XML Query Optimization in the Presence of Side Effects

Giorgio Ghelli
Università di Pisa, Italy
ghelli@di.unipi.it

Nicola Onose
UCSD, La Jolla, CA, USA
nicola@cs.ucsd.edu

Kristoffer Rose
IBM Research, NY, USA
krisrose@us.ibm.com

Jérôme Siméon
IBM Research, NY, USA
simeon@us.ibm.com

ABSTRACT

The emergence of database languages with side effects, notably for XML, raises significant challenges for database compilers and optimizers. In this paper, we extend an algebra for the W3C XML query language with operations that allow data to be immediately updated. We study the impact of that extension on logical optimization, join detection, and pipelining. The main result of this work is to show that, with proper care, a number of important optimizations based on nested relational algebras remain applicable in the presence of side effects. Our approach relies on an analysis of the conditions that must be checked in order for algebraic rewritings to hold. An implementation and experimental results demonstrate the effectiveness of the approach.

Categories and Subject Descriptors

H.2 [**Database Management**]: Languages; I.1.3 [**SYMBOLIC AND ALGEBRAIC MANIPULATION**]: Languages and Systems—*Evaluation strategies*

General Terms

Languages, Performances

1. INTRODUCTION

In order to facilitate Web development, a number of languages blending database and programming language capabilities have recently been proposed [5, 12, 15, 19, 25]. Two well-known examples are LINQ [19], which extends .NET languages such as C# or Visual Basic with querying primitives, and XQueryP [5], which extends the W3C XML Query language [2] with imperative features. Such languages aim at simplifying existing Web development practices, which typically rely on several different languages used at different tiers. The ability to blend data processing and programming capabilities frees the developer from the need to rely on low-level APIs for data access, but also raises significant challenges for compilers. Many database compilers already support expressive languages such as object-oriented languages [8], PL/SQL [11], XSLT [7], or XQuery [2], however, most of the work on query optimization has focused on languages without side effects. Side effects are essential in order to support key programming extensions, such as updates and variable assignment in XQueryP [5], or method calls in object languages [8].

In this paper, we propose techniques to adapt existing database compilers to support side effects in XQuery while preserving essential optimizations based on algebraic rewritings. Surprisingly, there has been very little work in this area in the past. One notable exception is [10] that uses a state monad [22] to support side effects in a nested-relational calculus. However, optimization at the algebraic, logical and physical level is not addressed. To the best of our knowledge, we provide the first treatment of side effects for a nested-relational algebra. Due to space constraints, we limit our scope to updates applied during query evaluation, and leave procedural extensions (notably variable assignment) for future work. We start with a motivating example.

Use case. Consider a simple scenario inspired by the sample retail application [1] provided with BEA's AquaLogic DSP [4]. This scenario assumes two XML data sources located at an on-line retailer site, named 'customers.xml' and 'orders.xml', made accessible by two XQuery functions, getCustomers and getUnconfirmedOrders. A customer can place orders, which are put on hold until they are confirmed by that customer. The application also maintains, through updates, access timestamps for customers data:

```
1  declare updating function getCustomers() {
2    for $c1 in doc('customers.xml')//customer
3    return
4    ( do replace value of $c1/timestamp with gettime(),
5      $c1 )
6  };
7  declare function getUnconfirmedOrders($cid) {
8    for $po in doc('orders.xml')//purchase-order
9    where $po/cid = $cid and $po/status = 'unconfirmed'
10   return $po
11 };
12 for $c in getCustomers()
13 let $u := getUnconfirmedOrders($c/cid)
14 return
15   <new-orders customer='{$c/cid}'
16       accesstime='{$c/timestamp}'>
17     {$u}
18   </new-orders>
```

In the absence of the **do replace** update on line 4, standard query optimizers would identify the query as a typical

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

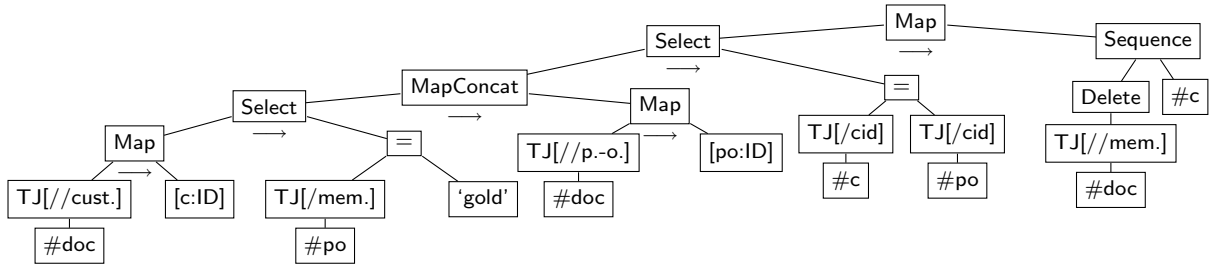

Figure 1: Original plan.

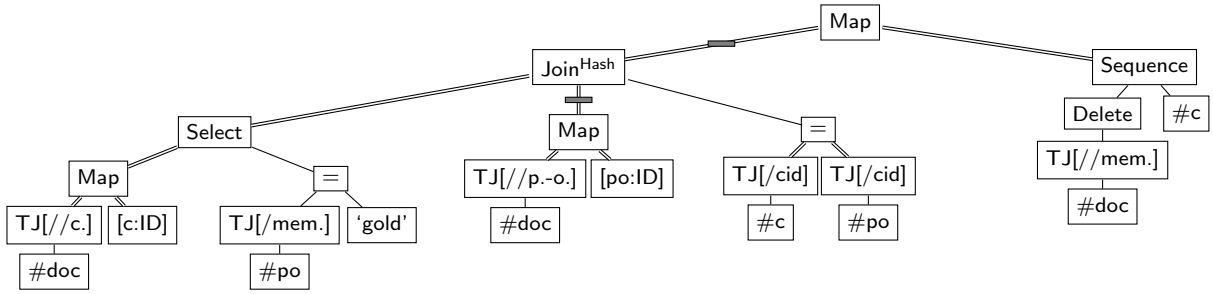

Figure 2: Optimized plan; hollow lines denote set arguments, a gray bar denotes materialization.

case of *outer join* between customers and orders (after function inlinining and using query unnesting techniques). In Section 3, we will see how similar nested queries can be optimized. We now give an overview of our approach.

Queries with side effects. We consider an extension of XQuery 1.0 [2] with simple update expressions. We adopt a semantics that imposes a strict left-to-right evaluation order and the immediate application of updates, in the style of XQueryP [5].

We use a simple, artificial example to describe the technical challenges and illustrate our approach. The following query returns the set of orders for customers with gold membership, and if non-empty, removes all membership level information.

```
1 for $c in $doc//customer[membership = 'gold']
2 for $po in $doc//purchase-order
3 where $c/cid = $po/cid
4 return ((do delete $doc//membership), $c)
```

The language semantics specifies the following evaluation order: (i) retrieve the list of customer elements; (ii) expand the list of "gold" customers to a list of tuples with each customer paired with all possible purchase-orders; (iii) filter away the tuples that violate the predicate; (iv) for each tuple first remove all membership element nodes in the document and then return the customer element (the deletion has no effect from the second tuple onward). In this query, the optimizer should be able to use a join plan, since the updates occur only after all previous clauses are evaluated.

Logical optimization. The query is first compiled into a logical plan, shown in Figure 1, using a nested relational algebra with extensions for XML processing [21,23], as well as updates [14]. In addition to the standard nested-relational Select and Map, which applies its second operand to all items returned by its first operand, specific operators used in that example include TreeJoin (denoted TJ on the figure) for path navigation and Delete for node deletion. MapConcat corresponds to the traditional dependent product, and we use an arrow to indicate information flow from the first operand to the second. [a:ID] builds a one-field tuple and $\#a$ selects the a field from the input tuple. Again, the presence of updates requires the semantics of the algebra to specify an explicit evaluation order. We assume here an *eager* semantics in which every operator completely evaluates its first operand before starting the evaluation of the next operand.

In the absence of side effects, an optimizer would now turn the nested loop (MapConcat) operator into a Cartesian product, and would then combine it with the selection to get a join. Unfortunately, such a rewriting may be unsound in the presence of side effects. For instance, if the second Map operand of the MapConcat performs some insert, there must be as many inserts as there are items resulting from the first Map operand. We say that the operand is not *idempotent*: the store effects of repeated evaluations differ from those of a single evaluation, and hence, in this case, the nested loop must be retained. More generally, to be valid in presence of side effects, most classical rewritings require that some conditions are satisfied by the components of the rewritten plan. We show that three side conditions are sufficient: *commutativity*, meaning that the evaluation order of two subplans can be exchanged, the aforementioned *idempotency*, and, in some cases, *purity*, indicating the absence of side effects. We also discuss how these conditions can be inferred statically using existing algorithms based on path analysis [16]. Using those techniques, our optimizer can rewrite the plan above into the join-based plan shown in Figure 2.

Physical planning. At the physical level, one essential difference with a traditional compiler relates to pipelining. In a *pure* expression, the execution of the different operators can be freely *interleaved* (or *pipelined*), since their evaluation

order is irrelevant. However, pipelining in the presence of updates requires additional care since the interleaved evaluation of steps from different operators may result in a later operator modifying data being accessed by an earlier operator. In the worst case, the semantics requires operators to materialize their input, making them behave as blocking operators. For example, a fully pipelined evaluation of the plan of Figure 2 would not be correct, since application of the Delete operator would interfere with the selection over the input customers.

Clearly, such blocking operators should be avoided when possible. We address this issue by controlling pipelining decisions during physical planning: each operator has both a version that materializes all its arguments (*eager*), and possibly additional versions that pipeline some of their arguments (*lazy*). The optimizer first produces a plan where all operators are eager, then uses rewrite rules to replace operators with lazy versions when it is safe to do so. In our example, the result is the plan in Figure 2, which only materializes on the input of the last Map operator, besides the materialization that is part of the hash-join algorithm.

