Data-Intensive XQuery Debugging with Instant Replay

Torsten Grust   Jan Rittinger   Jens Teubner

Technische Universität München
Munich, Germany

torsten.grust | jan.rittinger | jens.teubner@in.tum.de

ABSTRACT

We explore the design and implementation of *Rover*, a *post-mortem* debugger for XQuery. Rather than being based on the traditional breakpoint model, *Rover* acknowledges XQuery's nature as a functional language: the debugger follows a *declarative debugging* paradigm in which a user is enabled to observe the values of selected XQuery subexpressions. *Rover* has been designed to hook into *Pathfinder*, an XQuery compiler that emits relational algebra plans for evaluation on commodity relational database back-ends. The debugger instruments the subject query with `fn:trace()` calls which, at query runtime, populate database tables with relational representations of XQuery item sequences. Thanks to *Pathfinder*'s *loop-lifting* compilation strategy, a *Rover* trace (1) may span multiple XQuery `for` iteration scopes and (2) allows for interactive debugging sessions that can arbitrarily replay iterations in a unique forward/backward fashion. Since the query runtime as well as the debugger are database-supported, *Rover* is scalable and supports the observation of very data-intensive XQuery expressions.

1. DEBUGGING XQUERY

Our own experience has taught us that authoring a moderately complex XQuery expression from scratch more often than not tends to yield queries which exhibit unexpected behavior. Such bugs may sometimes have obvious fixes, but at times they might only occur with specific input XML instances or are, generally, hard to track down. While we do not postulate that XQuery authoring is inherently prone to error, there are a number of language characteristics and intricacies that may turn into pitfalls. Consider that

(1) much of the XQuery semantics is implicit and does not surface at the language level (*e.g.*, existential quantification or casting),
(2) expression evaluation depends on different notions of order (sequence and document order) some of which may be locally disabled (*e.g.*, via `unordered { }`),
(3) the interaction of atomic values and XML nodes calls for special attention and care,
(4) XQuery is a functional language and, admittedly odd, expressions like `let $x := $x + 1 return` e have a semantics different to what might be expected, and
(5) XQuery's orthogonal syntax and compositional nature—typical for an expression-oriented language—encourages (inexperienced) authors to write complex and deeply nested expressions.

The inherent irregularity of XML data as well as issues related to validation and dynamic typing add to this list to make a convincing case for appropriate XQuery debugging technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
XIME-P 2007, 4th International Workshop on XQuery Implementation, Experiences and Perspectives, July 15, Beijing, China
Copyright 2007 ACM 978-1-59593-800-8 ...$5.00.

```
let $days := doc("http://xoap.wea···?dayf=3")//day
    $when := <when><today num="1"/>
                   <tomorrow num="2"/></when>
    $nums := $when//@num
return
  <lows>
    { for $day in $nums
      return $days[$day]/low }
  </lows>
```

Figure 1: Query $Q_{\texttt{low}}$: Extracting today's and tomorrow's low temperatures from a 3-day forecast.

```
<weather>
  <loc><dnam>Beijing, China</dnam></loc>
  <dayf>
      ⎧<day t="Sunday"><hi>10</hi><low>2</low></day>
  n × ⎨  ⋮
      ⎩  ⋮
  </dayf>
</weather>
```

Figure 2: Sketch of the relevant WeatherChannel® XML n-day weather forecast data.

Processing XML weather forecast data. As a real-life example of an XQuery expression that unexpectedly went wrong, let us turn to Query $Q_{\texttt{low}}$ of Figure 1. $Q_{\texttt{low}}$ and the upcoming queries in this paper operate over WeatherChannel® XML weather forecast data. Figure 2 shows the relevant fragments of these XML instances.[1] The query extracts today's and tomorrow's low temperature values from a 3-day forecast. Variable `$days` will thus be bound to a sequence of `day` elements of length 3 with today's data at sequence position 1. In `$when`, we mnemonically specify

[1]An n-day forecast ($n = 1, 2, \ldots, 10$) may be retrieved via URL `http://xoap.weather.com/local/`*city*`?dayf=`n where *city* represents a city code (*e.g.*, Beijing, China $\widehat{=}$ `CHXX0008`).

the days of interest (today, tomorrow) and extract the relevant sequence positions $(1,2)$ into `$nums`. Finally, the `for` iteration indexes the `$days` sequence to extract the associated `low` elements. A result like `<lows><low>2</low><low>5 </low></lows>` is expected, but instead we get

```
<lows><low>2</low><low>5</low><low>3</low>
      <low>2</low><low>5</low><low>3</low></lows> .
```

Oops. (Section 3 will shed light on this bug.)

