An Algebraic Approach to XQuery View Maintenance

J. Nathan Foster
University of Pennsylvania
jnfoster@cis.upenn.edu

Ravi Konuru Jérôme Siméon Lionel Villard
IBM TJ Watson Research Center
{rkonuru,simeon,lvillard}@us.ibm.com

Abstract

View maintenance is a problem in data management that arises whenever a view is materialized over a source that changes over time. When the source is large, or when the source and view reside on different hosts, it is not practical to recompute the view and retransmit it over the network each time the source is updated. A better idea, commonly used in systems built with view maintenance in mind, is to translate source updates to ones that can be applied to the view directly. The cost of calculating, transmitting, and applying a translated update is typically dramatically less than the cost of recomputing and retransmitting the entire view.

This paper addresses the problem of maintaining XQuery views over XML data. The core algorithm translates updates through queries as expressed in the tree algebra used internally in the Galax engine. This algorithm extends previous work on maintenance for relational views, although there are significant complications due to the data model, which is both ordered and nested. To overcome these obstacles, we propose a scheme for storing auxiliary data that guides the translation of updates in this more complicated setting. A novel aspect of our approach compared to previous work is that the amount and content of annotations can be controlled by users, making it possible to balance the tradeoffs between the size of the auxiliary data and the quality of translated updates.

We have built a prototype implementation to test these ideas. Our system extends Galax, and handles a core set of operators and built-in functions capable of expressing many typical first-order queries. Its design is fully compositional, so it can easily be extended to new operators. We present preliminary results of experiments run on benchmark queries from the XMark suite.

Categories and Subject Descriptors H.2.4 [*Information Systems*]: Database Management—Query Processing

General Terms Algorithms, Languages, Performance

Keywords XML, XQuery, materialized views, incremental maintenance, tree algebra

1. Introduction

There is a well-known story about Carl Gauss. His schoolteacher set an apparently lengthy arithmetic problem—add up the numbers from 1 to 100—but Gauss derived the formula $\sum_1^n = \frac{n(n+1)}{2}$ and determined the answer instantly. The story usually ends there. But imagine that Gauss's teacher had then asked the class to recompute the sum, but to omit the number 50 from the input sequence. Even a much less clever student would realize that maintaining the answer already computed, by subtracting 50, is simpler than recomputing the whole thing from scratch.

ACM SIGPLAN Workshop on Programming Languages Technologies for XML
9 January 2008, San Francisco, U.S.A.

An analogous problem arises in data management whenever a view is materialized over a source that changes over time. In these situations, incrementally maintaining the view, by translating source updates to view updates, is often much cheaper than recomputing the entire view. This paper addresses the problem of maintaining XQuery views over XML data.

XQuery is a W3C-recommended language for querying, transforming, and (in recent extensions) updating XML data [1, 3]. XQuery views arise in a variety of real-world settings; the following list describes just a few characteristic use cases:

- In an online auction site, the web page for a single item can be generated as a view over an XML source that contains the data for the items, buyers, and sellers registered with the site.
- In a web-based employee record application, access restrictions to sensitive personal data such as social security numbers, salaries, and performance evaluations, can be enforced using security views.
- In a scientific application, a legacy tool can be retrofitted to work with data in new formats using a view that transforms data in the new format back to the old one. As a concrete example, the UniProtKB protein sequence database is represented in XML, but many tools expect data in the original ASCII format.

A common feature to all these use-case scenarios is that the views are *materialized* and not virtual: the web pages generated for the auction site and employee record application are serialized onto the network and displayed in a client's web browser, and the ASCII view of the protein sequence database is written out to the filesystem and handed off to the legacy tool. Views have to be materialized when the source and view reside on different hosts—e.g., in web applications. Another common use of materialized views is to optimize the performance of query answering: when several queries need to be posed over the same view, it is usually more efficient to cache a copy of the view. However, despite these advantages, materialized views also come with complications: because data is replicated in several places, whenever the source is updated, the view also needs to be refreshed to keep the data consistent.

A simple way to refresh the view is to recompute the query on the updated source. However this strategy is impractical when the size of the source is large compared to the view, and in settings where the source and view are stored on difference hosts (in the latter case, the cost of transmitting the updated view over the network can be prohibitive.) A better strategy, which addresses both of these issues, is to incrementally maintain the view by translating source updates to corresponding view updates—i.e., such that updating the source and recomputing the view yields the same result as applying the translated update to the view. Figure 1 depicts the architecture of a system built on this idea. (Note that for some queries and updates, the only option is to reevaluate the query on the update source; thus, sometimes the translated update must access the updated source.) Using this picture, the essential

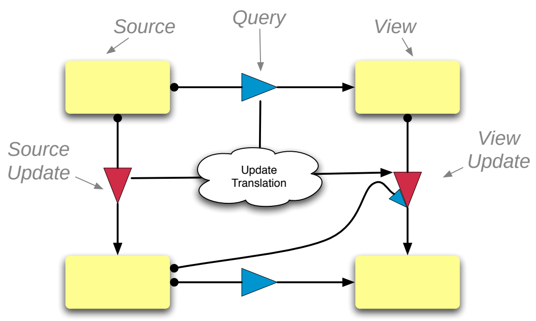

Figure 1. Correct update translation.

correctness condition for a view maintenance system can be stated as follows: the translation of an update is correct if the diagram commutes.

One reason that incremental maintenance works well in practice is that often, the effect of a source update on a view can be determined without accessing the source at all. As a concrete example, in the relational setting, if the query is a selection and the source update inserts a tuple, then the translated update either needs to insert the same tuple into the view (if it satisfies the selection predicate) or perform a no-op (if it does not). In either case, the translated update can be calculated independently of the source. Because the size of the source is usually large compared to the view, this is a huge win. Another reason that maintenance is effective is that, even when the translated update depends on the source, it can often be rewritten to only access certain relevant parts of the source. Finally, since the translation produces updates and not whole views, the problem of transmitting large views over the network is often avoided.

The problem of maintaining relational views has been well studied (see Section 7), but rather less work has focused on maintaining views of XML data. In this paper, we describe a system for maintaining views defined in the language XQuery. The main component is an update translator, which takes as inputs a source update, a query, and annotation hints generated from the source, and calculates an corresponding view update. Rather than working on queries expressed in XQuery's surface syntax, the update translator operates on the intermediate *algebraic* representation of queries used in Galax. This algebra combines operators from the relational algebra (interpreted with ordered semantics), with additional operators for manipulating and iterating over XML trees and sequences [22]. Working with the algebraic representation has several advantages. Unlike the surface syntax, which is complex and monolithic, the algebraic operators are simple, orthogonal, and composable. Simplicity streamlines the update translation function. Orthogonality exposes which operators are easy to maintain, and those that are more challenging. Compositionality makes our system easily extensible to new algebraic operators and built-in functions, and facilitates straightforward reasoning about correctness.

The main challenge in a solution to view update based on update translation is that many of the operators compute and discard intermediate data that is needed to translate updates. As an example, consider an update u and the query $\mathsf{If}(p_1)\{p_2, p_3\}$, which evaluates p_1 to obtain a sequence of items, and selects p_2 if that sequence is non-empty, or p_3 otherwise. Translating u through p_1 yields an update u_1 that applies to the sequence of items generated by p_1. The correct translation for the view is obtained either by translating u through p_2—if the sequence generated by p_1 was not empty and u_1 does not make it empty—or by translating it through p_3—if the sequence was empty and u_1 does not make it non-empty, or by discarding the view entirely and recomputing the whole query from scratch—if the update has the effect of changing which branch is selected by the conditional. Unfortunately, the information needed to distinguish these three cases—namely, the sequence produced by p_1—is not available in the view. Similar issues arising from lost intermediate data occur with several other algebraic operators.

One way to address this problem would be to cache every intermediate view. For example, the conditional operator could store both the sequence produced by p_1 and the actual view produced by p_2 or p_3. However this solution would require caching (and therefore maintaining!) a massive amount of auxiliary and potentially redundant data. Instead, we propose a sparse annotation scheme in which only fragments of these intermediate views are retained. These hints are stored in an annotation file that is provided to the translation function.