We use three classes of side conditions for physical rewrite rules: *purity* and *commutativity*, as before, as well as a notion of *interleavability* of two plans, which means that the effect of their interleaved evaluation is the same as that of a sequential evaluation. We show how such conditions can be inferred using a variant of the path analysis used at the logical level for idempotency and commutativity inference.

Other approaches. Other semantics have been used for updates, such as the so-called *snapshot semantics* in the XQuery Update Facility [6], which delays update application until the end of the query. As was observed in [5], this approach is sufficient for typical database updates, but usually it is not expressive enough for programming purposes. Limitations on the locations where updates may occur have been suggested [5] to simplify compilation and optimization. However, such restrictions do not fully address the problem of optimization: as we will see at the end of Section 3, some typical query unnesting optimizations require commutativity analysis even in the presence of some of those restrictions.

Contributions. The paper makes the following technical contributions.

- We extend a previously proposed algebra for XQuery plus side effects [14,23] by providing the corresponding physical algebra.
- We study the side conditions needed to recover classical logical rewritings for join optimization and query unnesting in presence of side effects.
- We study pipelining in the context of a physical algebra with side effects, and provide similar side conditions that enable pipelining in query plans.
- We report on our implementation of the proposed algebra and side-effect analysis in an existing XQuery 1.0 compiler, and provide experimental results that validate the approach.

Organization. In Section 2 we introduce the algebra with side effects. In Section 3, we study logical rewritings in the presence of side effects. Section 4 presents changes necessary to the physical compiler. In Section 5, we present an experimental evaluation of the proposed approach. In Section 6, we review related work. We conclude in Section 7.

2. LANGUAGE AND ALGEBRA

In this section we introduce the language and algebra used in the rest of the paper. We use XQuery extended with simple update primitives (delete, insert, replace), and the algebra of [23], extended with similar update operators. Both the semantics for the language and algebra follow a left-to-right evaluation order with an eager evaluation of parameters. Also, both are *fully compositional*, *i.e.*, updates may occur anywhere.

2.1 Definition (XQuery with updates). We use E for expressions in the update language defined as follows:

```
1 E ::= ... XQuery Expressions...
2   | do insert E1 (as last | as first)? into E2
3   | do replace E1 with E2
4   | do delete E
```

The three added update primitives respectively insert a new node in the document at a given location, replace an existing node by a new node, and delete a set of nodes.

The semantics of the language is the same as that of XQueryP, *i.e.*, expressions are evaluated in strict left-to-right order, and updates are applied immediately. For instance, the following query declares a global variable *$log*, a function logIt which inserts its parameter into *$log* and returns it, and a simple FLWOR expression calling logIt.

```
1 declare variable $log := <log/>
2 declare updating function logIt($x) {
3   do insert <a>{$x}</a> as last into $log,
4   $x
5 };
6 let $a1 := logIt(30) return (logit($log/a+$a1))
```

After evaluation, the *$log* variable contains the following XML value, in which two new a elements have been inserted in order: <log><a>30<a>60</log>.

The semantics for such a languages can be described in terms of effects onto an XML *store* [14, 15, 17]. We use here the semantics of [15, 17], except in the case of FLWOR expressions, for which we use a semantics based on the notion of tuple stream: each clause produces a sequence of tuples which is fully evaluated before the next clause. Consider the following query over the *$log* variable resulting from the previous example.

```
1 for $x in $log//a
2 for $y in $log//a
3 where $x = $y
4 return logIt($x*2)
```

This query is an equi-join where the first two **for** clauses generate four [x;y] tuples, the **where** clause selects two of them, and the **return** clause adds 60 and 120 to the log. In the nested-for semantics of [15, 17], the evaluation of line 2 and line 4 is interleaved, while in the tuple-stream semantics the logIt function in line 4 is only evaluated after all the other clauses. This is important as it ensures there is no interference between the iterations of the first two **for** clauses. More details about why that semantic distinction is important for algebraic compilation can be found in [14].

We now define an algebra to compile our language. It is based on the algebra of [23] extended with the operators Insert, Replace, and Delete. The algebra is defined on a logical data model whose values are either XML values, *i.e.*, ordered sequences of items in the XQuery data model, or "ordered tables", *i.e.*, ordered sequences of tuples. A tuple is a record with fields containing XML values and written $[\mathsf{a}_1{:}\mathsf{v}_1;...;\mathsf{a}_n{:}\mathsf{v}_n]$, where $\mathsf{a}_1...\mathsf{a}_n$ are field names and $\mathsf{v}_1...\mathsf{v}_n$ are the associated values. [ ] denotes the empty tuple. Sequences of tuples are used to compile the tuple stream of a FLWOR.

2.2 Definition (XML algebra with updates). Algebraic *plans* p are terms constructed as follows:

$$
\begin{aligned}
p ::= {} & \mathsf{ID} \mid p_1 \circ p_2 \mid \mathsf{Sequence}(p_1, p_2) \mid \mathsf{Empty}() \\
& \mid \mathsf{Scalar}[v]() \mid \mathsf{Element}[q](p) \mid \mathsf{Text}(p) \\
& \mid \mathsf{TreeJoin}[s](p) \mid \mathsf{Call}[f](p_1, \ldots, p_n) \mid p_1 \text{ and } p_2 \mid p_1 \text{ or } p_2 \\
& \mid [a{:}p] \mid p_1 +\!\!+ p_2 \mid \#a \\
& \mid \mathsf{Select}\{p_1\}(p_2) \mid \mathsf{Product}(p_1, p_2) \\
& \mid \mathsf{Join}\{p_1\}(p_2, p_3) \mid \mathsf{LOuterJoin}[n]\{p_1\}(p_2, p_3) \\
& \mid \mathsf{Map}\{p_1\}(p_2) \mid \mathsf{OMap}[n](p_1) \mid \mathsf{MapIndex}[q](p_1) \\
& \mid \mathsf{MapConcat}\{p_1\}(p_2) \mid \mathsf{OMapConcat}[n]\{p_1\}(p_2) \\
& \mid \mathsf{GroupBy}[q_a][q_i][n]\{p_1\}\{p_2\}(p_3) \\
& \mid \mathsf{Insert}(p_1, p_2) \mid \mathsf{Delete}(p_1) \mid \mathsf{Replace}(p_1, p_2)
\end{aligned}
$$

Each plan receives an input value, either an XML item or a tuple, when executed. For the operator subplans we use $\{p\}$ to indicate a *dependent* subplan, that gets its input from the subplans that follow in the parameter list $(p_1, \ldots, p_n)$ of the *independent* subplans, that process the input of the parent. $[q]$ ($[n]$) is used for constant parameters (fields used to encode null values, respectively).

ID returns the current input tuple, $\circ$ passes the result of the right side plan as input to the left side (like function composition). Sequence, Empty, Scalar, Element, and Text, construct the corresponding XML values, TreeJoin extracts a sequence of nodes corresponding to an XPath step, Call invokes an XQuery function, and we have Boolean operators. Next are operations to construct a singleton tuple $[a : p]$, tuple concatenation $p_1 +\!\!+ p_2$, and field access $(\#a)$. Most of the other operators are ordered forms of a standard nested-relational algebra [20, 23] and should be familiar to the reader (Select, Product, Join, GroupBy, *etc.*). Map is the general functional map on sequences of tuples. MapConcat is a dependent product, equivalent to the D-Join operator of [20]. The OMap, OMapConcat are outer-variants for the Map and MapConcat operators. The MapIndex and MapIndexStep operators introduce a new field containing an index for each tuple. Finally, we include the update operators. We allow infix notation for binary operators (*e.g.*, =).

The main difference with [23] is that operators are not always pure, but can change the state of the XML store being processed. This means that the algebra semantics must specify an evaluation order.

2.3 Definition (evaluation order for XML algebra with updates). The algebra operators have the semantics of [23], with the following evaluation order for their operands. Independent subplans are evaluated first, fully, once, and in the order in which they appear in the parameter list. The dependent parameter, if present, is evaluated next, once for each item in the result of the independent parameters, with the following exception: Composition $p_1 \circ p_2$ evaluates the dependent parameter p_1 just once, on the whole result of p_2. Finally, the evaluation order of the two dependent parameters of GroupBy is specified in Section 3. For instance, by this definition, $\mathsf{Join}\{p_1\}(p_2, p_3)$ first evaluates p_2, then p_3, takes their product, and finally it evaluates p_1 once for each tuple in the product.

2.4 Example. Here is the query plan obtained by compiling the query in the introduction.

```
1 Map
2 {Sequence(
3   Delete(TreeJoin[descendant::membership](#doc)),#c)}
4 (Select
5   {TreeJoin[child::cid](#c) = TreeJoin[child::cid](#po)}
6   (MapConcat
7    {Map{[po:ID]}(TreeJoin[descendant::purchase-order](#doc))}
8    (Select
9      {TreeJoin[child::membership](#po) = 'gold'}
10     (Map{[c:ID]}(TreeJoin[descendant::customer](#doc))))))
```

This plan corresponds to Figure 1, and uses tuples to encode variable bindings (tuples with field c and po are constructed through maps on lines 10 and 7), providing a direct representation of the tuple stream semantics of the language. The corresponding compilation rules are provided in [14]. This representation into a NRA is suitable for join and query unnesting optimization. In this example the plan after join optimization, which corresponds to Figure 2, is as follows:

```
1 Map
2 {Sequence(
3   Delete(TreeJoin[descendant::membership](#doc)),#c)}
4 (Join
5   {TreeJoin[child::cid](#c) = TreeJoin[child::cid](#po)}
6   (Select
7     {TreeJoin[child::membership](#po) = 'gold'}
8     (Map{[c:ID]}(TreeJoin[descendant::customer](#doc))),
9    Map{[po:ID]}(TreeJoin[descendant::purchase-order](#doc))))
```

3. LOGICAL OPTIMIZATIONS

We present here our techniques to retain classical relational and nested-relational rewritings in the presence of side effects. The side-effect aware version of those rewritings is summarized in Figure 3. Unlike traditional relational equivalences, our rewritings are guarded by side conditions that depend on properties of the subplans related to the presence of side effects. In the rest of the section, we discuss these properties and how the applicability conditions can be identified for each classical rewrite rule. We discuss most of the rules in the text and give a full discussion of correctness for the (map through group-by) rule.

