LPS_B))$			

5. Optimization

Efficient physical plans are derived by applying optimizing transformations to the constructed algebraic plans. The logical transformations of algebraic plans also enables further physical plan optimization.

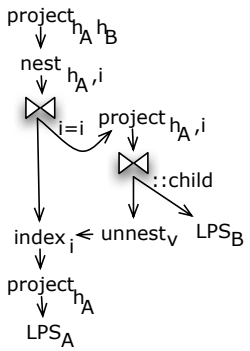


Fig. 2. The plan from Example 1

We explain all our transformations with the help of a directed graph in which vertices (nodes) represent algebraic operations and edges connect those operations to their operands (the Figure 2). In Example 1 above, $index_i(\pi_{h_A}(LPS_A))$ and $index_j(\pi_{h_A}(LPS_A))$ produce envelopes which differ only in headers. Such envelopes can be converted to each other with the *HR* operation. In such cases we consider the two subplans identical and leave only one in the graph (see Figure 2).

Definition 2 *An* operation block *is a subgraph that has no more than one node (*output node*) with incoming external edges, and has all outgoing external edges (if any) leading to only one external node. The targets of the outgoing edges are called* input nodes*.*

Below we consider an expression $B_1(B_2)$, where B_1 and B_2 are operation blocks, and all *input nodes* of B_1 have outgoing edges lead to the *output node* of B_2.

Definition 3 *An operation block B is* non-reducing *if an envelope En passed to the block is not empty and $\pi_{he(En)}(B(En)) \succeq En$. An operation block B is* non-modifying *if $\pi_{he(En)}(B(En)) \simeq En$.*

Non-reducing operation blocks keep all tuples of the input *envelope*. *Non-modifying* operation blocks are non-reducing blocks that do not add new tuples to the input *envelope*. Non-reducing and non-modifying properties can be naturally generalized from operation blocks to unary operations, join and left outer join (for joins, the left operand is considered as the input in the sense of Definition 3). Non-modifying operations include *duplication*, *function*, *sort*, etc. Non-reducing operations additionally include *left outer join*. *Join* and *selection* are reducing operations because tuples of the (left) operand may be eliminated.

Cycles appear in the graph as a result of constructing parts of the plan corresponding to for- (*for-cycles*) and let-clauses (*let-cycles*) that refer to previously defined variables. *Let-cycle* are non-modifying due to use of *nest* and *left outer join* (see rule 5 of Table 3). *For-cycles* are reducing operation blocks.

Definition 4 *Two* operation blocks B_1 *and* B_2 *are* independent *if $B_2(B_1(P)) \simeq B_1(B_2(P))$.*

If a block operates with attributes that were added to the header of the resulting envelope by another block, it *depends* on the other block. Independent blocks never depend on each other.

Optimizing transformations listed in the following sections preserve the value of $\pi_{re(P)}(P)$. This is achieved by using an *HR* operation to change the *return attribute* if needed. For compactness we omit usages of an *HR* operation.

5.1. Blocks pushdown

Optimizing transformations include reordering, removal, and replacement of operations. The goal of optimizing reordering is to make the intermediate results created during the plan execution smaller. With respect to this, it is important to perform selective operations as early as possible. In graph terms, it means that these operations should be placed closer to the leaves of the graph (i.e. pushed down). Often, selective operations can be pushed down only along with a containing operation block. We present a blocks pushdown algorithm that enables selective operations pushdown.

We refer to *operation blocks* corresponding to *for-, let-, where-, etc. clauses* as *clause-blocks*. Two *clause-blocks* are independent if one clause of normalized query does not refer to a variable introduced in another.

1. Suppose $P = B_s(B_r)$ is a plan, where B_s and B_r are operation blocks and B_s is a block to push down.

2. Locate a *clause-block* B_m, $B_r = B_e(B_m(B_b))$: B_s depends on B_m. If there is no such B_m, then $B_e = B_r$.
3. Recursively perform the algorithm on B_m, obtaining B'_m.
4. Push down B_s through B_e in order to obtain: $P = B'_e(B_s(B'_m(B'_b)))$.