1.1 Observing XQuery Expressions

The almost functional XQuery semantics, based on the side-effect free evaluation of expressions, naturally leads to a *declarative* style of debugging which centers around the *observation of expression values.* Indeed, an imperative debugging style that instead emphasizes the setting of breakpoints and subsequent inspection of program state, *e.g.*, the current value of a loop index variable, is inappropriate: in the XQuery expression `for $x in` $(i_1,\dots,i_n)$ `return` e, loop body e may be evaluated for all n bindings of `$x` in parallel—a notion of "current" or "next state" seems inapplicable.

In declarative XQuery debugging, we observe the values the expression e will have in the *different iterations* it will be evaluated in. These evaluations only depend on e and the values of all its free variables (like `$x`) but *not* on the order of evaluation or any preceding computation.

1.2 Post-Mortem Debugging with *Rover*

Rover noninvasively implements a declarative debugger for XQuery with the help of the standard built-in function `fn:trace()`:

`fn:trace(`e `as item()*,` s `as xs:string) as item()*` .

While `fn:trace(`e`,`s`)` simply evaluates and returns the value of e, the runtime system is free to perform any side effect during function call evaluation, *e.g.*, print the debug message s. *Rover*'s specific iteration-aware implementation of `fn:trace()` facilitates the post-mortem observation of expression e. To debug a complex XQuery expression, its author wraps interesting or suspect subexpressions in calls to `fn:trace()`. Alternatively, an interactive debugger might instrument a query with `fn:trace()` calls on the author's behalf. In such an interactive setup, it would also be feasible to instrument *all* subexpressions of a query to obtain a full evaluation trace.

Debugging data-intensive queries. XQuery has been designed as a data processing language and queries may inspect as well as construct XML instances of significant size. Since we want to strictly avoid to have authors rewrite their queries only to touch less data during debugging—this may hide old or introduce new bugs and generally gives way to "Heisenbugs"—*Rover* is prepared to cope with massive trace data. Unlike traditional program debuggers, *Rover* needs to be able to observe expression values whose representation size exceeds heap memory, even more so since a trace records the values of an instrumented subexpression for all its iterations.

To this end, *Rover*, which has been designed to debug expressions that have been translated by the XQuery compiler *Pathfinder* [9], receives excellent support from *Pathfinder*'s relational database back-end. *Pathfinder* emits relational algebra plans which faithfully implement the XQuery semantics in terms of operations over relational encodings of sequences of items—XML nodes or atomic values—and tree fragments. The execution of a *Rover*-instrumented query will, as a side effect, populate the database with trace tables. With *Rover*, we may observe values of any type and size: *Pathfinder*'s item encoding has been designed both, to embrace all XQuery types and to be stored in regular flat relational tables—no DOM or similar pointer-based XML representations are used.[2]

Finally, to perform interactive post-mortem debugging, the *Rover* front-end connects to the relational database back-end to access the trace tables.

We proceed as follows. The upcoming Section 2 reviews the relevant aspects of *Pathfinder*'s *loop-lifting* XQuery compilation strategy and lays the ground for the observation of expressions with forward/backward replay of iterations. Section 3 shows how *Rover* helps to debug Query $Q_{\texttt{low}}$ of Figure 1 and also sketches an alternative XQuery-based debugger client that operates over an XML serialization of the trace tables. The creation and inspection of large evaluation traces is the topic of Section 4. We review related research in Section 5 and close with remarks on work in flux (Section 6).

2. LOOP-LIFTED XQUERY COMPILATION

The *Pathfinder* XQuery compiler has been developed under the main hypothesis that the well-understood infrastructure of relational database kernels can also make for highly efficient XQuery processors. This purely relational approach to XQuery evaluation indeed yields scalable XQuery implementations [2]—provided that suitable relational encodings of (1) XML tree fragments [7] and (2) the dynamic semantics of XQuery [9] are used that allow the database back-end to play its trump: *set-oriented evaluation.* In this mode of evaluation, the database system applies an operation to *all* rows in a table. In absence of inter-row dependencies, the system may process the individual rows in any order or even in parallel.