Commutativity. Many rewrite rules change the evaluation order of some subplans. For example, the rules that push one operator inside the other, such as (push select) and (push product) in Figure 3, exchange the evaluation order of the plan that is pushed and the one that is traversed. The same happens with rule (merge select), for a subtler reason: assume we have three tuples, and p_2 is true for the first two; then the left hand side evaluates the subplans in the order $p_2(t_1)$, $p_2(t_2)$, $p_2(t_3)$, $p_1(t_1)$, $p_1(t_2)$, while the right hand side alternates the two subplans, evaluating $p_2(t_1)$, $p_1(t_1)$, $p_2(t_2)$, $p_1(t_2)$, $p_2(t_3)$. In both cases (push rules and merge select), the rewriting is sound if we can exchange any evaluation of $p_1(v)$ and $p_2(v')$, as expressed by commutativity.

3.1 Definition (commutativity). p_1 and p_2 *commute*, denoted $\mathrm{comm}(p_1, p_2)$, iff, for any input values v and v', the

Nested-relational rewritings.

$$\mathsf{MapConcat}\{p_1\}(\mathsf{ID}) \to p_1 \qquad \text{(remove map)}$$

$$\mathsf{MapConcat}\{p_1\}(p_2) \to \mathsf{Product}(p_2, p_1) \quad \mathrm{indep}(p_1, p_2),\ \mathrm{idem}(p_1) \qquad \text{(insert product i)}$$

$$\mathsf{MapConcat}\{p_1\}(p_2) \to \mathsf{Product}(p_2, p_1) \quad \mathrm{single}(p_2),\ \mathrm{indep}(p_1, p_2) \qquad \text{(insert product s)}$$

$$\mathsf{Select}\{p_1\}(\mathsf{Product}(p_2, p_3)) \to \mathsf{Join}\{p_1\}(p_2, p_3) \qquad \text{(insert join)}$$

$$\mathsf{Select}\{p_1\}(\mathsf{MapConcat}\{p_2\}(p_3)) \to \mathsf{MapConcat}\{p_2\}(\mathsf{Select}\{p_1\}(p_3)) \quad \mathrm{idem}(p_1),\ \mathrm{pure}(p_2),\ \mathrm{comm}(p_1, p_2),\ \mathrm{indep}(p_1, p_2) \qquad \text{(push select i)}$$

$$\mathsf{Select}\{p_1\}(\mathsf{MapConcat}\{p_2\}(p_3)) \to \mathsf{MapConcat}\{p_2\}(\mathsf{Select}\{p_1\}(p_3)) \quad \mathrm{single}(p_2),\ \mathrm{pure}(p_2),\ \mathrm{comm}(p_1, p_2) \qquad \text{(push select s)}$$

$$\mathsf{Select}\{p_1\}(\mathsf{Select}\{p_2\}(p_3)) \to \mathsf{Select}\{p_2\ \mathit{and}\ p_1\}(p_3) \quad \mathrm{comm}(p_1, p_2) \qquad \text{(merge select)}$$

$$\mathsf{MapConcat}\{p_1\}(\mathsf{Product}(p_2, p_3)) \to \mathsf{Product}(p_2, \mathsf{MapConcat}\{p_1\}(p_3)) \quad \mathrm{indep}(p_1, p_2),\ \mathrm{idem}(p_1),\ \mathrm{comm}(p_1, p_2) \qquad \text{(push product i)}$$

$$\mathsf{MapConcat}\{p_1\}(\mathsf{Product}(p_2, p_3)) \to \mathsf{Product}(p_2, \mathsf{MapConcat}\{p_1\}(p_3)) \quad \mathrm{indep}(p_1, p_2),\ \mathrm{single}(p_2),\ \mathrm{comm}(p_1, p_2) \qquad \text{(push product s)}$$

Unnesting rewritings.

$$\mathsf{OMapConcat}[n]\{\mathsf{OMap}[n_1](p_1)\}(p_2) \to \mathsf{OMapConcat}[n_1]\{p_1\}(p_2) \qquad \text{(remove duplicate null)}$$

$$[x : p_1 \circ \mathsf{Map}\{p_2\}(p_3)] \to \mathsf{GroupBy}[x, [\,], [n]]\{p_1\}\{p_2\}(\mathsf{OMap}[n](p_3)) \qquad \text{(insert group-by)}$$

$$\mathsf{MapConcat}\{\mathsf{GroupBy}[x, inds, ns]\{p_1\}\{p_2\}(p_3)\}(p_4) \to \mathsf{GroupBy}[x, inds + ind_1, ns + n_1]\{p_1\}\{p_2\}(\mathsf{OMapConcat}[n_1]\{p_3\}(\mathsf{MapIndex}[ind_1](p_4))) \quad \mathrm{comm}(p_1, p_3),\ \mathrm{comm}(p_2, p_3) \qquad \text{(map through group-by)}$$

$$\mathsf{OMapConcat}[n]\{\mathsf{Join}\{p_1\}(\mathsf{ID}, p_2)\}(p_3) \to \mathsf{LOuterJoin}[n]\{p_1\}(p_3, p_2) \quad \mathrm{idem}(p_2),\ \mathrm{comm}(p_1, p_2) \qquad \text{(insert outer-join)}$$

Figure 3: Logical algebraic rewritings.

resulting values and the store effects of $p_1(v)$ and $p_2(v')$ are the same independently of their evaluation order.

Observe that the (merge select) rule is sound since our and operator evaluates the parameters in order, and only evaluates the second when the first is true. If we used a strict and operator, which always evaluates both operands, the right hand side would evaluate p_1 on more tuples than the left hand side, hence we would need a further side condition requiring p_1 to be *pure*, *i.e.*, free of side effects.

As observed in [16], commutativity is undecidable in general. To address that issue, we use the path analysis proposed in [16]: for every subplan p we collect the set of *accessed* paths $a(p)$ and the set of *updated* paths $u(p)$, which provide a static upper approximation of the nodes in the store that the plan reads and modifies. Two plans p_1 and p_2 commute if $u(p_1)$ is disjoint from $a(p_2) \cup u(p_2)$ and vice versa. Of course, this condition is always satisfied when the two plans are pure, since $u(p_i)$ is empty in that case. The main technical difference with [16] is that we add tuples to the path language. This extension is relatively straightforward but not included here for simplicity.

Idempotency and Purity. While the ability to modify evaluation order is important, the real efficiency boost comes from the ability to evaluate one expression once rather than many times, or to skip evaluation altogether. The simplest, and arguably most important, example is the (insert product i) rewriting in Figure 3. The rewritten version evaluates p_1 once, while the original version evaluates it as many times as there are input tuples in p_2. This is a key rewriting, as it also clears the way for the introduction of joins in the plan. As discussed in the introduction, this is safe only if p_1 is idempotent, according to the definition below.

Idempotency is also needed in the push...i rules, since they also reduce the number of times p_1 is evaluated, by a factor that is equal to the size of p_2. When p_2 is a singleton, however, there is no difference in the number of times p_1 is evaluated, hence there is no need to check idempotency in the (push...s) and (insert product s) rules.

3.2 Definition (idempotency). p is said to be idempotent, denoted idem(p), iff, for any input value v, two evaluations of $p(v)$ result in the same effects and value as one evaluation of $p(v)$.

While idempotency is undecidable in general, we approximate it using path analysis: a plan p is idempotent if $a(p)$ is disjoint from $u(p)$, so that its second evaluation will produce the same results as the first, and p performs no insert operation. Again, every pure plan satisfies this test.

It is important to note that checking for idempotency only works if the cardinality of evaluation is reduced but the corresponding plan is evaluated *at least once*. Considering the (push select) rules, we note that, on the right hand side, p_2 is not applied on the tuples that do not satisfy p_1, while it is applied on every tuple on the left hand side. For that reason, we need to check for *purity* of p_2; purity inference is trivial.

Standard properties. In addition to properties related to side effects, we also use a couple of more standard properties, namely independence (whether a plan depends on the tuple fields returned or added by a particular plan), and cardinality (notably whether a plan returns a singleton tuple). They are defined as follows.

3.3 Definition (independence). p_1 is independent of p_2, denoted indep(p_1, p_2), iff p_1 does not *access* the fields *created* by p_2.

3.4 Definition (singleton). p is said to be a singleton plan, denoted single(p), iff p always returns a single tuple.

Note that both independence and singleton properties can be checked through static typing at the algebraic plan level.

Query unnesting.

3.5 Scenario (Implicit grouping). In the rest of the section, we focus on how the above framework applies to a typical query unnesting scenario, and we discuss the (map

through group-by) rule. Let us consider the following simple class of nested queries.

```
1 for $x in E1
2 let $a :=
3   F2(for $y in E3
4        where E4
5        return E5)
6 return E6
```

where, E_1, E_3, E_5, E_6 are arbitrary XQuery with updates, F_2 is an aggregate function, and E_4 is a join predicate between *$x* and *$y*. Specifically, this nesting form corresponds to the *implicit grouping pattern* from [20].