5.2. Redundant operation reduction

In this section we present logical optimizations that remove redundant operations or group several operations into complex ones. The optimizations include:

- removal of unnest operations;
- introduction of addition operations;
- removal of unnecessary index operations; and
- cycle removal.

Unnest removal is trivial and can be applied directly: $unnest_{h'}(P) \Rightarrow P$ if P does not contain $nest_{h''}$ and $h'' \subset h'$.

The **addition operation** is also introduced directly based on its definition (see section 3.2). The addition operation groups several operations (*nest, join, projection*) into a single one and allows further physical plan optimizations.

Removal of unnecessary index operations is a two-step operation that consists of preliminary reordering of index operations and subsequent index operations removal itself. After unnest removal and introdution of addition operation *let-blocks* are identified by the presence of addition operation. Such blocks are non-modifying operation blocks due to use of *left outer join* and *nest*. Consequently, if an index operation is performed before such block, indexes are retained in the result, and the following transformation is valid:

$$index_i(\pi_{he(P)\cup re(B)}B(index_j(P))) \Rightarrow \pi_{he(P)\cup re(B)\cup i}(B(index_i(index_j(P)))).$$

Subsequent application of this transformation produces a plan that contains sequences of *index* operations. The sequences are replaced with a single *index* operation.

Cycle removal is performed on for- and let-cycles that appear (as implied by the construction rules) whenever the respective bound expressions reference previously defined variables. The bound expressions can be one of three kinds: an xpath expression, a FLWOR expression, or a conditional expression.

If the bound expression of a for- or a let-clause is an xpath. Xpath expressions are mapped using *structural join* and *LPS* operations. Assume B_x is a block corresponding to some xpath. It consists of *structural join* and *LPS* operations. While *structural joins* are reducing operations, they have left outer versions ($\bowtie^l$) that are non-reducing. B_x^l is obtained from B_x by replacing *structural joins* with left outer joins.

We propose the following transformation for *let-cycles*:

$$\pi_{he(P)\cup re(B_x)}(nest_i\left(index_i(P) \bowtie^l_{i=j} (B_x\,(unnest_v(index_j(P))))\right)) \Rightarrow$$
$$\pi_{he(P)\cup re(B^l_x)}(nest_i\left(B^l_x(unnest_v(index_i(P)))\right)).$$

The transformation for *for-cycle* is:

$$\pi_{he(P)\cup re(B_x)}(index_i(P) \bowtie_{i=j} B_x\,(unnest_v(index_j(P)))) \Rightarrow$$
$$\pi_{he(P)\cup re(B_x)}(B_x(unnest_v(dup_v(P)))) \quad .$$

The topmost *join* operation in the original plans of the cycle is used to set up a correspondence between variable values of the query context with the new values obtained within the clause. *Unnest* operation can break the correspondence. To keep it, we use *dup* operation instead of *join* and then perform required unnesting on the introduced attribute. If the *unnest* operation is removed in optimization process, *dup* can also be eliminated.

If the bound expression of a let-clause is a nested FLWOR. The optimization is based on the fact that the topmost join $\bowtie^l_{i=j}$ that organizes a cycle can be removed if it has a non-reducing operation block as a right-hand operand.

After query normalization, each nested FLWOR starts with *for* and ends with *where*, *order by*, and *return*. The last three clauses are mapped with, respectively, σ, $sort$ and π operations. Let an operation block B_1 corresponds to *order by* and *return*, and B_2 corresponds to the sequence of *let-* and *for-clauses*. A plan for complete *let-clause* with a nested *FLWOR* is:

$$\pi_{he(P)\cup re(B_1)}(nest_i(index_i(P) \bowtie^l_{i=j} B_1(\sigma_p(B_2(index_j(P)))))). \qquad (1)$$

$\bowtie^l$ in this plan can be pushed down over B_1. The transformation also modifies *sort* operation to preserve original order.

Let $B_2 = B_{F_1}(\ldots B_{F_n})$, where B_{F_i} are *for-* or *let-blocks*.