The idea of using auxiliary data to guide update translation is not new. However, a novel feature of our approach is that the amount of auxiliary data can be controlled as an external parameter to the system. With less auxiliary data, the translation function falls back to recomputation in more cases, but the annotation files are compact; with more data the update translator produces "better" updates, but the annotation files are larger.

To summarize, the contributions of this paper are as follows:

- The design of a view maintenance system for XQuery, using a translation of updates through algebraic operators.
- An adjustable annotation scheme for managing auxiliary data used during update translation.
- A prototype implementation built on Galax that handles a core set of operators and built-in functions.
- Preliminary results from experiments run on simple benchmarks queries.

In outline, the rest of the paper proceeds as follows. Sections 2 and 3 review XQuery, the tree algebra, and the update language used in our system. Section 4 describes the annotation scheme and update translation function. Sections 5 and 6 give an overview to our implementation and the results of several timing experiments. Sections 7 and 8 discusses related and future work. We conclude in Section 9.

2. XQuery Syntax and Algebra

In this section, we briefly review the XQuery language and sketch its compilation to a tree algebra. We assume familiarity with XPath, XQuery and their data model [4, 1, 9].

As described in the introduction, our system works on queries as represented in a tree algebra rather than the surface syntax of XQuery. Compiling XQuery programs to this algebra breaks down complicated features such as iteration, navigation, variable binding, selection, grouping, and reordering, expressed as monolithic FLOWR blocks, into simple operators with compositional semantics. Working with queries in this more primitive representation has many advantages, which have also been noticed by the designers of other view maintenance systems [15, 6]. First, it streamlines the update translation algorithm—it can be formulated as a recursive function on algebraic queries (and a corresponding correctness theorem can be proved by induction). Second, since the operators have compositional semantics, the algorithm can be easily extended to handle new operators just by filling in the additional cases. Lastly, since the tree algebra contains operators from the relational algebra, the relationship to previous work on view maintenance in the relational setting is clearly exposed.

The main building blocks of XQuery programs are XPath expressions, used to navigate in trees, and FLWOR blocks, used to iterate over and manipulate sequences of values. As a simple example, consider the following program, which computes a join:

```
for $x in $d/self::a/text(),
    $y in $d/self::b/text()
where $x = $y
return <c> { $x } </c>
```

Informally, the evaluation of this query goes as follows. The `for` clause iterates over the value denoted by `$d`, and binds variables `$x` and `$y` to the values obtained by navigating along the XPath expressions `$d/self::a/text()` and `$d/self::b/text()` respectively. Next, the `where` clause selects out pairs `$x` and `$y` where `$x` equals `$y`. The `return` clause constructs a new element `c` containing `$x`. The final result is the sequence composed of all such elements. For example, when this query is evaluated in a context where `$d` is bound to the sequence

```
<a>1</a><a>2</a><a>3</a><b>2</b><b>3</b><b>4</b>
```

it computes a result:

```
<c>2</c><c>3</c>
```

A naive implementation of the semantics uses many nested iterations. For this reason, most serious XQuery implementations instead compile queries to algebraic plans similar to those used in relational engines.[1] This change of perspective makes it possible to apply standard optimizations—in particular, unnestings—and has been shown to improve the efficiency of query engines by several orders of magnitude [22]. Returning to our example, the following is an equivalent algebraic plan:

$$\begin{array}{l}\mathsf{Map}\{\mathsf{Elem}[c](\#x)\}\\ \quad(\mathsf{Select}\\ \qquad\{\mathsf{eq}(\#x/\mathtt{text}(), \#y/\mathtt{text}())\}\\ \qquad(\mathsf{Product}\\ \qquad\quad(\mathsf{Map}\{[x:\mathsf{ID}]\}(\mathsf{TreeJoin}[\mathtt{self}::a](\#d)),\\ \qquad\quad\ \mathsf{Map}\{[y:\mathsf{ID}]\}(\mathsf{TreeJoin}[\mathtt{self}::b](\#d)))))\end{array}$$

The operators in this plan manipulate XML values as well as *tables* of tuples—i.e., records with fields mapping to XML values. (A type algebra and typing rules for each operator are given in Appendix A.) To illustrate the semantics, let us trace its evaluation on a table representing the same source sequence as above. Assume that the input is a table containing a single tuple whose only field d maps to this sequence. The operators at the leaves:

$$\mathsf{Map}\{[x:\mathsf{ID}]\}(\mathsf{TreeJoin}[\mathtt{self}::a](\#d))$$

$$\mathsf{Map}\{[y:\mathsf{ID}]\}(\mathsf{TreeJoin}[\mathtt{self}::b](\#d))$$

each construct a new table by accessing this sequence and applying a navigation step. Let us examine the first in detail. It accesses the sequence ($\#d$) and navigates along the self axis ($\mathsf{TreeJoin}[\mathtt{self}::a]$), which produces a sequence of elements: `<a>1</a><a>2</a><a>3</a>`. The next operator, Map, iterates over this sequence and places each item into a newly constructed tuple ($[x:\mathsf{ID}]$). During the evaluation of this Map, the identity plan (ID) represents the dependent input–i.e., the a elements. Thus, the table produced by the first plan contains tuples with a field x mapping to elements named a. Likewise, the table produced by the second plan contains tuples with a field y mapping to elements named b. Moving up a level, the next operator, Product, computes the Cartesian product of these two tables. Next, the Select operator prunes this table, retaining only those tuples with identical x and y fields. At the top of the plan, the Map iterates over this table, accesses the x field from each tuple, and constructs an element named c ($\mathsf{Elem}[c](\#x)$); these elements comprise the result sequence.

[1] In fact, some XQuery engines go a step further—they "shred" XML into relations, and translate programs to relational queries that operate on data in this encoding [2].

Figure 2 lists the algebraic operators we consider in this paper. They are sufficient for expressing many first-order XQuery programs. The full algebra used in Galax has several additional operators as well as recursive functions [22], and is rich enough to serve as a compilation target for the full XQuery 1.0 language.

Each operator, when fully applied to parameters, denotes a function of appropriate type. We use several notational conventions when writing the parameters to an operator $\mathsf{Op}[x]\{p_1\}(p_2)$. Parameters enclosed in square brackets like x are static; parameters enclosed in parentheses like p_2 are independent—i.e., do not depend on the results computed by other subplans; and parameters enclosed in curly braces like p_1 are dependent. The semantics, written $\mathcal{A}[\![\cdot]\!]_{\$t}$, is given in Figure 2. The variable $\$t$ represents the input. For simplicity, and to keep the discussion moving, we give the semantics by translation back to the familiar surface syntax. Equivalent operational [7] and denotational (by translation to Nested Relational Calculus) [11] semantics can also be defined.

The compilation from XQuery programs to algebraic plans is described in detail in previously published papers [22, 11]. We refer the reader to those papers and only sketch the compilation at a high level here. The compilation of a FLOWR-block produces a plan in which bindings and uses of variables are transformed into operations on tables. For example, `for $x in` e compiles to a plan that uses Map to construct a tuple with a single field x for each value in the sequence produced by (the compilation of) e. Likewise, a `let`-binding for `$y` compiles to a plan that uses MapConcat to extend each tuple with an additional field y. A `where`-clause compiles to a Select. Sequence constructors, element constructors, and XPath navigation all compile to the corresponding algebraic plans, and variables compile to tuple accesses.

3. Update Language

Next we describe the update language used internally in our system. To avoid complicating the translation algorithm, this language is intentionally simple: it does not contain conditionals, navigation operators, or iteration. It is rich enough, however, to express the *effect* of any update on a given data model value. Thus, although our system only manipulates these simple updates internally, it can be used to propagate source updates expressed in any formalism—e.g., XQuery! [12] or the recent draft proposal from the W3C [3]. (To use our system with these more expressive update languages, one would first evaluate the high-level update on the actual source value, obtaining a set of "atomic" updates along fixed paths, and encode these updates in our language.)