To actually operate the database back-end in this set-oriented manner, *Pathfinder* draws the necessary amount of independent work from XQuery's `for` loops. In XQuery, the evaluation of a subexpression e in the scope of a `for` loop yields an ordered sequence of zero or more items in each loop iteration. In *Pathfinder*'s relational encoding, these items are laid out in a *single table* for *all loop iterations*, one item per row (we discuss the exact table layout below). An algebraic plan consuming this *loop-lifted* or "unrolled" representation of e may, effectively, process the results of the individual iterated evaluations of e in any order it sees fit—or in parallel. In some sense, such a plan is the algebraic embodiment of the independence of the individual evaluations of an XQuery `for` loop body (Section 1.1). A welcome consequence of loop-lifting is that the underlying relational engine does *not* need to provide a dedicated iteration primitive: the operators of the relational algebra inherently iterate over the rows of their input table(s) which already realizes the XQuery iteration semantics. Details of the loop-lifting technique beyond the scope of this paper are covered in [9].

2.1 Iteration Scopes

In the loop-lifting compiler, the XQuery `for` clause is the core language construct. Any expression e is considered to

[2] *Rover* abbreviates *Relational observation of expressions.*

```
    ┌ for $city in ("CHXX0008","GMXX0087")          ①
    │  ┌ let $dayf := doc(concat("http:···",$city,"?dayf=3"))
    │  │ return ②                         ③
    │  │   for $d in $dayf//day
 s0 │ s1│   return                         ④
    │  │     ┌ <hi city="{ $dayf//loc/dnam }">
    │  │  s2 │   { $d/@t, data($d/hi) }  ⑤
    │  │     └ </hi>                       ⑥
    └  └
```

```
(<hi city="Beijing, China" t="Sunday">10</hi>,
 <hi city="Beijing, China" t="Monday">11</hi>,
 <hi city="Beijing, China" t="Tuesday">9</hi>,
 <hi city="Munich, Germany" t="Sunday">4</hi>,
 <hi city="Munich, Germany" t="Monday">8</hi>,
 <hi city="Munich, Germany" t="Tuesday">7</hi>
)
```

Figure 3: Query $Q_{\texttt{hi}}$: Beijing and Munich high temperatures for the next three days, result on the right. The s_i denote iteration scopes, the ⓧ mark instrumented expressions (relational observations shown in Figure 4).

① `$city`

iter	pos	item
1	1	"CHXX0008"
2	1	"GMXX0087"

② `$dayf` (s_1)

iter	pos	item
1	1	$\texttt{doc}_1$
2	1	$\texttt{doc}_2$

③ `$dayf//day`

iter	pos	item
1	1	$\texttt{day}_1$
1	2	$\texttt{day}_2$
1	3	$\texttt{day}_3$
2	1	$\texttt{day}_4$
2	2	$\texttt{day}_5$
2	3	$\texttt{day}_6$

④ `$dayf` (s_2)

iter	pos	item
1	1	$\texttt{doc}_1$
2	1	$\texttt{doc}_1$
3	1	$\texttt{doc}_1$
4	1	$\texttt{doc}_2$
5	1	$\texttt{doc}_2$
6	1	$\texttt{doc}_2$

⑤ `···/dnam`

iter	pos	item
1	1	$\texttt{dnam}_1$
2	1	$\texttt{dnam}_1$
3	1	$\texttt{dnam}_1$
4	1	$\texttt{dnam}_2$
5	1	$\texttt{dnam}_2$
6	1	$\texttt{dnam}_2$

⑥ `data(···)`

iter	pos	item
1	1	"10"
2	1	"11"
3	1	"9"
4	1	"4"
5	1	"8"
6	1	"7"

map_{s_0,s_1}

outer	inner
1	1
1	2

map_{s_1,s_2}

outer	inner
1	1
1	2
1	3
2	4
2	5
2	6

Figure 4: Relational observations (trace tables) recorded during the evaluation of the instrumented expressions of the query in Figure 3. Right of dashed line: *map* relations connecting the iteration scopes s_0, s_1, s_2.

be in the scope of its innermost enclosing `for` loop—in case e is a top-level expression, we assume the presence of a "ghost loop" `for $_ in (0) return` e iterating over a singleton where `$_` may not occur free in e. We will refer to this top-level scope by s_0. Due to XQuery's compositionality, the `for` iteration scopes in a query form a tree-shaped hierarchy. In Query $Q_{\texttt{hi}}$ (Figure 3), the top-level scope plus the two nested `for` loops form the linear scope hierarchy s_0—s_1—s_2. Query Q_{12} of the W3C XQuery Use Case *STRONG* [4], for example, exhibits a scope nesting of the form $s_0 <^{s_3}_{s_1 - s_2}$.