There are two cases: $\forall i\ B_{F_i}$ is not dependent on P and $\exists i\ B_{F_i}$ is dependent on P. In case B_{F_i} is not dependent on P according to normalization rules, B_{F_1} is a *for-block*. According to construction rules and independence with P, $F_1 = P \times B_x$, where B_x is a block corresponding to a xpath expression. Since B_{F_2} is also independent of P, it is easy to see that $B_{F_2}(P \times B_{F_1}) = P \times B_{F_2}(B_{F_1})$. After application of this transformation sequentially for expression (1), combining σ and $\times$ into $\bowtie^l_p$, and removing outer $\bowtie^l_{i=i}$, we obtain:

$$\pi_{he(P)\cup re(B_1)}(nest_i(B_1(index_i(P) \bowtie^l_p B_{F_n}(\ldots(B_{F_1}))))).$$

In case B_{F_i} is dependent on P, we use the fact that *let-cycles* are non-modifying blocks (see section 5) and *for-cycles* are reducing due to their topmost *join* operations.

Let's denote $B^l_2 = B_{L_1}(\ldots B_{L_n})$, where $B_{L_i} = B_{F_i}$ if B_{F_i} is a *let-block* and $B_{L_i} = B^l_{F_i}$ if B_{F_i} is a *for-block*. $B^l_{F_i}$ is obtained by replacing its topmost *join* with *left outer join* in B_{F_i}. We perform (1) to the following:

$$\pi_{he(P)\cup re(B_1)}(nest_i(B_1(\sigma_p(B^l_2(index_i(P)))))).$$

Cycle removal is applied recursively for inner cycles.

If the bound expression of a let-clause is a conditional expression. If the bound expression is a conditional expression, transformations similar to those described for xpath expression case can be applied because the corresponding operation block is non-reducing. We omit details due to space limitations.

5.3. Transformations for lazy evaluation

Lazy evaluation is preferable in quantified expressions, positional predicates, and conditions of where, if-then-else, typeswitch clauses. In XAnswer, these types of expressions are mapped with ST-operations. When a physical plan is executed, the operands of the ST-operations are pre-evaluated, thus potentially making expensive unnecessary computations. To reduce such computations, certain support, e.g. pushing down of selective operations closer to leaves, is required in the process of algebraic plan modification.

Transformations of quantified expressions and positional predicates. Positional predicates allow physical plan execution to avoid the calculation of the whole operand of the operation. For quantified expressions, if one satisfying (or not satisfying - depending on whether “some” or “every” is used) value is found, there is no need to evaluate further the operand of the quantified operation. *Position* and *quant* operations are introduced into algebraic plans to allow pipelined evaluation at the physical level.

According to normalization and construction rules, positional predicates are mapped using selections and index operations. If $h_i = re(index_i)$ and p is a positional predicate $\sigma_{p(h_i)}$ is replaced with $position_{p(h_i)}$.

A similar mapping is used for quantified expressions. Unlike the case of positional predicates, the selection is performed over nested values. According to the proposed optimizations, the nestings are included in *addition* operations. Then, if $h_i = re(\bigoplus)$, an expression $\sigma_{p(h_i)}$ is substituted with $quant_{p(h_i)}$.

Transformations of conditions. Consider a where-clause *where A op B*. A and B are logical expressions or terminals (variable references). Op is a logical operator and or or. We assume without loss of genrality that A and B are terminals; if not, the process described below can be applied recursively. B_A and B_B are operation blocks corresponding to A and B respectively. In case of *if-then-else* we use B_C, B_T and B_F to denote corresponding operation blocks for the *condition*, *then-*, and *else-branches*. We assume that other blocks in the plan are independent with A and B. Without this assumption lazy evaluation does not bring performance benefits and there is no much point in applying this transformation.

Transformation of conditions is preceded by sequential pushdown of B_A, B_B (*where-clause*) or B_C, B_T, B_F (*if-then-else-clause*) using the algorithm described in section 5.1.

Logical “and” in a where-clause. Selection decomposition is:

$$\sigma_{A\&B}\left(B_B(B_A(P))\right) = \sigma_A\left(\sigma_B(B_B(B_A(P)))\right).$$

Due to normalization, B_A and B_B are independent blocks.