The update language is defined in Figure 3. Updates UNop, UDel, UIns, and URepl have the obvious semantics: respectively they leave the value unchanged, delete it, insert a value at the beginning of a sequence or table, and replace the entire value. Note that UIns and URepl carry algebraic plans, which are evaluated on the updated source to obtain the value to insert or to use for replacement. An update UNode optionally renames node and applies the encapsulated update to its children. A sequenced update consists of a list of updates, each indexed by an integer offset: when (i, u) appears in this list, u is used to update the ith element of the sequence (if the same offset appears twice, the updates are applied in sequence). Tuple and table updates do not apply to data model values directly but are used internally during update translation. A tuple updates consists of a finite map um from labels to updates; it applies $um(x)$ to the field x of the tuple. A table update is analogous to a sequence update.

We assume that that update lists are non-empty and that UNop and USeq do not appear immediately below other USeqs and sim-

$p ::=$	
$\mid \mathsf{ID}$	(identity)
$\mid \mathsf{Empty}()$	(empty sequence)
$\mid \mathsf{Elem}[qn](p_1)$	(element)
$\mid \mathsf{Seq}(p_1, p_2)$	(sequence)
$\mid \mathsf{TreeJoin}[s](p_1)$	(navigation)
$\mid \mathsf{If}(p_1)\{p_2, p_3\}$	(conditional)
$\mid \#x$	(tuple access)
$\mid [x : p_1]$	(tuple construction)
$\mid \mathsf{Map}\{p_1\}(p_2)$	(dependent map)
$\mid \mathsf{MapConcat}\{p_1\}(p_2)$	(concatenating map)
$\mid \mathsf{Select}\{p_1\}(p_2)$	(selection)
$\mid \mathsf{Product}(p_1, p_2)$	(product)
$ax ::= \mathtt{self} \mid \mathtt{child} \mid \mathtt{descendant}$	(axis)
$s ::= ax :: nt$	(navigation step)
$nt ::= a \mid *$	(node test)

$$\begin{aligned}
\mathcal{A}\llbracket \mathsf{ID} \rrbracket_{\$t} &= \$t \\
\mathcal{A}\llbracket \mathsf{Empty}() \rrbracket_{\$t} &= () \\
\mathcal{A}\llbracket \mathsf{Elem}[qn](p_1) \rrbracket_{\$t} &= \mathtt{element}\ qn\ \{\mathcal{A}\llbracket p_1 \rrbracket_{\$t}\} \\
\mathcal{A}\llbracket \mathsf{Seq}(p_1, p_2) \rrbracket_{\$t} &= (\mathcal{A}\llbracket p_1 \rrbracket_{\$t}, \mathcal{A}\llbracket p_2 \rrbracket_{\$t}) \\
\mathcal{A}\llbracket \mathsf{TreeJoin}[s](p_1) \rrbracket_{\$t} &= \mathtt{for}\ \$t_1\ \mathtt{in}\ \mathcal{A}\llbracket p_1 \rrbracket_{\$t}\ \mathtt{return}\ \$t_1/s \\
\mathcal{A}\llbracket \mathsf{If}(p_1)\{p_2, p_3\} \rrbracket_{\$t} &= \mathtt{if}\ (\mathcal{A}\llbracket p_1 \rrbracket_{\$t})\ \mathtt{then}\ \mathcal{A}\llbracket p_2 \rrbracket_{\$t}\ \mathtt{else}\ \mathcal{A}\llbracket p_3 \rrbracket_{\$t} \\
\mathcal{A}\llbracket \#x_i \rrbracket_{\$t} &= \$t.x_i \\
\mathcal{A}\llbracket [x : p_1] \rrbracket_{\$t} &= [x = \mathcal{A}\llbracket p_1 \rrbracket_{\$t}] \\
\mathcal{A}\llbracket \mathsf{Map}\{p_1\}(p_2) \rrbracket_{\$t} &= \mathtt{for}\ \$t_2\ \mathtt{in}\ \mathcal{A}\llbracket p_2 \rrbracket_{\$t}\ \mathtt{return}\ \mathcal{A}\llbracket p_1 \rrbracket_{\$t_2} \\
\mathcal{A}\llbracket \mathsf{MapConcat}\{p_1\}(p_2) \rrbracket_{\$t} &= \mathtt{for}\ \$t_2\ \mathtt{in}\ \mathcal{A}\llbracket p_2 \rrbracket_{\$t} \\
&\qquad \mathtt{for}\ \$t_1\ \mathtt{in}\ \mathcal{A}\llbracket p_1 \rrbracket_{\$t_2}\ \mathtt{return}\ \$t_1\mathtt{++}\$t_2 \\
\mathcal{A}\llbracket \mathsf{Select}\{p_1\}(p_2) \rrbracket_{\$t} &= \mathtt{for}\ \$t_2\ \mathtt{in}\ \mathcal{A}\llbracket p_2 \rrbracket_{\$t}\ \mathtt{return} \\
&\qquad \mathtt{if}\ (\mathcal{A}\llbracket p_1 \rrbracket_{\$t_2})\ \mathtt{then}\ \$t_2\ \mathtt{else}\ () \\
\mathcal{A}\llbracket \mathsf{Product}(p_1, p_2) \rrbracket_{\$t} &= \mathtt{for}\ \$t_1\ \mathtt{in}\ \mathcal{A}\llbracket p_1 \rrbracket_{\$t}\ \mathtt{return} \\
&\qquad \mathtt{for}\ \$t_2\ \mathtt{in}\ \mathcal{A}\llbracket p_1 \rrbracket_{\$t}\ \mathtt{return} \\
&\qquad\quad \$t_1\mathtt{++}\$t_2
\end{aligned}$$

Figure 2. XQuery algebra.

$u ::=$	
$\mid \mathsf{UNop}$	(no-op)
$\mid \mathsf{UDel}$	(deletion)
$\mid \mathsf{UIns}(p)$	(insertion)
$\mid \mathsf{URepl}(p)$	(replacement)
$\mid \mathsf{UNode}(qno, u)$	(node update)
$\mid \mathsf{USeq}(ul)$	(sequence update)
$\mid \mathsf{UTup}(um)$	(tuple update)
$\mid \mathsf{UTab}(ul)$	(table update)
$qno ::= None \mid Some\ qn$	(optional name)
$ul ::= [\,] \mid (i, u) :: ul$	(update list)
$um ::= \{\} \mid \{x \mapsto u\}\mathtt{++}um$	(update map)

$$\begin{aligned}
\mathcal{U}\llbracket \mathsf{UNop} \rrbracket_{\$t,\$x} &= () \\
\mathcal{U}\llbracket \mathsf{UIns}(p) \rrbracket_{\$t,\$x} &= \mathtt{insert\ node}\ \mathcal{A}\llbracket p \rrbracket_{\$t}\ \mathtt{before}\ \$x \\
\mathcal{U}\llbracket \mathsf{UDel} \rrbracket_{\$t,\$x} &= \mathtt{delete\ node}\ \$x \\
\mathcal{U}\llbracket \mathsf{URepl}(p) \rrbracket_{\$t,\$x} &= \mathtt{replace\ node}\ \$x\ \mathtt{with}\ \mathcal{A}\llbracket p \rrbracket_{\$t} \\
\mathcal{U}\llbracket \mathsf{UNode}(None, u) \rrbracket_{\$t,\$x} &= \mathtt{let}\ \$x' := \$x/*\ \mathtt{return}\ \mathcal{U}\llbracket u \rrbracket_{\$t,\$x'} \\
\mathcal{U}\llbracket \mathsf{UNode}(Some\ qn, u) \rrbracket_{\$t,\$x} &= (\mathtt{rename\ node}\ \$x\ \mathtt{as}\ qn, \\
&\qquad \mathtt{let}\ \$x' := \$x/*\ \mathtt{return}\ \mathcal{U}\llbracket u \rrbracket_{\$t,\$x'}) \\
\mathcal{U}\llbracket \mathsf{USeq}(ul) \rrbracket_{\$t,\$x} &= \mathtt{let}\ \$x_1 := \$x[i_1], \ldots, \$x_k := \$x[i_k]\ \mathtt{return} \\
&\qquad (\mathcal{U}\llbracket u_1 \rrbracket_{\$t,\$x_1}, \ldots, \mathcal{U}\llbracket u_k \rrbracket_{\$t,\$x_k}) \\
&\quad \text{where } ul = [(i_1, u_1), \ldots, (i_k, u_k)] \\
\mathcal{U}\llbracket \mathsf{UTup}(um) \rrbracket_{\$t,\$x} &= \mathtt{let}\ \$x_{l_1} := \$x.l_1, \ldots, \$x_{l_k} := \$x.l_k\ \mathtt{return} \\
&\qquad (\mathcal{U}\llbracket u_{l_1} \rrbracket_{\$t,\$x_1}, \ldots, \mathcal{U}\llbracket u_{l_k} \rrbracket_{\$t,\$x_k}) \\
&\quad \text{where } um = \{l_1 \mapsto u_1, \ldots, l_k \mapsto u_k\} \\
\mathcal{U}\llbracket \mathsf{UTab}(ul) \rrbracket_{\$t,\$x} &= \mathtt{let}\ \$x_1 := \$x[i_1], \ldots, \$x_k := \$x[i_k]\ \mathtt{return} \\
&\qquad (\mathcal{U}\llbracket u_1 \rrbracket_{\$t,\$x_1}, \ldots, \mathcal{U}\llbracket u_k \rrbracket_{\$t,\$x_k}) \\
&\quad \text{where } ul = [(i_1, u_1), \ldots, (i_k, u_k)]
\end{aligned}$$