After function inlining and some simple syntactic rewrites, the first example in the introduction is an instance of this nesting form. For exposition purposes, we use a version of the query with additional side effects, in which the system maintains logs containing the number of unconfirmed orders per customer (using logcount) and a backup copy of those individual orders (using logord):

```
1 for $c in getCustomers()
2 let $u := logcount(
3   for $po in $doc//purchase-order
4   where $po/cid = $c/cid and $po/status = 'unconfirmed'
5   return logord($po)
6 )
7 return <new-orders>{$u}</new-orders>
```

In this example, E_1 and E_3 correspond to accessing customers and orders, respectively. F_2 is the logcount function, E_4 is the conjunctive join predicate, and E_5, E_6 are, respectively, the inner and outer return clauses. Our goal is to be able to unnest the query using traditional rewritings [9, 20, 21, 23]. Indeed, for this query, we can reuse the same sequence of rewriting rules as used in [23]. The query is first compiled into the following plan.

```
1 Map{Element[new-orders](#u)}
2  (MapConcat
3    {[u:Call[logcount](
4       Map{Call[logord](#po)}
5        (Select{TreeJoin[ch.::cid](#c)=TreeJoin[ch.::cid](#po)
6                and TreeJoin[ch.::status](#po) = 'unconfirmed'}
7          (MapConcat
8            {Map{[po:ID]}
9                 (TreeJoin[desc.::purchase-order](#doc))}
10           (ID))))
11   ]}
12   (Map{[c:ID]}(Call[getCustomers]())))
```

The [u:Call[logcount](..)] subplan (lines 4–10) corresponds to the **let *$u*** clause and nested FLWOR in the query, and the MapConcat operator evaluates it once for each customer. The plan is unnested into the plan below. The unnested plan, instead of using a nested loop, performs an outer join followed by a single GroupBy that evaluates logord for each unconfirmed order, and logcount for each group of unconfirmed orders (details of the semantics of GroupBy are given below).

```
1 Map{Element[new-orders](#u)}
2  (GroupBy[u][i][n]
3    {Call[logcount](ID)}
4    {Call[logord](#po)}
5    (LOuterJoin[n]
6     {TreeJoin[child::cid](#c) = TreeJoin[child::cid](#po)
7       and TreeJoin[child::status](#po) = 'unconfirmed'}
8     (MapIndex[i](Map{[c:ID]}(Call[getCustomers]())),
9      Map{[po:ID]}(TreeJoin[desc.::purchase-order](#doc)))))
```

The main sequence of rewritings is: (insert group-by) introduces an empty GroupBy instead of tuple construction on line 3; (map through group-by) pushes the MapConcat through the GroupBy resulting in the proper grouping over i, which is an index used to keep track of sequence order; the inner MapConcat operator on line 7 is turned into a trivial Cartesian product, which combined with the Select yields a Join; finally, after simplifying the combination of OMapConcat and OMap into an OMapConcat, (insert outer-join) is used to introduce the final LOuterJoin.

Most of the side conditions for those rewritings are based on the same principles that we explained earlier in the section. The (map through group-by) rewrite, though, is more complex, and worth explaining (we discuss here the side conditions only, since the rule is not new [23]). We first define the semantics of GroupBy$[a][k_1...k_m][n]\{p_1\}\{p_2\}(p_3)$, and, specifically, the order in which the dependent subplans p_2 and p_1 are evaluated. The parameter $k_1...k_m$ is the set of grouping fields, a is the field name for the new result of the aggregation function, and n is the null field, that we ignore here, since it adds nothing to the discussion of side effects and conditions. The semantics is specified as follows.

1. First p_3 is evaluated, resulting in a table T, and the sequence of grouping keys is computed; this set is the projection of T on $k_1...k_m$.

2. Then for each key tuple kt: (a) p_2 is computed for each tuple $t_1^{kt}...t_n^{kt}$ of T that coincides with kt over $k_1...k_m$, (b) the aggregate value av is computed by applying p_1 to the set $\{p_2(t_1^{kt}), \ldots, p_2(t_n^{kt})\}$, and the tuple $[a : av] +\!\!+ kt$ is emitted; hence, the evaluations of p_2 and p_1 are alternated.

The right side of the (map through group-by) rule enriches each tuple of p_4 with an index field ind_1, computes the (outer) dependent product of p_3 and $p_4 + ind_1$, and then applies a GroupBy; the difference between OMapConcat and MapConcat is not relevant to our discussion here.

We first observe that on both sides of the rewriting rule, p_4 is evaluated before any other plan so we don't need any condition involving p_4.

A second simple observation is that after rewriting, all evaluations of p_3 occur before those of p_1 and p_2, while its evaluations were originally alternating with those of p_1 and p_2. Hence p_3 must commute with both p_1 and p_2.

The remaining question is whether p_1 and p_2 need to commute or be idempotent. The answer follows from the specific semantic details of GroupBy: the sequence of p_1 and p_2 evaluations is exactly the same in the two cases. Let us illustrate this point on our example. The following shows the plan just before the (map through group-by) rewrite is applied.

```
1 Map{Element[new-orders](#u)}
2  (MapConcat
3   {GroupBy[u][][n]
4    {Call[logcount](ID)}
5    {Call[logord](#po)}
6    (OMap[n]
7     (Select{TreeJoin[child::cid](#c)=TreeJoin[child::cid](#po)
8           and TreeJoin[child::status](#po) = 'unconfirmed'}
9      (MapConcat
10       {Map{[po:ID]}
11          (TreeJoin[descendant::purchase-order](#doc))}
12       (ID)))))}
13  (Map{[c:ID]}(Call[getCustomers]())))
```

Both p_1 (logcount) and p_2 (logord) have side effects. We use $a_1, ..., a_n$ to denote updates performed by the logcount function, and $b_1, ..., b_n$ to denote updates performed by the logord function.

When this plan is processed, first p_4 is fully evaluated, and then, for each of the bindings of c, the GroupBy is evaluated and the updates applied. Let's assume the group-by results in three groups, the first and last with two matching tuples, and the second one with one matching tuple. The following table contains one subtable for each tuple generated by p_4. Each subtable contains: (i) the table (with fields c, po, and n) produced by p_3, (ii) The side effects produced by p_2 applied to each element in the output of p_3 (indicated as b_i), and then by p_1 applied to the whole output of p_2 (indicated as a_i), (iii) the final value produced by p_1.

c	po	n	effects	u
c_1	po_1	f	b_1, b_2, a_1	2
c_1	po_2	f		
c_2	po_3	f	b_3, a_2	1
c_3	po_4	f	b_4, b_5, a_3	2
c_3	po_5	f		

Thus the final sequence of effects is (b_1, b_2, a_1, b_3, a_2, b_4, b_5, a_3). Applying the rewriting, which pushes the MapConcat through the GroupBy, leads to the following plan:

```
1 Map{Element[new-orders](#u)}
2  (GroupBy[u][i][n+n1]
3   {Call[logcount](ID)}
4   {Call[logord](#po)}
5   (OMapConcat[n1]
6    {OMap[n]
7     (Select{TreeJoin[child::cid](#c)= TreeJoin[child::cid](#po)
8           and TreeJoin[child::status](#po) = 'unconfirmed'}
9      (MapConcat
10       {Map{[po:ID]}
11         (TreeJoin[descendant::purchase-order](#doc))}
12       (ID))))}
13   (MapIndex[i](Map{[c:ID]}(Call[getCustomers]()))))
```

inside which the OMapConcat will be merged with the OMap and then rewritten into a LOuterJoin. Note that the effects for p_3 are all applied before the GroupBy is evaluated. (The GroupBy operator groups on the i field, and, for each group, applies p_2, followed by p_3.) This results in the following table and effects.

c	i	u	effects
c_1	1	2	b_1, b_2, a_1
c_2	2	1	b_3, a_2
c_3	3	2	b_4, b_5, a_3

The final sequence of effects is (b_1, b_2, a_1, b_3, a_2, b_4, b_5, a_3). The relative order of effects between p_1 and p_2 is unchanged.

Hence, in order to apply (map through group-by) to our query, we have only to verify that p_1 and p_2 commute with p_3. With reference to the general query, we can easily verify that p_1 corresponds to F_2, p_2 to E_5, and p_3 to E_3 restricted by E_4, hence this rewrite rule is applied when F_2 and E_5 do not interfere with E_3 and E_4. No other rule applied during the unnesting process has side effect related conditions, apart from the crucial one, (insert outer-join). Through a similar analysis, we can verify that it requires idempotency of E_3 and commutativity of E_3 and E_4.

Discussion. Applying the same sequence of rewritings to the nested form in Scenario 3.5, we can see that the complete set of conditions which must be checked is as follows.

- idempotency: idem(E_3)
- commutativity: comm(F_2, E_4), comm(F_2, E_3), comm(E_5, E_4), comm(E_5, E_3), comm(E_3, E_4).

Based on those conditions, we can make the following general remarks.

- If commutativity-based side conditions were substituted with purity requests, the presence of a side effect in any position apart from E_1 and E_6 would prevent unnesting; on the other side, if one is ready to do commutativity (and idempotency) analysis, then optimization would be compatible with the presence of side effects in any position, provided that they do not interfere with the rest of the query; hence, commutativity is buying us a lot.
- We expect that updating and accessing the same data in the internal parts of a query will be quite rare in practice, so that the commutativity side conditions should be satisfied in the vast majority of situations.
- Our rules only impose idempotency for E_3, and this seems minimal, because some of the essence of join optimization is to evaluate E_3 only once.