Note how Query $Q_{\texttt{hi}}$ requests a 3-day weather forecast to collect the high temperature values for Beijing and Munich into a sequence of `hi` elements. The result is shown on the right of Figure 3. While the query's scope s_0 is iterated once only, there will be two iterations in scope s_1, one for each binding of variable `$city`. In both iterations, variable `$d` will iterate over a sequence of `day` elements. Since, for the example run of Figure 3, the weather forecast data contained three `day` elements for both cities, there will be a total of $2 \times 3 = 6$ iterations in the innermost scope s_2. (For other XML instances there might be more or less iterations—the algebraic plan determines the exact number at query runtime.)

2.2 Relational Observations

Given the algebraic plan for Query $Q_{\texttt{hi}}$, what exactly will we be able to observe if we instrument selected subexpressions? In Figure 3, six subexpressions have been marked for observation.[3] For each observed subexpression e, the *Pathfinder* compiler introduces a *trace* operator $\curlyvee$ in the emitted plan that saves the relational observation for e in the database (Section 4).

Figure 4 depicts the relational observations made for the six instrumented subexpressions (left of dashed line). As a consequence of the loop-lifting compilation strategy, each trace table contains the expression values observed in all iterations: a row $[i, p, v]$ in such an iter|pos|item table indicates that, in iteration i, the observed expression evaluated to item v at sequence position p. Note that, even though XQuery expressions evaluate to item *sequences*, the relational representation still uses flat 1NF tables. Further, trace tables only store XML node identifiers (as opposed to whole XML fragments) in the item column. Since the interactive post-mortem debugger acts as a database client, it can use an element node identifier like $\texttt{day}_1$ to look up and serialize the element's content on demand.

[3]An instrumentation `fn:trace(`e`,"`ⓧ`")` is indicated by $\underset{ⓧ}{\llcorner e}$ in Figure 3.

As expected, we find two observations (iter $\in \{1, 2\}$) for the subexpressions ①–③ located in iteration scope s_1 and six observations (iter $\in \{1, \ldots, 6\}$) for ④–⑥ which are in scope s_2. An interactive debugger can, for example, (1) read table ② to visualize that variable `$dayf` is bound to two different document nodes ($\texttt{doc}_1$, $\texttt{doc}_2$) in separate iterations, or (2) access table ③ to visualize that expression `$dayf//day` evaluates to a sequence of three `day` elements in each of the two iterations. To observe the three expressions `$city`, `$dayf`, and `$dayf//day` (all in scope s_1) together, the debugger would perform the relational equi-joins ① $\bowtie_{\text{iter}}$ ② $\bowtie_{\text{iter}}$ ③. An interactive debugger UI could then use the joined table data to let the user browse forward and backward through iterations and learn how the values of the expressions change in synchronization.

Note that there are occurrences of variable `$dayf` in two scopes, s_1 and s_2 (observed at ② and ④, respectively). Since `$dayf` is bound in s_1, for each iteration of the outer `for` loop, the variable's value will appear constant in the iterations of the inner `for` loop in scope s_2. This correspondence between the iterations of the outer and inner `for` loops is captured by binary relation map_{s_1,s_2} with schema outer|inner (Figure 4, right of dashed line). During the first iteration of the outer loop, for example, the inner loop performs its first three iterations, thus $[1, \text{inner}] \in map_{s_1,s_2}$ with inner $\in \{1, 2, 3\}$. At query runtime, the *Pathfinder*-generated algebraic plans use these inter-scope *map* relations to derive the relational representation of variables in scopes deeper than their binding site. For `$dayf`, we have (let $\pi_{\text{a:b}}$ rename column b into a):

```
        let $days := doc("http:···?dayf=3")//day
            $when := <when><today num="1"/>
              ④               <tomorrow num="2"/></when>
            $nums := $when//@num
        return
  s0      <lows>
            { for $day in $nums
                                  ③           ①
                return s1 [ $days[$day]/low }
                                   ②
          </lows>
```

Figure 5: Instrumented buggy Query $Q_{\texttt{low}}$ (instrumentations added in order ①...④).