Figure 3. Update language.

ilarly for update maps. These conventions can be enforced using constructors that flatten and simplify sequence, tuple, and table updates. We often define update lists using the notation $::_{i=1}^{n}(o_i, u_i)$ and $@_{i=1}^{n} l_i$. The first denotes the list $[(o_1, u_1), \ldots, (o_n, u_n)]$ obtained by consing the (o_i, u_i)s, and the second denotes the list $(l_1 @ \ldots @ l_n)$ obtained by appending the l_is.

The semantics of updates, written $\mathcal{U}\llbracket\cdot\rrbracket_{\$t,\$x}$, is defined in Figure 3. The $\$t$ parameter specifies the value to be used as the source when evaluating the algebraic plans in insertions and replacements; $\$x$ specifies the value on which the update is executed.

4. Update Translation

Now we turn to the two central pieces of our view maintenance system—the update translation algorithm, and our scheme for representing auxiliary data in annotation files. We do not discuss the maintenance of auxiliary data in this paper; it can be performed using an extension of the algorithm discussed here.

The update translation algorithm is formulated as a recursive algorithm that propagates updates from bottom to top through the tree of nested operators that make up the algebraic query plan. For some operators the translation is simple. For example, the semantics of the identity plan maps any source to itself, so every update has the same effect on the source and view. Other operators, however, compute intermediate data that is not included in the view but is needed to rewrite source updates to view updates. For example, as described in the introduction, the conditional operator selects a branch using a sequence of items that it computes and then discards the information about which branch was picked. Other examples of operators that discard intermediate data are the sequence and map operators, which concatenate several sequences into one, forgetting the positions marking the boundaries of the original sequences, and the select operator, which discards tuples that do not satisfy the selection predicate, forgetting the positions and values of the discarded tuples. To correctly propagate updates to views defined using these operators, the translation algorithm needs access to the forgotten data. For example, with the conditional operator, the algorithm needs to determine which branch was selected, and whether the source update affects that choice. For the sequence and map operators, it needs to know the boundaries of the original sequences so that it can merge updates to each sequence into a single update that applies to the concatenated sequence. For the selection operator, it needs to take an update to the original table and rewrite it to one that applies to tuples retained in the view.

One way to make this information available to the algorithm would be to cache all of the intermediate data that is computed during the evaluation of a query. However, this strategy requires storing a huge amount of redundant data. To avoid this problem we instead store only some of the intermediate information for each operator. This reduces the amount of auxiliary storage that needed, and also allows us to tune the amount and content of data that is stored in creative ways. When more auxiliary data is available, the update translator produces "better" updates, but the annotation files are large; when less auxiliary data is available, it falls back to recomputation in more cases, but the annotation files are small. As a concrete example, the auxiliary data for the conditional operator could either be the entire intermediate sequence, or just the boolean value it encodes. If we store the whole sequence, then we can compute the effect of a source update on the branch selected exactly and obtain, in some sense, an optimal translation. If we only store the boolean value, then the annotation is more compact, but the algorithm has to rely on a conservative analysis to determine the effect of the update on the intermediate sequence. In the limit, we could keep no annotation data at all. In this case, update translation falls back to recomputation in most cases, which sounds bad. However, if the conditional appears below a map operator, then the recomputation will only need to access the items in the source directly affected by the source update. Thus, keeping no annotations for some operators may be a reasonable strategy for some queries. Our approach allows programmers balance these tradeoffs.

In the remainder of this section, we describe the annotation scheme and update translation algorithm in detail. When p is a query and s is a source, we write $\mathsf{annot}(p, s)$ for the annotation computed from p and s. Additionally, when u is an update, and $x = \mathsf{annot}(p, s)$, we write $u \overset{p}{\leadsto}_x u'$ to indicate that u translates to u' (with respect to p and x).

Together, the update translation and annotation functions satisfy the following correctness theorem:

Theorem 4.1 (Correct Update Translation)**.** *Let s, s', p, u, and v, with $v = \mathcal{A}[\![p]\!]_s$, $s' = \mathcal{U}[\![u]\!]_{s,s}$, $x = \mathsf{annot}(p, s)$ and $u \overset{p}{\leadsto}_x u'$. Then $\mathcal{A}[\![p]\!]_{s'} = \mathcal{U}[\![u']\!]_{s',v}$.*

which just states formally that the diagram in Figure 1 commutes. The proof goes by induction on p.

In the remainder of this section, we give the recursive definitions of these two functions, examining each case in detail. To lighten the description of the algorithm, we leave some cases undefined and adopt the convention that we fall back to recomputation in these cases. Formally, this convention is modeled by a "catch-all" rule

$$\frac{\text{no other rule applies}}{u \overset{p}{\leadsto} \mathsf{URepl}(p)}$$

(note that the annotation is missing). In our implementation, annotations are represented as XML values and the translation function produces a URepl update when it needs some auxiliary data but the annotation is empty.

Identity The identity operator, $p = \mathsf{ID}$, maps every source to itself. Since updates affect the source and view in exactly the same way, they are translated exactly. No auxiliary data is needed.

$$\mathsf{annot}(p, s) = () \qquad\qquad u \overset{p}{\leadsto} u$$

Empty Sequence The operator $p = \mathsf{Empty}()$ maps every source to the empty sequence. Since the semantics is a constant function, source updates do not affect the view. Accordingly, updates are translated to no-ops. Again, no auxiliary data is needed.

$$\mathsf{annot}(p, s) = () \qquad\qquad u \overset{p}{\leadsto} \mathsf{UNop}$$

Element Constructor The operator $p = \mathsf{Elem}[qn](p_1)$ constructs an element node with name qn and children obtained by p_1. As with the empty operator, part of the view—the name—is constant, so a source update can only affect the children of the view. Source updates are recursively translated through p_1, and wrapped a UNode update that leaves the name unchanged. No additional auxiliary data is needed; the annotation just records $\mathsf{annot}(p_1, s)$:

$$\mathsf{annot}(p, s) = \mathsf{annot}(p_1, s) \qquad\qquad \frac{u \overset{p_1}{\leadsto}_x u_1}{u \overset{p}{\leadsto}_x \mathsf{UNode}(None, u_1)}$$