Finally, we come back to the issue of language restrictions. Remember that the XQueryP [5] proposal only allows updates in the `return` clause of FLWOR expressions. The typical nesting pattern we studied in this section can be written solely with side effects in return clauses. Here is the example we used, written in that form.

```
1 for $c in getCustomers()
2 return
3  <new-orders>{
4   logcount(
5    for $po in //purchase-order
6    where $po/cid = $c/cid
7    return logord($po)
8   )
9  }</new-orders>
```

As before, producing the proper unnested plan in this example requires checking commutativity between the log functions and the subquery that accesses the purchase order. Hence, the XQueryP restrictions, although quite strong, are not sufficient to avoid the need for techniques such as commutativity or idempotency analysis.

4. PHYSICAL PLANNING

In this section, we look at the impact of the presence of side effects on code selection (picking up physical algorithms), and on pipelining. As we have seen in the introduction, pipelining can interact heavily with side effects. Other aspects related to cost-based optimization are beyond the scope of our work at this point. We first give an overview of the proposed approach.

Overview. We represent physical operators as logical operators annotated with two additional parameters, as in $\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SEE}}$ (NL means nested loop) or $\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SLE}}$; the superscript indicates the algorithm being used, while the subscript indicates positionally which arguments are evaluated eagerly

(E), which are evaluated lazily (L), and which return a singleton (S), like in the case of the Join predicate. The algorithm parameter is omitted when we only have one implementation, as in $\mathsf{Select}_{\mathsf{SE}}$.

To produce a physical plan, our compiler first generates a plan that only uses default algorithms and eager operators, such as $\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SEE}}$, since this combination is guaranteed to respect the semantics in the presence of side effects. Then, a set of rewrite rules is applied which allows better algorithms to be selected. Finally, another set of rewrite rules is applied which allows eager arguments to be substituted with lazy arguments, hence reducing the amount of data to be materialized. As for logical rewritings, rules in both sets are based on side conditions that are related to the presence of side effects. Note that these rules only change aspects of the evaluation for each operator and never modify the shape of the query plan, The full set of rewritings is given in Figure 4. Before we explain the side conditions in details, let us first illustrate how they are used.

Consider the optimized plan in Example 2.4 (Figure 2). The compiler first produces the following plan. In that plan, every operator has a default implementation and every input is eagerly evaluated (*i.e.*, materialized), as indicated by the corresponding annotations.

```
1 Map_EE
2 {Sequence_EE(Delete(TreeJoin[desc.::membership](#doc)),#c)}
3 (Join^NL_SEE
4   {TreeJoin[child::cid](#c) = TreeJoin[child::cid](#po)}
5   (Select_SE
6     {TreeJoin[child::membership](#po) = 'gold'}
7     (Map_EE{[c:ID]}(TreeJoin[desc.::customer](#doc))),
8    Map_EE{[po:ID]}(TreeJoin[desc.::purchase-order](#doc))))
```

$\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SEE}}$ can be rewritten as a hash join, since its first argument is pure and contains an equality, as follows.

```
1 Map_EE
2 {Sequence_EE(Delete(TreeJoin[desc.::membership](#doc)),#c)}
3 (Join^Hash_SEE
4   {TreeJoin[child::cid](#c) = TreeJoin[child::cid](#po)}
5   (Select_SE
6     {TreeJoin[child::membership](#po) = 'gold'}
7     (Map_EE{[c:ID]}(TreeJoin[desc.::customer](#doc))),
8    Map_EE{[po:ID]}(TreeJoin[desc.::purchase-order](#doc))))
```

Now, one specific challenge in the context of such physical plans is that rewritings in various parts of the plan may not be independent. For instance, let us assume we first apply a rewrite rule that introduces laziness on the final Map. This rewriting is sound since the branch of the join that accesses customers' membership status is evaluated eagerly, and the final delete operator does not modify any of the data used by the join predicate. This results in the following optimized plan.

```
1 Map_LL
2 {Sequence_EE(Delete(TreeJoin[desc.::membership](#doc)),#c)}
3 (Join^Hash_SEE
4   {TreeJoin[child::cid](#c) = TreeJoin[child::cid](#po)}
5   (Select_SE
6     {TreeJoin[child::membership](#po) = 'gold'}
7     (Map_EE{[c:ID]}(TreeJoin[desc.::customer](#doc))),
8    Map_EE{[po:ID]}(TreeJoin[desc.::purchase-order](#doc))))
```

After this rewrite, however, we can no longer pipeline the left branch of the join that accesses customers, since the evaluation of the Join input and the outer Map would be interleaved, resulting in membership information being deleted while still processing customer information. In the absence of a cost model, we solve this problem by applying the rules for pipelining introduction in a bottom-up fashion, *i.e.*, rules are always applied in the context of an eager argument or at the root of the tree. This strategy is simple and seems to be a reasonable starting point. With this bottom-up application approach, the two inner Map operators and Select operators are rewritten as lazy since their input plans are pure. The $\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SEE}}$ is rewritten to pipeline its left branch. The rule to turn $\mathsf{Sequence}_{\mathsf{EE}}$ into its lazy form has no side conditions, hence it is applied. Moving to the root, the outer $\mathsf{Map}_{\mathsf{EE}}$ to $\mathsf{Map}_{\mathsf{LE}}$ comes for free, and we get the following result where only the right branch of the join, and the input of the outermost map are materialized as in Figure 2.

```
1 Map_LE
2 {Sequence_LL(Delete(TreeJoin[desc.::membership](#doc)),#c)}
3 (Join^Hash_SLE
4   {TreeJoin[child::cid](#c) = TreeJoin[child::cid](#po)}
5   (Select_SL
6     {TreeJoin[child::membership](#po) = 'gold'}
7     (Map_LL{[c:ID]}(TreeJoin[desc.::customer](#doc))),
8     Map_LL{[po:ID]}(TreeJoin[desc.::purchase-order](#doc))))
```

Rewrite rules. Not all the operators are subject to eager/lazy optimization. Specifically, we do not need to consider the operators whose output is a singleton, (*e.g.*, Empty, Scalar, Element, TreeJoin). Figure 4 shows the necessary rules for the remaining operators.

To discuss those rules, we need a notation to define the semantics of physical operators. To this aim, we use the yield syntax made popular by C# 2.0. In our notation, P $x_1 \ldots x_n$:= new iterator{*block*} builds an object P with two methods, MoveNext and Current. P.Current returns the value of an object field named Current, which is initially null, and which is updated by P.MoveNext. P.MoveNext evaluates *block* until it finds a "yield *value*" statement, then stores *value* into P.Current, saves the execution state of the block, and returns true. The next time P.MoveNext is called, it starts from the execution state where it stopped the previous time; when it finds no more yield *value* to execute, it returns false, and sets P.Current to null. For example, consider the following iterator:

```
1 PList := new iterator { foreach (x in (1,2)) {yield x}; yield 3 }
```

PList.MoveNext returns true three times, setting Current to 1, 2, and 3, and returns false when called for the fourth time onwards. PList.Current evaluates to 1, 2 and 3 after the first three calls to PList.MoveNext, and returns null before the first call and after the fourth. Using such syntax, we can easily write the semantics for the eager and lazy versions of our operators.

We start with the Cartesian product $\mathsf{Product}_{\mathsf{XX}}$ operators. The foreach (t in Expr){*block*} statement builds a new iterator I = Expr, and evaluates *block* once for each successful invocation of I.MoveNext, binding t to I.Current. Every plan has a parameter x, used to transmit context information, for example to a subplan p_1 that depends on the output of another subplan p_2, as in Join{XXX}{p_1}(p_2,p_3). Most operators just pass x to their subplans, while the field lookup (#a) and navigation (TreeJoin[*path*]) operators actually use it. (In a typical implementation, the parameter is passed to a plan P using a P.Open(x) operation, but this is irrelevant here.)

General applicability condition: the lhs must be an eager argument of its parent, or must be the root of the tree