Blocks B'_A, B'_B, P' (possibly empty) are obtained by pushdown of B_A and B_B according to the algorithm in section 5.1: $B_B(B_A(P)) \Rightarrow B'_B(B_B(B'_A(B_A(P'))))$. B'_B has blocks depending on it, all blocks are independent with B'_A. We also assume that B'_B contains *sort* operation that restores original order of tuples. The transformation result is:

$$B'_B(\sigma_B(B_B(B'_A(\sigma_A(B_A(P')))))).$$

Logical "or" in a where-clause. Since $A \cup B = A \cup (B \backslash A)$, the selection decomposition is:

$$sort_i[\pi_{he(P)\cup i}\,(\sigma_A(B_B(B_A(index_i(P)))))$$
$$\cup\pi_{he(P)\cup i}\,(\sigma_{\neg A\&B}(B_B(B_A(index_i(P)))))].$$

Block pushdown then gives us:

$$B'_B(\pi\,(\sigma_A(B_A(P'))) \cup \pi\,(\sigma_B(B_B(B'_A(\sigma_{\neg A}(B_A(P')))))).$$

An algebraic expression for *if-then-else clause case* is:

$$sort_i[HR_{(re(B_T)\to k)}(\sigma_C(P'')) \cup HR_{(re(B_F)\to k)}(\sigma_{\neg C}(P''))],$$

where $P'' = B_F(B_T(B_C(index_i(P))))$.

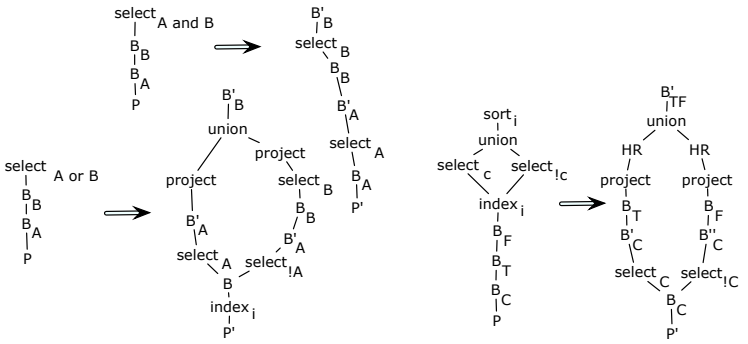
Blocks pushdown results in:

$$B'_{TF}\,(HR(\pi(B_T(B'_C(\sigma_C(B_C(P'))))))) \cup HR(\pi(B_F(B''_C(\sigma_C(B_C(P'))))))\,.$$

The blocks B'_C, B''_C, B'_{TF}, and P' are obtained by pushing down B_C, B_T, and B_F blocks. Only B_T is not independent with B'_C and B_F is not independent with B''_C. As with B'_B in the previous case, B'_{TF} contains *sort* operation that restores original order of tuples.

Similar optimization, whose description we omit here, is used for *typeswitch-clauses*.

The proposed optimizations are presented on the Figure 3.



(a) *Where-clause* decomposition **(b)** *If-then-else* decomposition

Fig. 3. Optimizations for lazy evaluation

6. Experiments

Our experimental implementation of XAnswer is based on XML DBMS eXist [1]. EXist storage provides DLN-indexes [12] as unique ids for all nodes and all

structural operations can be evaluated using just the ids. Thus, envelopes were implemented as arrays of tuples that contain either atomic values (e.g. string, integer), ids, or sequences of the ids or atomic values.

The code was written in Java and experiments were conducted under Windows XP on Intel Dual Core T2300 @1.6 GHz with 2Gb of main memory. The XMark benchmark [3] was used as data set.

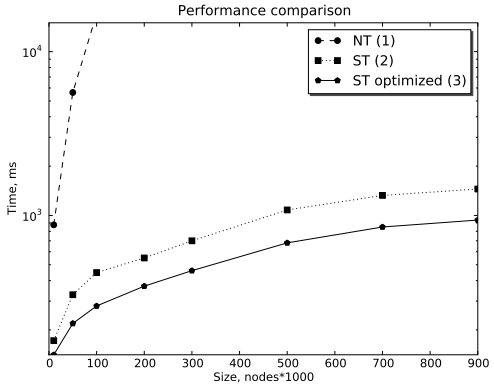


Fig. 4. Experiment E_1

Known approaches can perform unnesting for relatively simple queries with nested FLWORs. That kind of optimization allows switching from NT to ST-execution, which can bring dramatic performance gain, since it allows the hash join algorithm to be used instead of nested loops. Using normalization techniques similar to ours, known approaches achieve performance comparable to XAnswer. However, known approaches cannot perform unnesting when the query contains quantifiers, where-clauses, or conditional clauses with nested FLWORs. Moreover, many such cases require lazy evaluation.