Sequence Constructor The operator for constructing sequences $p = \mathsf{Seq}(p_1, p_2)$ applies the subplans p_1 and p_2 to the source, yielding two sequences, and then concatenates (and flattens) these into a single sequence. Recursively translating a source update through p_1 and p_2 yields updates that apply to the original pair of sequences. To finish the job, we need to rewrite these updates so that they apply to the appropriate portions of the concatenated sequence. The annotation for a sequence stores the lengths n_1 and n_2 of the original sequences needed to do this rewriting (it also stores the annotations x_1 and x_2 generated for p_1 and p_2 from s):

$$\mathsf{annot}(p, s) = \begin{array}{l} \texttt{<Seq>} \\ \quad \texttt{<p1>}x_1\texttt{</p1><p2>}x_2\texttt{</p2>} \\ \quad \texttt{<ns>}n_1\ n_2\texttt{</ns>} \\ \texttt{</Seq>} \end{array}$$

The update translation rule uses a helper function flatten that takes an update u, an offset o, and a length n, and calculates an update list—i.e., of pairs of indices and updates—that, when wrapped in a USeq update, describes the update that applies u to the n items from position o. The definition of flatten is as follows (the helper function $\mathsf{mk\_list}(o, n, u)$ constructs an list where u is paired with every index from o to $o + n$ inclusive)

$$\begin{array}{ll}
\mathsf{flatten}(o, n, \mathsf{UNop}) & = [] \\
\mathsf{flatten}(o, n, \mathsf{UIns}(p_1)) & = [(o, \mathsf{UIns}(p_1)] \\
\mathsf{flatten}(o, n, \mathsf{URepl}(p_1)) & = \\
\quad (o, \mathsf{URepl}(p_1)) :: (\mathsf{mk\_list}(o+1, n-1, \mathsf{UDel})) & \\
\mathsf{flatten}(o, n, \mathsf{UDel}) & = \mathsf{mk\_list}(o, n, \mathsf{UDel}) \\
\mathsf{flatten}(o, \_, \mathsf{UNode}(qno, u_{11}) & = [(o, 1, \mathsf{UNode}(qno, u_{11}))] \\
\mathsf{flatten}(o, \_, \mathsf{USeq}(ul) & = ::_{j=1}^{|ul|} (o'_j, u'_j) \\
\quad \text{where } o'_j = o_j + o \text{ and } u'_j = u_j &
\end{array}$$

The interesting cases are URepl, which produces a list of indexed updates whose head is the replacement and tail contains $n - 1$ deletions; UNode, which can only be validly applied to a sequence of length one, and is therefore flattened to a singleton list; and USeq, which shifts the index of each member of its update list by o. Using flatten, the update translation rule for sequences is:

$$\frac{\begin{array}{cc} u \overset{p_1}{\leadsto}_{x_1} u_1 & l_1 = \mathsf{flatten}(1, n_1, u_1) \\ u \overset{p_2}{\leadsto}_{x_2} u_2 & l_2 = \mathsf{flatten}(n_1 + 1, n_2, u_2) \end{array}}{u \overset{p}{\leadsto}_x \mathsf{USeq}(l_1 @ l_2)}$$

Updates translated using this rule have the effect stated above: u_1 is applied to the first n_1 items in the view, and u_2 to the subsequent n_2 items. (To keep the presentation of the rules simple, we access annotations such as the x_is and n_is by name instead of navigating to them from x using an XPath expressions.)

In our implementation, we handle some other cases as optimizations. For example, the rule

$$\frac{u \overset{p_1}{\leadsto}_{x_1} \mathsf{URepl}(p'_1) \qquad u \overset{p_2}{\leadsto}_{x_2} \mathsf{URepl}(p'_2)}{u \overset{p}{\leadsto}_x \mathsf{URepl}(\mathsf{Seq}(p'_1, p'_2))}$$

calculates an equivalent, but more compact update in the case where u_1 and u_2 are both replacements.

Navigation The navigation operator $p = \mathsf{TreeJoin}[ax :: nt](p_1)$ first maps the source to a sequence using p_1, and then returns the sequence obtained by retaining the items along the paths specified by the navigation step $ax :: nt$. In this section we focus on the `child` axis; translations for other axes are discussed below. Intuitively, maintaining a view defined by navigating in this way is simple: first calculate the update to the intermediate sequence obtained from p_1, then symbolically interpret the navigation step on the update. In practice, however, this second step requires precise information about the paths in the intermediate sequence that produced items in the result. We store this data in the annotation. Let $(e_1, \ldots, e_k) = \mathcal{A}[\![p_1]\!]_s$. Define $n_{ij} = 1$ if the jth child of e_i is included in the view and 0 otherwise for every such i and j. Also, let $x_1 = \mathsf{annot}(p_1, s)$. The annotation is as follows:

$$\mathsf{annot}(p, s) = \begin{array}{l}\texttt{<TreeJoin>}\\ \quad\texttt{<p1>}x_1\texttt{</p1>}\\ \quad\texttt{<ns>}n_{11}\ldots n_{kl}\texttt{</ns>}\\ \texttt{</TreeJoin>}\end{array}$$

To shorten the description below, we abbreviate the total number of items in the view obtained from e_i as $t_i = \sum_j n_{ij}$.

The update translation rule uses two helper functions. The first, rwkid, takes as arguments i, t, and u, which, when invoked from the other helper function rw, represent the index of an item e_i, the count t_i, and an update u that applies to the children of e_i. It rewrites u to an update that just applies to the children in the view.

$$\begin{array}{ll}
\mathsf{rwkid}(\_, \_, \mathsf{UNop}) & = \mathsf{UNop}\\
\mathsf{rwkid}(\_, \_, \mathsf{UIns}(p_1')) & = \mathsf{UIns}(\mathsf{TreeJoin}[\texttt{self}::nt](p_1'))\\
\mathsf{rwkid}(\_, t, \mathsf{URepl}(p_1')) & = \\
\quad \begin{cases}\mathsf{UIns}(\mathsf{TreeJoin}[\texttt{self}::nt](p_1')) & \text{if } t = 0\\ \mathsf{URepl}(\mathsf{TreeJoin}[\texttt{self}::nt](p_1')) & \text{if } t > 0\end{cases} & \\
\mathsf{rwkid}(\_, t, \mathsf{UNode}(qno, u_1)) = & \\
\quad \begin{cases}\mathsf{UNop} & \text{if } (t > 0 \wedge qno = \mathit{None})\\ & \vee\, (t > 0 \wedge qno = \mathit{Some}\ qn \wedge qn \models nt)\\ & \vee\, (t = 0 \wedge qno = \mathit{Some}\ qn \wedge qn \not\models nt)\\ \mathsf{UDel} & \text{if } (t > 0 \wedge qno = \mathit{Some}\ qn \wedge qn \not\models nt)\end{cases} & \\
\mathsf{rwkid}(i, \_, \mathsf{USeq}(ul)) & = \mathsf{USeq}(::_{j=1}^{|ul|}(o_j', u_j'))\\
\quad \text{where } o_j' = 1 + \sum_{k=1}^{j-1} t_k \text{ and } u_j' = \mathsf{rwkid}(i, n_{ij}, u_j) &
\end{array}$$

The notation $qn \models nt$ indicates that qn satisfies the condition expressed by nt. It is shorthand for $nt = *$ or $nt = qn$.

Let us examine several of the cases in detail. Insertions are rewritten using navigation along the `self` axis. This ensures that the values satisfy the condition expressed by the node test. Replacements use an analogous rewriting; additionally, when t is 0, then the old child was not contained in the view, so the replacement is actually an insertion. For UNode updates that do not change the condition expressed by the node test, the effect on the view is a no-op. Otherwise, if $t > 0$ and the UNode renames the element to one that does not satisfy the node test, then the child is deleted. Note that the case for $t = 0$ and $qno = \mathit{Some}\ qn$ and $qn \models nt$ is not defined. This represents the situation where an item that was previously omitted needs to be inserted into the view. The inserted item could be obtained by applying the encapsulated update in UNop to the omitted item if it were available. Unfortunately, because our annotation is sparse, it is not. Thus, we leave the case undefined and (by the convention introduced previously) fall back to recomputation. An annotation scheme that cached all the children could handle this case better, at the cost of larger annotation files. The final case for rwkid handles sequenced updates: it applies rwkid to the positions mentioned in its update list, and uses the n_{ij}s to track the offsets of children in the view.