$\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SEE}}\{s_1\}(p_2,p_3)$	$\rightarrow$	$\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SEE}}\{s_1\}(p_2,p_3)$	$\mathsf{IsEquiJoin}(s_1)$, and $\mathrm{pure}(s_1)$
$\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SEE}}\{s_1\}(p_2,p_3)$	$\rightarrow$	$\mathsf{Join}^{\mathsf{Sort}}_{\mathsf{SEE}}\{s_1\}(p_2,p_3)$	s_1 order relation, and $\mathrm{pure}(s_1)$
$\mathsf{LOuterJoin}^{\mathsf{X}}_{\mathsf{SEE}}[\mathsf{q}]\{s_1\}(p_2,p_3)$	$\rightarrow$	Same rules as $\mathsf{Join}^{\mathsf{X}}_{\mathsf{SEE}}$	
$\mathsf{Sequence}_{\mathsf{EE}}(p_1,p_2)$	$\rightarrow$	$\mathsf{Sequence}_{\mathsf{LL}}(p_1,p_2)$	
$\mathsf{Selects}_{\mathsf{SE}}\{s_1\}(p_2)$	$\rightarrow$	$\mathsf{Selects}_{\mathsf{SL}}\{s_1\}(p_2)$	$\mathrm{intrlv}(p_2,s_1)$
$\mathsf{Product}_{\mathsf{EE}}(p_1,p_2)$	$\rightarrow$	$\mathsf{Product}_{\mathsf{LE}}(p_1,p_2)$	$\mathrm{comm}(p_1,p_2)$
$\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SEE}}\{s_1\}(p_2,p_3)$	$\rightarrow$	$\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SLE}}\{s_1\}(p_2,p_3)$	$\mathrm{comm}(p_3,p_2)$, and $\mathrm{intrlv}(p_2,s_1)$
$\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SEE}}\{s_1\}(p_2,p_3)$	$\rightarrow$	$\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SLE}}\{s_1\}(p_2,p_3)$	$\mathrm{comm}((p_3,s_1),p_2)$, and $\mathrm{intrlv}(p_3,s_1)$
$\mathsf{Join}^{\mathsf{Sort}}_{\mathsf{SEE}}\{s_1\}(p_2,p_3)$	$\rightarrow$	$\mathsf{Join}^{\mathsf{Sort}}_{\mathsf{SLE}}\{s_1\}(p_2,p_3)$	$\mathrm{comm}((p_3,s_1),p_2)$
$\mathsf{LOuterJoin}^{\mathsf{X}}_{\mathsf{Y}}[q]\{s_1\}(p_2,p_3)$	$\rightarrow$	Same rules as $\mathsf{Join}^{\mathsf{X}}_{\mathsf{Y}}$	
$\mathsf{Map}_{\mathsf{EE}}\{p_1\}(p_2)$	$\rightarrow$	$\mathsf{Map}_{\mathsf{LE}}\{p_1\}(p_2)$	
$\mathsf{Map}_{\mathsf{LE}}\{p_1\}(p_2)$	$\rightarrow$	$\mathsf{Map}_{\mathsf{LL}}\{p_1\}(p_2)$	$\mathrm{intrlv}(p_2,p_1)$
$\mathsf{MapConcat}_{\mathsf{EE}}\{p_1\}(p_2)$	$\rightarrow$	$\mathsf{MapConcat}_{\mathsf{LE}}\{p_1\}(p_2)$	
$\mathsf{MapConcat}_{\mathsf{LE}}\{p_1\}(p_2)$	$\rightarrow$	$\mathsf{MapCOncat}_{\mathsf{LL}}\{p_1\}(p_2)$	$\mathrm{intrlv}(p_2,p_1)$
$\mathsf{OMapConcat}_{\mathsf{EE}}[q]\{p_1\}(p_2)$	$\rightarrow$	$\mathsf{OMapConcat}_{\mathsf{LE}}[q]\{p_1\}(p_2)$	
$\mathsf{OMapConcat}_{\mathsf{LE}}[q]\{p_1\}(p_2)$	$\rightarrow$	$\mathsf{OMapCOncat}_{\mathsf{LL}}[q]\{p_1\}(p_2)$	$\mathrm{intrlv}(p_2,p_1)$
$\mathsf{OMap}_{\mathsf{E}}[q](p_1)$	$\rightarrow$	$\mathsf{OMap}_{\mathsf{L}}[q](p_1)$	
$\mathsf{MapIndex}_{\mathsf{E}}[q](p_1)$	$\rightarrow$	$\mathsf{MapIndex}_{\mathsf{L}}[q](p_1)$	
$\mathsf{GroupBy}_{\mathsf{SEE}}[q_a][q_i][q_n]\{p_1\}\{p_2\}(p_3)$	$\rightarrow$	$\mathsf{GroupBy}_{\mathsf{SLE}}[q_a][q_i][q_n]\{p_1\}\{p_2\}(p_3)$	$\mathrm{intrlv}(p_2,p_1)$

Figure 4: Physical algebraic rewritings.

```
1 Product_EE(p1,p2) x :=
2  new iterator
3  {Mat1 := foreach (t1 in p1(x)) { t1 };
4   Mat2 := foreach (t2 in p2(x)) { t2 };
5   foreach (t1 in Mat1) {
6    foreach (t2 in Mat2) {
7     yield (t1++t2) }}}
```

```
1 Product_LE(p1,p2) x :=
2  new iterator
3  {Mat2 := foreach (t2 in p2(x)) { t2 };
4   foreach (t1 in p1(x)) {
5    foreach (t2 in Mat2) {
6     yield (t1++t2) }}}
```

Both operators support the MoveNext-Current interface, but $\mathsf{Product}_{\mathsf{EE}}$ implements the eager semantics by materializing its input. In $\mathsf{Product}_{\mathsf{EE}}$, the first application of MoveNext (hereafter, MN) builds both Mat_1 and Mat_2, and returns the control to the caller when the first output tuple is built (using tuple concatenation $\mathsf{t}_1{+\!+}\,\mathsf{t}_2$); the next call will start from that point. In $\mathsf{Product}_{\mathsf{LE}}$, the first application of MN builds Mat_2 and calls $\mathsf{p}_1(\mathsf{x})$.MN just once; the next call to $\mathsf{p}_1(\mathsf{x})$.MN, implicit in the foreach (t_1 in $\mathsf{p}_1(\mathsf{x})$) notation, will be caused by the next call to ($\mathsf{Product}_{\mathsf{EE}}(\mathsf{p}_1,\mathsf{p}_2)$ x).MN. In other words, $\mathsf{Product}_{\mathsf{LE}}$ pipelines on its left input, as in a traditional side effect free relational implementation.

We need to define one more notion before talking about correctness of the rewritings. We define a *consecutive* evaluation of a plan P to be a sequence of calls to P.MoveNext, where the last call returns false and no other store action is interleaved between two calls (*consecutive*). We define the *total result* of such an evaluation of a plan P as the pair of the ordered sequence of all the values (the *value result*) and the store effect (the *store result*) that it produces.

Looking at the rule for product on Figure 4, the total result of $\mathsf{Product}_{\mathsf{EE}}(\mathsf{p}_1,\mathsf{p}_2)$ x is obtained through a sequence of calls $\mathsf{p}_1(\mathsf{x})$.MN,..., $\mathsf{p}_1(\mathsf{x})$.MN, followed by $\mathsf{p}_2(\mathsf{x})$.MN,..., $\mathsf{p}_2(\mathsf{x})$.MN, while the $\mathsf{Product}_{\mathsf{LE}}$ version uses the inverse sequence $\mathsf{p}_2(\mathsf{x})$.MN,..., $\mathsf{p}_2(\mathsf{x})$.MN, $\mathsf{p}_1(\mathsf{x})$.MN,..., $\mathsf{p}_1(\mathsf{x})$.MN.

From this observation we deduce the soundness of our first rewrite rule, where commutativity $\mathrm{comm}(\mathsf{p}_1,\mathsf{p}_2)$ is formally defined below.

> If (a) $\mathsf{Product}_{\mathsf{EE}}(p_1,p_2)$ is an eagerly evaluated subplan in the context $P[\,]$ and (b) $\mathrm{comm}(p_1,p_2)$, then $P[\mathsf{Product}_{\mathsf{EE}}(p_1,p_2)] \rightarrow P[\mathsf{Product}_{\mathsf{LE}}(p_1,p_2)]$

To understand (a), consider that commutativity of p_1 and p_2 ensures that the *total results* of $\mathsf{Product}_{\mathsf{EE}}(p_1,p_2)$ and $\mathsf{Product}_{\mathsf{LE}}(p_1,p_2)$ coincide, but still the *partial* effects of invoking, for example, the first MoveNext is quite different in the two cases: the EE operator will fully evaluate its subplans, while the LE operator will just invoke the second one. When $\mathsf{Product}_{\mathsf{EE}}(p_1,p_2)$ is an E operand for its parent, we only care about the total result, hence the rewriting is sound.

Let us now look at the join operator. Here is a typical description of the nested-loop join using our iterator syntax.

```
1 Join^NL_SEE{s1}(p2,p3) x :=
2  new iterator
3  {Mat2 := foreach (t2 in p2(x)) { t2 };
4   Mat3 := foreach (t3 in p3(x)) { t3 };
5   foreach (t2 in Mat2) {
6    foreach (t3 in Mat3) {
7     if (s1(t2++t3)) yield (t2++t3) }}}
```

As usual, hash join requires s_1 to contain an equality condition. As was observed in [23], the situation is slightly more complex with XQuery because of the existential semantics of equality and because equality depends on types. This has little to do with side effects, hence we adopt here a basic approach: our hash join algorithm assumes that s_1 has the "equijoin shape" $\mathsf{s}_{11} \wedge (\mathsf{s}_{12} = \mathsf{s}_{13})$, where s_{12} and s_{13} only depend on tuple fields produced, respectively, by p_2 and p_3 (as in the plan in Figure 2). We use a predicate $\mathsf{IsEquiJoin}(\mathsf{s}_1)$ to check this condition and a function $\mathsf{DecomposeEquiJoin}(\mathsf{s}_1)$ to decompose s_1 (in Figure 4). Here is the corresponding implementation. Note that the creation of the hashtable (lines 5 to 6) amounts to materialization.

```
1 Join^Hash_SEE{s1}(p2,p3) x :=
2  new iterator
3  {Mat2 := foreach (t2 in p2(x)) return t2;
4   HashTab := new HashTable {};
5   (s11,s12,s13) = DecomposeEquiJoin(s1);
6   foreach (t3 in p3(x)) { HashTab.insert(s13(t3),t3) };
7   foreach t2 in Mat2 {
8    foreach h3 in HashTab.lookup(s12(t2))
9     if (s11(t2++t3)) yield (t2++t3) }}
10 }
```

The evaluation of $\mathsf{Join}^{\mathsf{NL}}_{\mathsf{SEE}}$ results in the following sequence of MoveNext calls: $\mathsf{p}_2, \ldots, \mathsf{p}_2, \mathsf{p}_3, \ldots, \mathsf{p}_3, \mathsf{s}_1, \ldots, \mathsf{s}_1$. The call sequence for $\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SEE}}$ may be represented as follows: $\mathsf{p}_2, \ldots, \mathsf{p}_2, \mathsf{p}_3, \ldots, \mathsf{p}_3, \mathsf{s}_{13}, \ldots, \mathsf{s}_{13}, \mathsf{s}_{12}, \mathsf{s}_{11}, \ldots, \mathsf{s}_{11}, \mathsf{s}_{12}, \mathsf{s}_{11}, \ldots, \mathsf{s}_{11}$. The only difference is that s_1 evaluation is decomposed, and s_1 is not applied to every tuple pair. We hence require s_1 to be pure, and have the first rule in Figure 4.