To compare other approaches with ours, we implemented NT-style plan creation for conditional clauses as well as ST-operations of XAnswer. In experiments we focused only on such queries over the XMark's data set. We present details for only two experiment sets. The first set (E_1) evaluates performance of naive NT-execution (the default plan generated by eXist), ST (initial XAnswer plan), and optimized-ST (optimized XAnswer plan) on a typical XMark query (such as query Q8) where earlier approaches can perform unnesting.

The second set (E_2) evaluates performance of naive NT-execution (the default plan generated by eXist), partial NT (having NT-style condition operation and ST for clauses without conditions and quantifiers), ST (XAnswer plan without lazy evaluation), and optimized-ST (optimized XAnswer plan) for a query with a conditional clause that requires lazy evaluation.

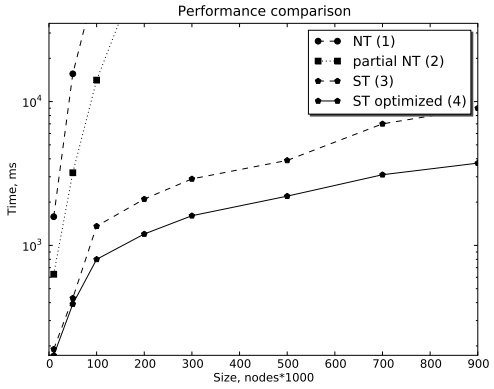


Fig. 5. Experiment E_2

Query 1 A query example from the set E_2

```
let $doc := doc('doc.xml')
for $ca in $doc//closed-auction
let $b :=
  for $p in $doc//person
  where $p/@id = $ca/buyer/@person
  return $p
return
  if ($ca/price >200)
  then (
    let $c := \b/country
    let $n := $b/name
    return <rb>{$c,$n}</rb>)
  else (
    let $item :=
      for $it in $doc//regions//item
      where $it/@id=$ca/itemref/@item
      return $it
    let $l := $item/location
    let $n :=$item/name
    return <ri>{$l,$n}</ri>)
```

Figure 4 demonstrates performance comparison for E_1. The huge performance gain of ST-plan compared to NT is observed due to use of hash join over input sets instead of evaluating inner expression in a loop. Since the query has a relatively simple where-clause that does not require lazy evaluation, the ST-optimized plan does not demonstrate major performance gain. The increase is mostly achieved through removal of a cycle and use of addition operation, which reduces amount of hash operations in the physical plan.

Figure 5 demonstrates performance evaluation for E_2. Note, that the time axis has a logarithmic scale. Since partial NT-plan utilizes ST-execution for the first nested FLWOR, the performance gain compared to naive NT is similar to the one obtained in E_1. But NT-style condition leads to evaluation of nested expressions in a loop, which causes difference in performance with the ST plan. The gain of the optimized-ST is due to the smaller sizes of intermediate results. The sizes are smaller because evaluation of the first nested FLWOR *then*- and *else*- branches happens only when teh condition has corresponding values: for example FLWOR and *then* are evaluated only when $price > 200$ in line 7 of the code in Query 1.

7. Conclusion

Heavy query workloads in modern applications require algebras that support cost-based optimization. Set-at-a-time execution is a natural way to support cost estimation. While set-at-a-time execution appeared in earlier work on efficient XQuery processing techniques, known algebras combine it with node-at-a-time to cover the complete set of XQuery operations.

In this paper we presented an algebra with only set-at-a-time operations that supports compilation of complete XQuery except recursive functions. We introduced optimization rules eliminating unnecessary operations and enabling lazy evaluation. It was experimentally shown that for complex queries with conditional and quantified expressions the proposed methods yield plans with much better performance than those that can be obtained with known algebras.