The second helper, rw, translates the update calculated for the sequence returned by p_1 to a corresponding view update. It takes as arguments the index of an item e_i and the source update.

$$\begin{array}{lcl}
\mathsf{rw}(\_, \mathsf{UNop}) & = & \mathsf{UNop}\\
\mathsf{rw}(\_, \mathsf{UIns}(p_1')) & = & \mathsf{UIns}(\mathsf{TreeJoin}[\texttt{child}::nt](p_1'))\\
\mathsf{rw}(\_, \mathsf{URepl}(p_1')) & = & \mathsf{URepl}(\mathsf{TreeJoin}[\texttt{child}::nt](p_1'))\\
\mathsf{rw}(\_, \mathsf{UDel}) & = & \mathsf{UDel}\\
\mathsf{rw}(i, \mathsf{UNode}(\_, u_{11})) & = & \mathsf{rwkid}(i, t_i, u_{11})\\
\mathsf{rw}(\_, \mathsf{USeq}(ul)) & = & \mathsf{USeq}(::_j^{|ul|}(o_j', u_j'))\\
\multicolumn{3}{l}{\quad \text{where } o_j' = 1 + \sum_{k=1}^{j-1} t_k \text{ and } u_j' = \mathsf{rw}(j, u_j)}
\end{array}$$

The important cases are UNode, which invokes rwkid on the update to the children, and USeq, which handles the bookkeeping needed to rewrite the indices on updates using the counts from the annotation data. The final translation of updates is as follows:

$$\frac{u \overset{p_1}{\rightsquigarrow}_{x_1} u_1 \qquad \mathsf{rw}(1, u_1) = u'}{u \overset{p}{\rightsquigarrow}_{x} u'}$$

These rules handle updates to views defined by navigation along the `child` axis. The `self` axis can be handled similarly (in fact, the rules are simpler, since the navigation is at the same level). The `descendant` axis, however, is more complicated—the view consists of all the items matching the node test at any depth in the tree, in document order. One option is to generalize the annotations and rwkid and rw, storing auxiliary data about *every* descendant. However, this approach is very complicated and produces huge annotations. Section 8 discusses an alternative approach for `descendant` that we believe has promise.

Conditional The conditional operator $\mathsf{If}(p_1)\{p_2, p_3\}$ evaluates the subplan p_1 to select p_2 or p_3, and then evaluates that branch. As discussed in previous sections, we have some freedom in the amount of annotation data that is stored for conditional. Here we discuss a scheme that stores three pieces of data: the annotation $x_1 = \mathsf{annot}(p_1, s)$, an annotation x_b, which is either $\mathsf{annot}(p_2, s)$ if p_2 was selected or $\mathsf{annot}(p_3, s)$ otherwise, and the length n of the sequence computed by p_1.

$$\mathsf{annot}(p, s) = \begin{array}{l}\texttt{<If>}\\ \quad\texttt{<p1>}x_1\texttt{</p1><pb>}x_b\texttt{</pb>}\\ \quad\texttt{<ns>}n\texttt{</ns>}\\ \texttt{</If>}\end{array}$$

The update translation rule uses a conservative static analysis to determine whether the source update affects the selection of a branch. We formulate this analysis using several auxiliary predicates. The predicate $\mathsf{pre}_\top(u)$ holds when u can be statically determined to preserve non-emptiness. For example, $\mathsf{pre}_\top(\mathsf{UIns}(p))$ holds since inserting any value into a non-empty sequence yields a non-empty sequence. Similarly, the predicate $\mathsf{chg}_\top(u)$ holds when u can be statically determined to change an empty sequence into a non-empty one. The predicates $\mathsf{pre}_\bot(u)$ and $\mathsf{chg}_\bot(u)$ are dual. Finally, predicates $\mathsf{empty}(p)$ and $\mathsf{nonempty}(p)$ are true of algebraic plans that can be statically determined to produce the empty or non-empty sequences respectively. We give definitions for $\mathsf{pre}_\top$ only; the others are similar:

$$\mathsf{pre}_\top(\mathsf{UNop}) \qquad \mathsf{pre}_\top(\mathsf{UIns}(\_)) \qquad \frac{\mathsf{nonempty}(p)}{\mathsf{pre}_\top(\mathsf{URepl}(p))}$$

$$\mathsf{pre}_\top(\mathsf{UNode}(\_, \_)) \qquad \frac{\mathsf{pre}_\top(u_i) \qquad (o_i, u_i) \in ul}{\mathsf{pre}_\top(\mathsf{USeq}(ul))}$$

Using these predicates, the translation of an update through a conditional is defined by the several rules. We give the rules where the annotation data n satisfies $n > 0$; the case for $n = 0$ is similar.

$$\frac{u \overset{p_1}{\rightsquigarrow}_{x_1} u_1 \qquad n > 0 \qquad \mathsf{pre}_\top(u_1) \qquad u \overset{p_2}{\rightsquigarrow}_{x_b} u'}{u \overset{p}{\rightsquigarrow}_x u'}$$

$$\frac{u \overset{p_1}{\rightsquigarrow}_{x_1} u_1 \qquad n > 0 \qquad \mathrm{chg}_\bot(u_1)}{u \overset{p}{\rightsquigarrow}_x \mathsf{URepl}(p_3)}$$

Note that the static analyses are conservative, so when $n > 0$ and neither $\mathsf{pre}_\top(u_1)$ nor $\mathrm{chg}_\bot(u_1)$, by convention the algorithm falls back to recomputation.

Tuple Access A tuple access $p = \#x$ returns the sequence of items obtained by projecting x from each tuple of the input table. As with the sequence operator, the lengths of the sequence produced by each tuple are needed to rewrite an update to the input table to one that operates on the appropriate parts of the view. Let n_i be the length of every such sequence. The annotation is as follows:

$$\mathsf{annot}(p, s) = \texttt{<Access>}n_1 \dots n_k\texttt{</Access>}$$

The translation of source updates uses a helper function to rewrite update, which we again call rw.

$$\begin{array}{lcl} \mathsf{rw}(\mathsf{UNop}) & = & \mathsf{UNop} \\ \mathsf{rw}(\mathsf{UIns}(p_1')) & = & \mathsf{UIns}(\mathsf{Map}\{\#x\}(p_1')) \\ \mathsf{rw}(\mathsf{URepl}(p_1')) & = & \mathsf{URepl}(\mathsf{Map}\{\#x\}(p_1')) \\ \mathsf{rw}(\mathsf{UDel}) & = & \mathsf{UDel} \\ \mathsf{rw}(\mathsf{UTup}(um)) & = & um(x) \\ \mathsf{rw}(\mathsf{UTab}(ul)) & = & \mathsf{UTab}(@_j^{|ul|} l_j) \\ \multicolumn{3}{l}{\quad \text{where } l_j = \mathsf{flatten}(1 + \sum_{i=1}^{j-1} n_i, n_j, \mathsf{rw}(u_j))} \end{array}$$

The interesting cases are UTup, which accesses the x field from the tuple map um and UTab, which rewrites table updates using the n_is and the helper function flatten, defined previously, to apply each update to the correct portion of the view. The update translation rule just invokes rw:

$$\frac{\mathsf{rw}(u) = u'}{u \overset{p}{\rightsquigarrow}_x u'}$$

Tuple Constructor The operator $p = [x : p_1]$ constructs a tuple with a single field x leading to the value obtained by p_1. Updates are recursively translated through p_1, placed in a finite map, and wrapped in a UTup constructor. The annotation only stores the annotation for the subplan:

$$\mathsf{annot}(p, s) = \mathsf{annot}(p_1, s) \qquad \frac{u \overset{p_1}{\rightsquigarrow}_x u_1}{u \overset{p}{\rightsquigarrow}_x \mathsf{UTup}(\{x \mapsto u_1\})}$$

Maps A mapping operator, such as $p = \mathsf{Map}\{p_1\}(p_2)$, expresses iteration. We will focus on the simpler Map operator in the case where p_2 produces a table and p_1 transforms each tuple into a sequence of items; the rules for Map at other types, as well as the MapConcat operator are similar.