Finally, the lazy version of the hash join ($\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SLE}}$) only differs in that it omits the materialization of the left branch (line 3). In this case, the call sequence is $\mathsf{p}_3, \mathsf{s}_{13}, \ldots, \mathsf{p}_3, \mathsf{s}_{13}, (\mathsf{p}_2, \mathsf{s}_{12}, \mathsf{s}_{11}, \ldots, \mathsf{s}_{11}, \ldots, \mathsf{p}_2, \mathsf{s}_{12}, \mathsf{s}_{11}, \ldots, \mathsf{s}_{11})$. To rewrite $\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SEE}}$ into $\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SLE}}$ we first assume that p_2 and p_3 commute, hence the trace of the first is equivalent to $\mathsf{p}_3, \ldots, \mathsf{p}_3, \mathsf{p}_2, \ldots, \mathsf{p}_2, \mathsf{s}_{13}, \ldots, \mathsf{s}_{13}, \mathsf{s}_{12}, \mathsf{s}_{11}, \ldots, \mathsf{s}_{11}, \mathsf{s}_{12}, \mathsf{s}_{11}, \ldots, \mathsf{s}_{11}$. Equivalence follows if we further assume that s_{12} and s_{11} calls can be commuted with p_2, and that s_{13} calls can be commuted with both p_3 and p_2. This sounds complex, but amounts to inferring that all of p_2, p_3 and s_1 are mutually read-write disjoint.

Interleavability. Before presenting the interleavability side condition, it is better to introduce a couple of definitions that will allow us to be more precise about commutativity as well. We first define an *interleaved evaluation* of p_1 and p_2 as a sequence of calls to p_1.MN interleaved with calls to p_2.MN (and nothing else), where the last calls p_1.MN and p_2.MN both return false. The consecutive evaluation of one plan followed by the consecutive evaluation of the other is hence a special case of interleaved evaluation.

4.1 Definition (Commutativity). $\mathrm{comm}(p_1, p_2)$ iff, for any interleaved evaluation of p_1 and p_2 that start from a given store: (a) the store result is the same; (b) the value result of p_1 is the same; (c) the value result of p_2 is the same.

This notion of commutativity is quite stronger than the logical one, since we may actually interleave steps of one operator with steps of the other. Most of our rewrite rules would be consistent if we adopted a weaker notion of commutativity, where we only require that the consecutive evaluation of p_2 after p_1 has the same result as the consecutive evaluation of p_1 after p_2. However, we statically infer $\mathrm{comm}(p_1, p_2)$ from read-write disjointness of p_1 and p_2, and both versions of commutativity follow from such disjointness, hence we just consider the strong definition above. Besides purity and commutativity, we adopt a third *interleavability* side condition, which is a weaker, ordered, form of commutativity, defined as follows.

4.2 Definition (Interleavability). p_2 is interleavable after p_1, written $\mathrm{intrlv}(p_1, p_2)$ iff, for any interleaved evaluation of p_1 and p_2, that start from a given store, where the first call is to p_1.MoveNext: (a) the store result is the same; (b) the value result of p_1 is the same; (c) the value result of p_2 is the same.

While this sounds esoteric, $\mathrm{intrlv}(p_1, p_2)$ is a common condition, and is significantly weaker than $\mathrm{comm}(p_1, p_2)$. For example, it holds whenever p_1 is an eager plan, *i.e.*, if it only accesses the shared store during its first step. Static approximation of $\mathrm{intrlv}(p_1, p_2)$ is performed by extending path analysis so that it analyzes, for each physical path, what is accessed and updated by the whole plan, $a(p)$ and $u(p)$, and what is accessed an updated from the second step onwards $a_2(p)$ and $u_2(p)$. $\mathrm{intrlv}(p_1, p_2)$ is deduced when $a(p_1)$ and $u(p_1)$ do not interfere with $a_2(p_2)$ and $u_2(p_2)$. Both a_2 and u_2 are empty for eager plans, even if they are impure, which makes this condition widely applicable.

We now observe that to rewrite $\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SEE}}$ into $\mathsf{Join}^{\mathsf{Hash}}_{\mathsf{SLE}}$, we do not need $\mathrm{comm}(p_3, s_1)$, but $\mathrm{intrlv}(p_3, s_1)$ is enough, since both plans start executing p_3 before s_1. However, since the first steps of p_2 and s_1 are exchanged, $\mathrm{comm}(p_2, s_1)$ cannot be relaxed to $\mathrm{intrlv}(p_2, s_1)$.

We have proceeded in this way to specify the semantics of all our operators, and deduced the side conditions for the rewrite rules that we exploit to do code selection and to introduce pipelining; these are the rules in Figure 4. (Remark: $\mathsf{Join}^{\mathsf{Sort}}_{\mathsf{SEE}}$ is a join implementation that can be used when p_1 is an order relation. It materializes p_3 and sorts it, and then iterates over p_2 and joins its result with the tuples from p_3 which are now in a convenient order.)

P_1	P_2	P_3	P_4	P_5	P_6	*conds.*
u						
u			u			
u			u	u		$\mathrm{comm}(P_4, P_5)$
u			u	u	u	$\mathrm{comm}(P_4, P_5)$
u			u		u	
u				u		
u				u	u	
u					u	
			u			
			u	u		$\mathrm{comm}(P_4, P_5)$
			u	u	u	$\mathrm{comm}(P_4, P_5)$
			u		u	
				u		
				u	u	
					u	

Table 1: Unnesting Study.

5. EXPERIMENTS

We implemented our approach in the Galax XQuery processor [13], which we extended to support the core of the XQueryP language (without limitations on where side effects may occur). We also implemented side-effect analysis and modified the rewriting rules in the optimizer as described in this paper. We report on two classes of experiments, the first of which verifies the completeness and correctness of the compiler, while the second one looks at its performance.

5.1 Completeness and Correctness

We designed queries on the XMark [24] schema following the generic pattern of Scenario 3.5, and inserted updates into expressions E_1, E_4, E_5, E_6, such that the unnesting conditions are satisfied. The tests cover all possibilities of using updates in one, two, three or four of the expressions mentioned above. E_3 and F_2 did not contain update statements because E_3 always needs to be pure,[1] and into F_2 we could not insert updates because of limitations of the GroupBy implementation in the original compiler. Table 1 summarizes all cases in which unnesting can be performed, depending on conditions satisfied by plans P_i of each E_i. Each row characterizes one class of queries: the u annotation in column P_i indicates that P_i contains update statements and the *conds* column describes additional conditions to be checked. We

[1]The current implementation checks purity instead of the more general idempotency.

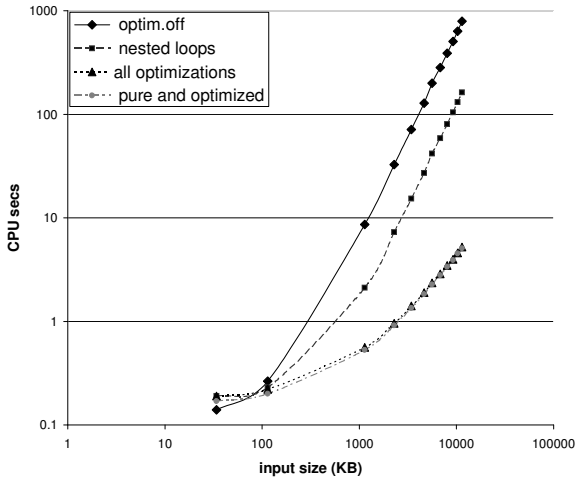

Figure 5: Running times for query q08.

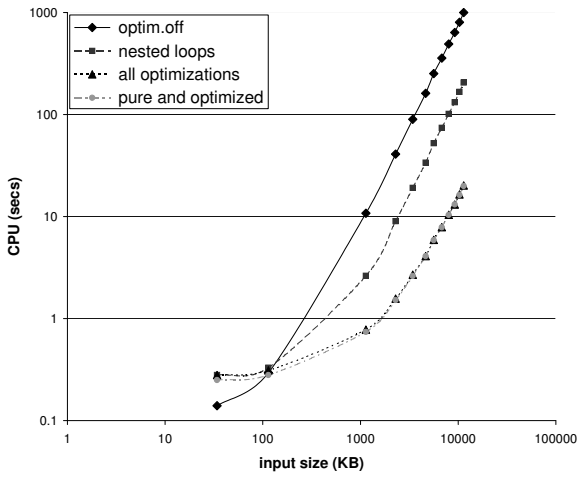

Figure 6: Running times for query q09.

mention that all these queries could be pipelined, *i.e.*, the output of the Map operators could be streamed and the left branch of the joins was evaluated lazily.

We ran the queries over a sample 1MB XMark document, checking that: (i) the optimized query plan is an instantiation of the generic optimized plan and (ii) the results obtained with and without optimizations are identical. We also tested queries covering all possibilities of adding updates inside the E_i expressions such that the unnesting conditions were not satisfied. These are representatives of the classes of queries from Table 1 for which the additional conditions are violated, together with all other combinations of u annotations not appearing in Table 1 (the cases in which E_3 contains updates). For them, we first checked that the optimizer does not push the MapConcat through GroupBy or that it does not introduce an outer join, depending on the condition that was violated. We verified that the evaluation of the partially optimized plans obtained for these queries produces the same results as in the case when optimizations are switched off (i.e., plans using maps and nested loops evaluation, resulting from naive compilation into the algebra). It is worth pointing out that both these tests were instrumental in finding several bugs in the implementation of the physical join operators and of the pipelining optimizations.

5.2 Performances

In order to validate the performance gains made possible by our optimizations, we tested XMark queries 8 through 12, which perform various types of joins and grouping. We modified these queries by adding updates in the inner return clauses, corresponding to E_5 from our general unnesting form. The updates consist in modifying the statistics of the number of transactions, considering each inner for-loop to correspond to a transaction. The experiments were run on a machine with a Pentium 4 CPU, 3.2 GHz, 1 GB of RAM, running the *etch* distribution of Debian GNU/Linux.