① ···/low

iter	pos	item
1	1	$\texttt{low}_1$
1	2	$\texttt{low}_2$
1	3	$\texttt{low}_3$
2	1	$\texttt{low}_4$
2	2	$\texttt{low}_5$
2	3	$\texttt{low}_6$

② $days[$day]

iter	pos	item
1	1	$\texttt{day}_1$
1	2	$\texttt{day}_2$
1	3	$\texttt{day}_3$
2	1	$\texttt{day}_4$
2	2	$\texttt{day}_5$
2	3	$\texttt{day}_6$

③ $day

iter	pos	item
1	1	$\texttt{@num}_1$
2	1	$\texttt{@num}_2$

④ $nums

iter	pos	item
1	1	$\texttt{@num}_1$
1	2	$\texttt{@num}_2$

Figure 6: Relational observations made during the debugging session for $Q_{\texttt{low}}$.

$$④ = \pi_{\mathsf{iter:inner,pos,item}}(② \bowtie_{\mathsf{iter=outer}} map_{s_1,s_2}) \ .$$

At debug time, we can use the same mechanism to monitor any expression e in the context of an arbitrary nested scope, even if e did not occur in that scope in the subject query. For example, to observe variable `$city` (bound in scope s_1) in synchronization with the expression `$dayf//loc/dnam` of scope s_2, *e.g.*, to relate the city code and its WEATHERCHANNEL® display name, the debugger would evaluate the query $\left(\pi_{\mathsf{iter:inner,pos,item}}(① \bowtie_{\mathsf{iter=outer}} map_{s_1,s_2})\right) \bowtie_{\mathsf{iter}} ⑤$.

Rover will, consequently, also place $\curlyvee$ operators on top of those subplans that compute the *map* relations [9] to ensure their availability at debug time. The number of saved trace tables thus equals the number of `fn:trace()` occurrences plus the number of edges connecting those scopes in the scope hierarchy that contain instrumented expressions (*i.e.*, $6 + 2$ for the example of Figure 3).

3. BUGS UNDER OBSERVATION

Let us return to the defective Query $Q_{\texttt{low}}$ of Figure 1 and try to find and fix the bug with the support of *Rover*'s observations. Such a debugging session starts out with the instrumentation of subexpressions. Here we choose to not fully instrument every subexpression but instead work our way back from the phenomenon that the resulting `lows` element contains too many `low` child nodes (six instead of the expected two).

(1) We let *Rover* wrap the path expression `$days[$day]/low` in a call to `fn:trace()` (observation ① in Figure 5) and indeed observe an overall of six `low` elements, three per iteration. (The debugger can group the trace table ① on column `iter` to render the relational observation in the familiar XQuery syntax (`<low>2</low>,...`).) This is unexpected, since a `day` element contains a single `low` element (Figure 2).

(2) How many `day` elements are there? We instrument expression `$days[$day]` (observation ②) to find six `day` nodes, again three each for today and tomorrow instead of the expected single node per iteration. The XPath predicate does not seem to have any filtering effect.

(3) We observe variable `$day` and its binding sequence `$nums` from scope s_0 together (observations ③ and ④). Both contain `@num` attribute nodes with values `"1"` and `"2"`. But the *effective Boolean value* of an attribute node is *true*, regardless of its contents [1]!

Thus, to fix this bug and to correctly index the `$days` sequence, we either (1) need to convert `$day` into a number and could write `$days[number($day)]/low` or (2) validate the incoming WEATHERCHANNEL® data to annotate the `@num` attributes nodes with a numeric type and modify Query $Q_{\texttt{low}}$ to read `let $nums := data($when//@num)`.

Since most XQuery authors tend to have quite a clear idea of the expected value of a given subexpression, we have found expression observation to be an effective debugging methodology. For example, an "imperative variable update" bug like

```
let $x := 1
for $n in 1 to 3                  ①
  let $x := $x + 1 return $x ,
```

surfaces as an unexpected constant observation of variable `$x` being bound to value 2 (see column `item` in the associated trace table on the right), regardless of which iteration the user replays.