Intuitively, the benefits of maintaining views versus recomputing them should be especially evident with maps—when the source update only affects a few tuples in the table, then only a few items in the view will need to be updated. This intuition is essentially correct, although some bookkeeping is needed to determine the parts of the view to update. The Map operator first evaluates p_2 to obtain a table, then iterates over this table, applying p_1 to each tuple and concatenating the resulting items into a single sequence. As with sequences, the annotation for a Map needs to store the lengths of the sequences computed for each tuple. Let $x_2 = \mathsf{annot}(p_2, s)$ and let $(t_1, \dots, t_k) = \mathcal{A}[\![p_2]\!]_s$ be the table computed by p_1. Also let $x_{1i} = \mathsf{annot}(p_1, t_i)$ be the annotation for p_1, as computed on the input t_i, and n_i be the length of the sequence $\mathcal{A}[\![p_1]\!]_{t_i}$. Then the annotation for Map is the following.

$$\mathsf{annot}(p, s) = \begin{array}{l} \texttt{<Map>} \\ \quad \texttt{<p2>}x_2\texttt{</p2>} \\ \quad \texttt{<p1>}x_{11}\texttt{</p1>} \dots \texttt{<p1>}x_{1l}\texttt{</p1>} \\ \quad \texttt{<ns>}n_1 \dots n_k\texttt{</ns>} \\ \texttt{</Map>} \end{array}$$

Note that there are k annotations for p_1, one for each tuple. The update translation rule uses a helper function, also named rw. It takes arguments i, the index of a tuple t_i, p_i a plan that computes the t_i, and an update u, and computes a view update that applies to the items in the view affected by u:

$$\begin{array}{lcl} \mathsf{rw}(\_, \_, \mathsf{UNop}) & = & \mathsf{UNop} \\ \mathsf{rw}(\_, p_i, \mathsf{UIns}(p_2')) & = & \mathsf{UIns}(p_2'[p_i/\mathsf{ID}]) \\ \mathsf{rw}(\_, p_i, \mathsf{URepl}(p_2')) & = & \mathsf{URepl}(p_2'[p_i/\mathsf{ID}]) \\ \mathsf{rw}(\_, \_, \mathsf{UDel}) & = & \mathsf{UDel} \\ \mathsf{rw}(i, p_i, \mathsf{UTup}(um)) & = & u'[p_i/\mathsf{ID}] \\ \multicolumn{3}{l}{\quad \text{where } \mathsf{UTup}(um) \overset{p_1}{\rightsquigarrow}_{x_{1i}} u'} \\ \mathsf{rw}(\_, p_i, \mathsf{UTab}(ul)) & = & \mathsf{UTab}(@_j^{|ul|} l_j) \\ \multicolumn{3}{l}{\quad \text{where } l_j = \mathsf{flatten}(1 + \sum_{k=1}^{j-1} n_k, n_j, \mathsf{rw}(j, p_i[j], u_j))} \end{array}$$

The case for UTup rewrites u to u' using p_1. This yields an update that applies to the view. However, if u' contains replacements or insertions, then the inputs to those plans needs to be replaced with the portion of the source that produced the tuple. This is accomplished by substituting p_i for the input ID. The other interesting case is for UTab. It first rewrites each sequenced update using a recursive call to rw, passing $p_i[j]$—a plan that generates the jth tuple. In this way, even if the translation of the update through p_1 triggers a recomputation, it is limited to only a part of the input. To finish the case, it then rewrites the update to apply to the appropriate portion of the view using flatten. Using rw, the translation is as follows:

$$\frac{u \overset{p_2}{\rightsquigarrow}_{x_2} u_2 \qquad \mathsf{rw}(1, p_2, u_2) = u'}{u \overset{p}{\rightsquigarrow}_x u'}$$

Relational Operators The translations of updates through the operators Select and Project are similar to ones developed for relational data models. However, the updates need to additionally respect the order of tuples in the view. To illustrate how we handle this new challenge, we give the update translation rules for Select; Project is a similar generalization to ordered data.

The operator $p = \mathsf{Select}\{p_1\}(p_2)$ first evaluates p_2 on the source to obtain a table, and then applies p_1 to each tuple in the table, retaining only those tuples that produce a non-empty sequence of values. The main challenge in maintaining these views is that the intermediate table is not available. Thus, when an update is recursively translated through p_2, it is not possible to determine which tuples in the are affected by the update it expresses. Let $x_2 = \mathsf{annot}(p_2, s)$ and let $(t_1, \dots, t_k) = \mathcal{A}[\![p_2]\!]_s$ be the table computed by p_1. Also let $x_{1i} = \mathsf{annot}(p_1, t_i)$ be the annotation for p_1, as computed on the input t_i, and $n_i = 1$ if $\mathcal{A}[\![p_1]\!]_{t_i}$ is non-empty, and $n_i = 0$ otherwise. The annotation for Select is the following.

$$\mathsf{annot}(p, s) = \begin{array}{l} \texttt{<Select>} \\ \quad \texttt{<p2>}x_2\texttt{</p2>} \\ \quad \texttt{<p1>}x_{11}\texttt{</p1>} \dots \texttt{<p1>}x_{1k}\texttt{</p1>} \\ \quad \texttt{<ns>}n_1 \dots n_k\texttt{</ns>} \\ \texttt{</Select>} \end{array}$$

Updates are translated using a helper function rw which takes as arguments the index i of a t_i, p_i a plan that produces that tuple, and an update. It is defined as follows.

$$
\begin{array}{lcl}
\mathsf{rw}(\_, \_, \mathsf{UNop}) & = & \mathsf{UNop} \\
\mathsf{rw}(\_, \_, \mathsf{UIns}(p_2')) & = & \mathsf{UIns}(\mathsf{Select}\{p_1\}(p_2')) \\
\mathsf{rw}(\_, \_, \mathsf{URepl}(p_2')) & = & \mathsf{URepl}(\mathsf{Select}\{p_1\}(p_2')) \\
\mathsf{rw}(\_, \_, \mathsf{UDel}) & = & \mathsf{UDel} \\
\mathsf{rw}(i, p_i, \mathsf{UTup}(um)) & = & \\
\multicolumn{3}{l}{\quad \begin{cases} \mathsf{UNop} & \text{if } n_i = 1 \wedge \mathsf{pre}_\top(u') \vee n_i = 0 \wedge \mathsf{pre}_\bot(u') \\ \mathsf{UDel} & \text{if } n_i = 1 \wedge \mathsf{chg}_\bot(u') \\ \mathsf{URepl}(p') & \text{otherwise with } p' = \mathsf{Select}\{p_1\}(p_i) \end{cases}} \\
\multicolumn{3}{l}{\quad \text{where } \mathsf{UTup}(um) \overset{p_1}{\rightsquigarrow}_{x_{1i}} u'} \\
\mathsf{rw}(\_, p_i, \mathsf{UTab}(ul)) & = & \mathsf{UTab}(::_j^{|ul|}(o_j', u_j')) \\
\multicolumn{3}{l}{\quad \text{where } o_j' = 1 + \sum_{k=1}^{j-1} n_k \text{ and } u_j' = \mathsf{rw}(j, p_i[j], u_j)}
\end{array}
$$

The UTup case has several interesting subcases. First, if the tuple is in the view and the update u' obtained by translating through p_1 preserves the non-emptiness of the sequence computed by p_1, or if the tuple was not in the view and u' changes the sequence to empty, then the update is translated to a no-op. The second subcase handles situations where the tuple was in the view and the update removes it. In the third case, since the annotation does not contain the tuples that were not included in the view, the translated update is a replacement. However, we calculate the replacement using p_i, will often be a smaller table than p_2.

Using rw, the update translation rule is the following:

$$
\frac{u \overset{p_2}{\rightsquigarrow}_{x_2} u_2 \qquad \mathsf{rw}(1, p_2, u_2) = u'}{u \overset{p}{\rightsquigarrow}_{x} u'}
$$

5. Implementation

To test these ideas, we have implemented a prototype system as an extension of the Galax engine. It consists of approximately 2,500 lines of OCaml code, and has functionality spread across three modules: an update compiler, a query instrumentor, and the update translator itself.