We ran these queries on documents up to 11MB in three different scenarios: with all optimizations turned off, with optimizations on, but joins executed as nested loops, and finally with joins executed efficiently using hash or sort-merge join algorithms. Since we rely on an existing compiler, our goal is essentially to demonstrate that we could recover the proper behavior when side effects are present. As a point of reference, we also provide the running times obtained with the same queries after removing all update statements (which we call *pure* queries) and enabling all optimizations.

Figure 5 shows the running times (on a log-log scale) for query q08, which is an instance of the generic Scenario 3.5 performing an equi-(outer)join. As one can see, with optimizations on, the pure and the impure version behave almost the same. Also, as expected, executing joins as nested loops reduces the running time, but does not change the asymptotic behavior. Similar results were obtained for query q10 (See Figure 7).

The results for query q09 are shown on Figure 6. This query is a nested 3-way join and even though optimization leads to a substantial improvement in running time, it does not change the asymptotic behavior very significantly. Still, we can see that the optimized query with updates does behave similarly to the one without updates, which seems to point to a limitation in the original compiler.

Queries q11 and q12 are very similar, which is why we only show the graph for q11, in Figure 8. These are queries that touch a larger set of data, using a *greater-than* predicate in the join condition. Turning all optimizations on, which includes selecting a sort-merge physical join operator, leads to substantial gains: for instance, for the 11MB document, running query q11 takes 21s, compared to 998s with the naive plan and q12 takes 22s, compared to 1,000s. Part of this gain is due to pipelining: if all intermediate results are materialized, evaluation of the optimized logical plans each take 44s. There is however a more noticeable increase in evaluation time than in q08–q10, explainable by the much larger number of updates that are performed (up to 118,271 updates compared to at most 3742 updates for q08–q10).

To provide a better image of how pipelining contributes to the overall optimization process, we display in Table 2 the evaluation times for q12 on a subset of the documents

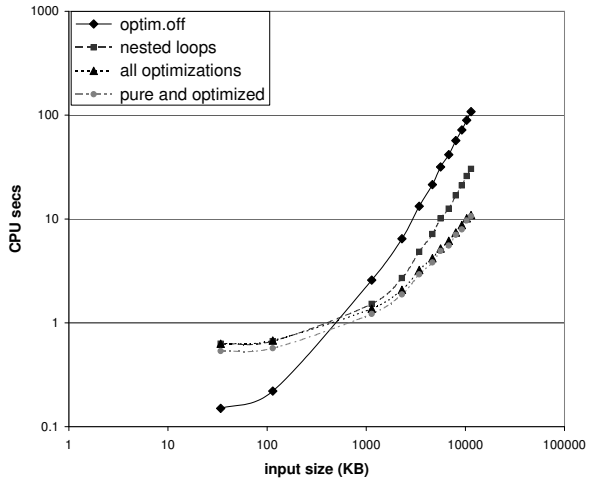

Figure 7: Running times for query q10.

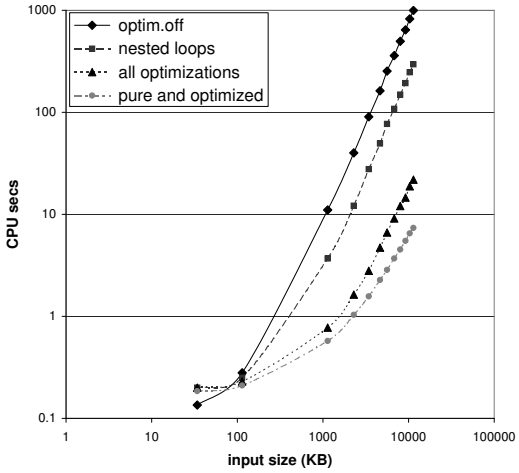

Figure 8: Running times for query q11.

doc.size	no optim.	joins, but no pipelining	all optim.
34 KB	0.13 s	0.23 s	0.22 s
114 KB	0.28 s	0.27 s	0.25 s
1135 KB	11.05 s	1.25 s	0.79 s
3425 KB	90.85 s	5.27 s	2.82 s
5616 KB	255.12 s	13.01 s	6.42 s
8003 KB	495.03 s	24.56 s	11.84 s
11397 KB	1000.36 s	44.90 s	21.52 s

Table 2: Comparative running times for q12.

we used. As we can see, join optimizations alone (third column in Table 2), including use of the proper join algorithm, reduces running time by one order of magnitude and adding pipelining (fourth column) provides an additional improvement with a factor between 1.5 and 2.1. Similar results were obtained for the other queries, namely a speedup factor between 1.32 and 3.43 (except for the 34KB and 114KB documents). The effect of pipelining is even more important—between 2.54 and 4.81 speedup for the non-optimized case—if we run these queries with pipelining turned on, but logical optimizations turned off. The explanation is that MapConcat and Select operators are used instead of Join and GroupBy, and require a significant amount of materialization. This seems to indicate that pipelining could be very important for fragment of the plans that cannot be optimized (*e.g.*, when some side conditions do not hold).

6. RELATED WORK

There has been almost no work on optimization for query languages in the presence of side effects. The closest related work to ours is [15], considering join optimization for the XQuery! language. However, it does not provide a detailed analysis of the side conditions needed during rewriting. In the functional programming community, state monads are commonly used to support imperative features in pure languages [22]. A generally accepted opinion is that lazy evaluation and side effects are, from a practical point of view, incompatible [18]. This is consistent with our approach in which we use an eager semantics, and only introduce limited lazyness for the purpose of pipelining. Monads have been also been used to support query languages semantics and implementation [3, 9, 26]. However, this work does not consider side effects. The only exception seems to be Fegaras [10], who studies optimization for an object-oriented query languages with updates, by using a state monad but without addressing algebraic optimization.

7. CONCLUSION

In this paper, we have presented a study of the impact of adding side-effects into an algebraic XML query compiler. We have shown that with the proper care and sufficient static analysis, one can safely extend a traditional database compiler with side-effecting operations. We believe this is an important step toward supporting a new generation of languages which blend database querying and imperative programming, such as XQuery scripting extensions. There is still much to do as future work, most notably in broadening the scope of rewritings being considered, and in the area of cost-based optimization.

Repeatability Assessment Result. All the results in this paper were verified by the SIGMOD repeatability committee. Code and/or data used in the paper are available at: `http://www.sigmod.org/codearchive/sigmod2008/`

Acknowledgements. We would like to thank Christopher Ré who worked on a preliminary study of the impact of side effects on XQuery optimization, and an early implementation, on which this paper is based.

8. REFERENCES

[1] BEA AquaLogic DSP documentation: Sample retail application. http://edocs.bea.com/aldsp/docs25/install/sampleapp.html.

[2] Scott Boag, Don Chamberlain, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language, W3C Recommendation, 2007.

[3] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[4] Michael Carey. Data delivery in a service-oriented world: the BEA AquaLogic Data Services Platform. In *SIGMOD Conference*, pages 695–705, 2006.

[5] Don Chamberlain, Michael Carey, Daniela Florescu, Donald Kossmann, and Jonathan Robie. XQueryP: Programming with XQuery. In *XIME-P*, 2006.

[6] Don Chamberlain, Daniela Florescu, and Jonathan Robie. XQuery Update Facility, W3C Working Draft 11 July 2006, 2007.

[7] James Clark. XSL Transformations (XSLT), W3C Recommendation, 1999.

[8] Sophie Cluet. Designing OQL: Allowing Objects to be Queried. *Information Systems*, 23(5):279–305, 1998.

[9] Leonidas Fegaras. Query Unnesting in Object-Oriented Databases. In *SIGMOD Conference*, pages 49–60, 1998.

[10] Leonidas Fegaras. Optimizing Queries with Object Updates. *J. Intell. Inf. Syst.*, 12(2-3):219–242, 1999.

[11] Steven Feuerstein and Bill Pribyl. *Oracle PL/SQL Programming*. O'Reilly, 2005.

[12] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: An XML programming language for Web service specification and composition. In *International conference on World Wide Web*, pages 65–76, May 2002.

[13] Galax: An Implementation of XQuery. "http://www.galaxquery.org/".

[14] Giorgio Ghelli, Nicola Onose, Kristoffer Høgsbro Rose, and Jérôme Siméon. A Better Semantics for XQuery with Side-Effects. In *DBPL*, pages 81–96, 2007.

[15] Giorgio Ghelli, Christopher Re, and Jérôme Siméon. XQuery!: An XML Query Language with Side Effects. In *EDBT Workshops*, pages 178–191, 2006.

[16] Giorgio Ghelli, Kristoffer Høgsbro Rose, and Jérôme Siméon. Commutativity Analysis in XML Update Languages. In *ICDT*, pages 374–388, 2007.

[17] Jan Hidders, Jan Paredaens, and Roel Vercammen. On the Expressive Power of XQuery-Based Update Languages. In *XSym*, pages 92–106, 2006.

[18] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In "Engineering theories of software construction", ed Tony Hoare, Manfred Broy, Ralf Steinbruggen, IOS Press, 2001.

[19] The LINQ Project. http://msdn.microsoft.com/XML/linqproject.

[20] N. May, S. Helmer, and G. Moerkotte. Strategies for Query Unnesting in XML Databases. *ACM Transactions on Database Systems*, 31(3):968–1013, 2006.

[21] Guido Moerkotte. Building Query Compilers, Draft Manuscript, December 2005. http://db.informatik.uni-mannheim.de/~moer.

[22] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *20th Symposium on Principles of Programming Languages*, North Carolina, USA, January 1993. ACM Press.

[23] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *ICDE*, page 14, 2006.

[24] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, pages 974–985, 2002.

[25] Philip Wadler. Web Development without Tiers. In *5th International Symposium on Formal Methods for Components and Objects*, 2006.

[26] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.