① $x

iter	pos	item
1	1	2
2	1	2
3	1	2

Debugging and static typing. Since *Rover* depends on its side effect, the algebraic *trace* operator $\curlyvee$ is exempt from being removed by *Pathfinder*'s algebraic optimizer (see Section 4). Nevertheless, a user's instrumentation may vanish already early in the compilation process if the XQuery compiler simplifies expressions during XQuery Core normalization [5] or otherwise statically detects that a subexpression does not contribute to the query's result. In case the compiler implements XQuery's static typing feature, one such effect may be witnessed in Query $Q_{\texttt{low}}$: the static type of the instrumented expression ③ (variable `$day`) in Figure 5 will be inferred as `attribute(num,untypedAtomic)`. The *effective Boolean value* of any expression of this type is *true*, so that the compiler simplifies the path expression in scope s_1 to `$days/low`. In effect, in an optimized subject query, observation ③ would be unavailable.[4] In a functional language like XQuery, however, such unexpected disappearances of instrumentations are quite dependable evidences of bugs (as is the case for $Q_{\texttt{low}}$).

XQuery-based debugger clients. While *Pathfinder* originally records *relational* observations, *Rover* can reuse the existing serialization infrastructure in the database back-end to offer an integrated XML view of the trace tables (Figure 7). In this serialization, the nesting of the `scope` elements reflects the query's scope hierarchy (Section 2). Observations (`<trace>···</trace>`) made in the same scope and iteration are grouped below a common `iteration` parent, *i.e.*, the XML format already materializes the required trace table equi-joins on column `iter` discussed in Section 2.2.

Since *Rover* inlines observed XML nodes as the contents of the `item` elements, this serialization paves the way for *purely XQuery-based* debugger clients. The interactive shell of such a debugger can offer the already familiar XPath language to

[4] This relates to "`No symbol "···" in current context.`" messages in procedural debuggers like GNU's *gdb*.

```
<scope id="s0">
 <iteration iter="1">
  <scope id="s1">
   <iteration iter="1">
    <trace msg="①">
     <item pos="1"><low>2</low></item>
     <item pos="2"><low>5</low></item>
     <item pos="3"><low>3</low></item>
    </trace>
    <trace msg="②">
     <item pos="1"><day t="Sunday">···</day></item>
     <item pos="2"><day t="Monday">···</day></item>
     <item pos="3"><day t="Tuesday">···</day></item>
    </trace>
   </iteration>
   <iteration iter="2">···</iteration>
  </scope>
  <trace msg="④">···</trace>
 </iteration>
</scope>
```

Figure 7: XML-serialized observations of Query $Q_{\texttt{low}}$ (excerpt; iteration 2 in scope s_1 omitted for brevity).

Instrumentation	Q_{13} # items (rows)	Q_{13} ⏱ (ms)	Q_{16} # items (rows)	Q_{16} ⏱ (ms)
none	0	61.2	0	61.0
partial	16, 525	63.9	30, 302	65.1
full	38, 586	70.3	168, 898	92.3

Table 1: XMark queries Q_{13}, Q_{16}: size of relational observations and evaluation times for increasing density of instrumentation (MonetDB/XQuery).

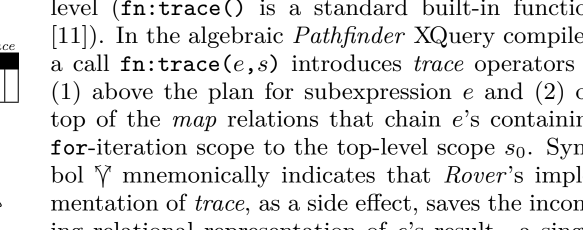
dissect and zoom into observations. For the Query $Q_{\texttt{low}}$, for example, the XPath expression

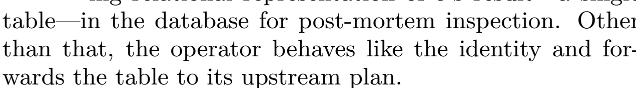
```
//scope[@id > s0]//trace[item[@pos > 1]]
```

extracts all non-top-level observations that yield item sequences of two or more elements (the query author expected all observed subexpressions in scope s_1 to yield singletons). Here, the debugger would display the two suspect `trace` elements with `@msg` values "①" and "②" that we have already used earlier to successfully track down the bug.