References

1. Exist DBMS Home Page. http://exist-db.org/, exist DBMS Home Page
2. Sedna DBMS Home Page. http://modis.ispras.ru/sedna/, sedna DBMS Home Page
3. XMark Benchmark Home Page. http://monetdb.cwi.nl/xml/, xMark Benchmark Home Page
4. Chean, et al.: From tree patterns to generalized tree patterns: On efficient evaluation of xquery. In: VLDB. pp. 237–248 (2003)
5. Deutsch, A., et al.: The next framework for logical xquery optimization. In: VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases. pp. 168–179. VLDB Endowment (2004)
6. Fankhauser, P.: Xquery formal semantics state and challenges. SIGMOD Rec. 30(3), 14–19 (2001)
7. Ioannidis, Y.: Query optimization. ACM Comput. Surv. 28(1), 121–123 (1996)

8. Ioannidis, Y.: The history of histograms. In: VLDB '2003: Proceedings of the 29th international conference on Very large data bases. pp. 19–30. VLDB Endowment (2003)
9. Jagadish, H., et al.: Tax: A tree algebra for xml. In: DBPL '01: Revised Papers from the 8th International Workshop on Database Programming Languages. Springer-Verlag, London, UK (2002)
10. Jagadish, V., et al: Timber: A native xml database. The VLDB Journal 11(4), 274–291 (2002)
11. May, N., et al.: Nested queries and quantifiers in an ordered context. Proc. ICDE Conference, Boston, USA pp. 239–250 (2004)
12. Meir, W.: Index-driven xquery processing in the exist xml database (2006)
13. Re, C., et al.: A complete and efficient algebraic compiler for xquery. In: ICDE '06: Proceedings of the 22nd International Conference on Data Engineering. p. 14. IEEE Computer Society, Washington, DC, USA (2006)
14. Sartiani, C., Albano, A.: Yet another query algebra for xml data. In: IDEAS '02: Proceedings of the 2002 International Symposium on Database Engineering and Applications. pp. 106–115. IEEE Computer Society (2002)
15. Wang, S., et al.: Optimization of nested xquery expressions with orderby clauses. In: ICDEW '05: Proceedings of the 21st International Conference on Data Engineering Workshops. IEEE Computer Society (2005)
16. Weiner, A., Mathis, C., Haerder, T.: Towards cost-based query optimization in native xml database management systems. In: Proceedings of the SYRCoDIS 2007 Colloquium on Databases and Information Systems (2008)
17. Extensible Markup Language (XML) 1.0 (Second Edition) (6 October 2000), w3C Recommendation
18. XML Query Algebra (15 February 2001), w3C Recommendation
19. Zhang, X., Pielech, B., Rundesnteiner, E.: Honey, i shrunk the xquery!: an xml algebra optimization approach. In: WIDM '02: Proceedings of the 4th international workshop on Web information and data management. pp. 15–22 (2002)

Maxim Lukichev is Sr. Software Engineer at Whodini, Inc. Previously Maxim was a researcher at HP Labs Russia and was a member of Information Management Research group at St.Petersburg University. Maxim received his Masters and Ph.D. in Comp.Sci. from St.Petersburg University. His research interests are in the area of query optimization, indexing and data structures, semantic data and information extraction.

Boris Novikov is a professor and a head of Operations Research Laboratory and Information Management Research Group at St.Petersburg University. He also works as a database expert for the industry. Boris received his PhD in Comp.Sci.in 1981, and Dr. Sci. degree in 1994 from St.Petersburg University. His research interests include design of database systems, transactions, indexing and data structures, query processing and optimization, performance tuning, and management of distributed heterogeneous information resources.

Pankaj Mehra is SVP and CTO at Whodini, Inc. Pankaj also serves as an industry advisor to CodeX, a joint program of Stanford Law School and Computer

Science. He is on the editorial board of IEEE Internet Computing magazine. Previously Pankaj was a Chief Scientist of HP Labs Russia, and before that faculty member at IIT Delhi. Pankaj received his B Tech from IIT Delhi, and his Ph.D. in Computer Science from University of Illinois at Urbana-Champaign.

Received: August 4, 2010; Accepted: July 1, 2011.