4. LARGE EVALUATION TRACES

Rover has been designed to be unobtrusive on its hosting XQuery processor. No changes are required on the language level (`fn:trace()` is a standard built-in function [11]). In the algebraic *Pathfinder* XQuery compiler, a call `fn:trace(`e,s`)` introduces *trace* operators ⋎ (1) above the plan for subexpression e and (2) on top of the *map* relations that chain e's containing `for`-iteration scope to the top-level scope s_0. Symbol ⋎ mnemonically indicates that *Rover*'s implementation of *trace*, as a side effect, saves the incoming relational representation of e's result—a single table—in the database for post-mortem inspection. Other than that, the operator behaves like the identity and forwards the table to its upstream plan.





These straightforward semantics of ⋎ restricts *Rover*'s impact on the relational database systems that the retargetable *Pathfinder* compiler can use as its back-end. For the *MonetDB*-based back-end[5]—an extensible database kernel in which all tables undergo full vertical fragmentation—the implementation of ⋎ simply marks its input table to persist after query evaluation has completed. Because *MonetDB*'s evaluation engine fully materializes all intermediate query results, ⋎ neither leads to any extra table allocation nor performs additional copy work.

This minimal runtime influence on a debugged query is reflected in Table 1. The table reports on the overall number of observed items (one item $\widehat{=}$ one trace table row) and the query evaluation times for the XMark [15] Queries Q_{13} and Q_{16} run against a 58 MB XML instance ($\approx 2,340,000$ nodes, XMark scale factor 0.5). The density of instrumentation was varied: *none* represents the original non-debugged benchmark queries while the *full* case instrumented each XQuery subexpression to obtain a full computation trace that provides the maximum choice of possible observations at debug time. For *partial*, only selected expressions were instrumented (in the spirit of Figure 5).

The measurements of Table 1, made on a dual 3.2 GHz Intel Xeon™ processor system equipped with 8 GB main memory, show *Rover*'s typical performance penalty of about 5% for partially instrumented queries and 15–50% for full traces.

However, most of the performance we sacrifice is *not* due to the relatively small overhead incurred by ⋎. Although the introduction of ⋎ operators does not reshape the original query plan, their presence may hinder such reshaping. A full trace scatters ⋎ operators all over a plan which may bring core parts of *Pathfinder*'s algebraic optimizer to almost a standstill: the compiler is forced to faithfully preserve the side effects of all ⋎ occurrences. In specific cases, this can have significant consequences for both, the performance of the observed query and trace table cardinality. The instrumentation of a subexpression in the innermost `for` loop of XMark Query Q_8, for example, disables *Pathfinder*'s logic that otherwise would rewrite the query's nested iteration into a relational join [8]. Query evaluation time grows from 225 ms (*none*) to 2, 764 s (*full*) for the 58 MB XML instance—the full computation trace observes no less than 1, 616, 694, 891 items.

Loop-lifted *vs.* iterated observation. While this number appears overwhelming, it is encouraging to see *Rover*'s set-oriented observation model scale to such extremes. In a loop-lifted plan, there will be a single invocation of ⋎ for each instrumentation, regardless of the number of iterations or items the operator will observe. An XQuery processor with an iterated evaluation strategy, on the other hand, will not be able to share the overhead of tracing among observations: each iteration will invoke (the internal representation of) `fn:trace()` anew. *Saxon 8.9* [10], an XQuery processor of the latter category, shows this behavior. For the 58 MB XMark instance, the `for` loop contained in Query Q_{16} will be iterated 4,875 times, with the described negative impact on any instrumentation placed inside the loop's body: compared to *Saxon 8.9*, *MonetDB/XQuery* requires 1/3 of the evaluation time for the non-debugged query but less than 1/100 of the time for a full trace.

[5] *Pathfinder* and *MonetDB* form the relational XQuery processor *MonetDB/XQuery* [2] (`pathfinder-xquery.org`).

"What if" debugging. *Rover* supports a style of exploratory debugging in which the user is enabled to *force* an instrumented expression e to yield value v and then restart query evaluation to answer the question "*What if e had value v instead?*" Internally, this leads to updates on e's trace table. For example, to force the bindings $\{1, 2\}$ instead of $\{$`@num`$_1$, `@num`$_2\}$ for variable `$day` in Query $Q_{\texttt{low}}$, *Rover* would update column item of trace table ③ in Figure 6. To shield the user and the query processor from havoc through arbitrary updates, *Rover* uses static sequence type information (*e.g.*, occurrence indicators) and derived relational properties of the original trace table (foremost functional dependencies like iter, pos $\rightarrow$ item [8]) to ensure consistent forcing. The actual query restart process benefits from the fact that the format of *Rover*'s trace tables and *Pathfinder*'s runtime tables coincide: the algebraic subplan for e is removed and its upstream plan is directly fed from e's updated trace table instead.

5. RELATED RESEARCH

The idea of declarative or algorithmic debugging which centers around the construction and navigation of computation trees ($\widehat{=}$ full traces), has been introduced in [16], originally for Prolog. Such debuggers often rely on far-reaching (source-level) transformations of the subject program [14] to prepare the collection of traces—which renders *Rover*'s `fn:trace()`/$\curlyvee$-based approach even less invasive. Program transformations like *lambda-lifting* [13] are necessary to observe expressions with free variables, for example. Instead, *Rover* relies on scope *map* relations (Section 2.2).

The declarative debugging literature has repeatedly advocated the use of database support to cope with the sizable trace data generated by faulty real-world subject programs [3, 6]. In [6], the program trace query language PTQL is proposed to inspect such database-resident traces. While PTQL is a SQL dialect, its basic idea is closely related the XQuery-based debugger client we sketched in Section 3. Loop-lifting further helps to control the fan-out of the computation tree [3] whose shape remains isomorphic to the query's algebraic plan independent of the number observed iterations.

Finally, *Rover*'s replay of `for` iterations bears a close resemblance with the *instant replay* feature of debuggers for concurrent programs [12]. A forward/backward traversal of the trace tables leads to consistent "replays", even for side-effecting XQuery concepts like node constructors (*e.g.*, two nodes constructed in distinct trees will always exhibit the same document order relationship) [1, § 2.4.1].

6. WORK IN FLUX

Rover already provides all the infrastructure required to bring a diagnostic debugging approach to XQuery that goes a step beyond what we have described so far. Debuggers of this type, particularly widespread in the functional programming language domain [14], traverse the computation tree top-down to automatically generate a (minimal) set of simple *yes/no questions* about the observed and expected behavior of functions [16]. The user's responses then guide the tree traversal to identify suspect functions in the subject program.

This truly declarative, functional style of debugging can be applied to XQuery if we regard an XQuery expression as a function of its free variables. Expression `$days[$day]/low` in scope s_1 of $Q_{\texttt{low}}$ would be identified with the function $f(x) =$ `$days[`x`]/low` (the `let`-bound variable `$days` would be considered a constant). To assess whether $f(x)$ behaves as expected, the debugger feeds observed bindings for x into f to generate questions in the style of "*If* `$day` *is* `@num`$_1$, *is* `(<low>2</low>,<low>5</low>,<low>3</low>)` *the expected value of* `$days[$day]/low`*?*" The presence of the database back-end would enable *Rover* to mark rows as *(un)expected.* Such marks help to minimize the question set (a *no* response to the above question would immediately flag f and thus `$days[$day]/low` as suspect) and recall user responses to avoid tedious repetitive questions—a challenge posed in [3].

Acknowledgments. This research is supported by the German Research Council (DFG) under grant GR 2036/2-1.

7. REFERENCES

[1] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3 Consortium, 2007.

[2] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. SIGMOD*, 2006.

[3] R. Caballero, C. Herrmans, and H. Kuchen. Algorithmic Debugging of Java Programs. In *Proc. WFLP*, 2006.

[4] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. W3 Consortium, 2007.

[5] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, 2007.

[6] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational Queries over Program Traces. In *Proc. OOPSLA*, 2005.

[7] T. Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD*, 2002.

[8] T. Grust. Purely Relational FLWORs. In *Proc. XIME-P Workshop*, 2005.

[9] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, 2004.

[10] M. Kay. The Saxon XSLT and XQuery Processor. `http://www.saxonica.com/`.

[11] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3 Consortium, 2007.

[12] C.E. McDowell and D.P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4), 1989.

[13] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1986.

[14] B. Pope and L. Naish. Practical Aspects of Declarative Debugging in Haskell 98. In *Proc. PPDP*, 2003.

[15] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, 2002.

[16] E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.