Update Compiler The compiler translates expressions in the update language to query plans that can be executed in Galax. It implements the semantics defined in Figure 3.

Query Instrumentor The instrumentor takes a source query and rewrites it to one that computes the auxiliary data needed during update translation. All of the auxiliary data that goes into the annotation files is known during the initial evaluation of the query. Thus, in principle, one could replace the instrumentor with a modified engine that calculates annotations "for free" as it evaluates the view. Doing so however, would require deep changes to the back-end—i.e., to the implementations of the physical operators. To avoid these complications, we use a simpler approach: the instrumentor rewrites algebraic plans to ones that, when applied to the source, calculate the auxiliary data instead of the view. To make it easy to access this data, we represent the annotations as XML (a serious implementation would use a more compact representation.)

The instrumentor uses several simple optimizations to reduce the sizes of annotation files. For example, with a Map operator that iterates over a sequence and constructs a single tuple from each item, the annotation storing the number of tuples produced by each iteration is not needed. Additionally, as discussed in previous sections, the amount and content of annotation files can be controlled by the user of the view maintenance system. Concretely, these controls are specified as parameters to the query instrumentor. We provide two mechanisms for controlling the size of annotation files. First, it is possible to limit the amount of auxiliary data generated by only constructing the annotations for operators up to a fixed depth in the tree of nested operators. Second, the content of the annotations for several operators can be controlled individually. For example, the conditional operator can either cache the entire sequence produced by its first plan, the length of this sequence, or nothing at all. The update translator is engineered to use whatever annotation data is available in the file, and to gracefully fall back to recomputation when the auxiliary data is missing.

Update Translator The final component of our system is the update translator itself. It is formulated as a simple, recursive function that traverses the query and propagates updates from bottom to top. In addition to the core set of operators described in this paper, the implementation handles some built-in functions that appear in the algebraic query plans produced by the Galax compiler.

6. Experiments

Using our prototype implementation, we have run timing experiments on some simple queries to test its performance. We use queries form the XMark suite [24], which includes a collection of "typical" XQuery programs and a utility for generating XML documents of varying size populated with pseudo-random values.

We ran the experiments on a 1.4GHz Intel Pentium III machine equipped with 2GB of memory and running the SuSE operating system with Linux kernel version 2.6.18. We ran each experiment five times on inputs varying in size from a few dozen kilobytes up to a few dozen megabytes and calculated an average time by discarding the shortest and longest time and computing the arithmetic mean of the remaining values. We collected the wall-clock times using POSIX system calls.

For each experiment, we measured the time needed to sequentially update the source and then recompute the query as well as the time to translate and apply the update on the view. To simulate an online view maintenance system, in which the source and view are kept in memory, we pre-loaded all of the documents and only counted time spent actually translating and applying updates or evaluating queries. Thus, our experiments did not directly measure the time needed to calculate the annotations or materialize any of the structures.

The first experiment uses the XMark Q1 query, which is an XPath expression that selects out a single item from an XML document that represents data for an online auction site. We applied an update that modified a portion of the document along a different path than the one used for the query. Thus, the update was irrelevant to the view. Irrelevant updates are a simple case, but they are common (e.g., in the online auction site updates to the source posted by other clients will usually not affect the web page for the item being viewed) and it is critical that they be detected. Using the annotations—in particular the numeric counts stored for the TreeJoin operator—our implementation correctly detects that the update is irrelevant and translates the source update to a no-op.

The second experiment uses the XMark Q5 query, which selects values from closed auctions where the selling price is greater than 40.00. For the source update, we deleted the element representing the first closed auction. Thus, the view update will either be a no-op, if the price of the deleted element was less than or equal to 40.00, or a delete otherwise. The pseudo-random data tests both cases, and our implementation correctly produces both updates.

The third experiment also uses the Q5 query, but updates the source by inserting a value instead. In this case, the translated update recomputes most of the view—only a few nodes at the top of the view are maintained. We included this experiment to measure the overhead in a case where the update replaces most of the view.

The results of these experiments are given in Figure 4 and the following table:

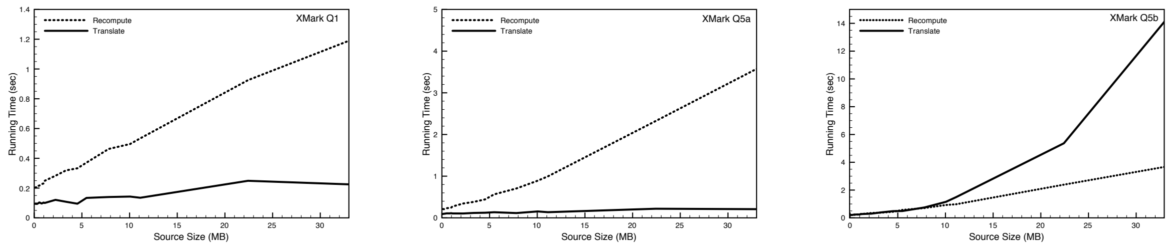

Figure 4. Experimental results.

Src(MB)	Recomp(sec)			Trans(sec)			Annot(kB)	
	Q1	Q5a	Q5b	Q1	Q5a	Q5b	Q1	Q5
.1	0.2	0.2	0.2	0.1	0.1	0.2	2	4
.5	0.2	0.2	0.2	0.1	0.1	0.2	11	18
1	0.2	0.2	0.3	0.1	0.1	0.3	21	39
10	0.4	0.8	0.9	0.1	0.2	1.1	190	329
22	0.9	2.3	2.4	0.2	0.2	5.3	419.1	729
33	1.2	3.6	3.7	0.2	0.2	14	628.2	1091

As they show, an approach based on update translation can achieve huge performance gains over the naive maintenance strategy, using a relatively simple annotation scheme, but there is some overhead when the view must be recomputed. We believe that the steep curve for translation in the third experiment results from limitations in the physical representation of data model values—in its current version, our tool stores the original source and view and updated source and view in memory simultaneously.

These preliminary experiments only scratch the surface. In the future, we hope to design a more comprehensive evaluation by measuring the performance of our system on complex queries, with only partial auxiliary data, and in settings where the source and view live on different hosts.

7. Related Work

View maintenance has been studied in a variety of settings. Early work on maintaining materialized relational views focused on techniques for detecting irrelevant updates and algorithms for propagating "deltas"—simple transactions consisting of insertions and deletions of tuples—from source to view. The survey article and collection edited by Gupta and Mumick describe this early work [16, 17]. They also developed algorithms for recursive views [18]. Qian and Wiederhold developed an algorithm that works on algebraic queries, like the approach used in this work [21]. This algorithm was later corrected by Griffin, Libkin, and Trickey, and extended to bags and deferred maintenance [15, 14, 5].

Early results on maintenance of views over graph-structured data was described by Zhuge and Garcia-Molina [27]. They observed that auxiliary data can be used to improve update propagation. Suciu showed how query decomposition can be used to maintain views over graph- and tree-structured data [25]. The maintenance of views over semi-structured data was studied by Liefke and Davidson [19]. They worked with an unordered data model and a restricted query language in which distributivity of queries over updates holds. This restriction simplifies update translation, but limits the expressiveness of the query language. In particular, queries must be monotonic with respect to updates. Sawires et al. developed maintenance techniques for views specified as arbitrary XPath expressions [23]. Their system also uses annotations, but the size of the annotation is bounded by the size of the query and the view. It operates in two phases, first identifying portions of the tree directly affected by an update, and then calculating the nodes affected indirectly. Villard et al. and Onizuka et al. each describe maintenance systems for insertions and deletions into views specified in XSLT [26, 20] using an analysis of path expressions.

The closest related work to our system is the view maintenance system for XQuery developed for the Rainbow system by Rudensteiner et al. [8, 6]. Like this work, they translate updates through operators in a tree algebra, XAT, using auxiliary data as needed. The first version of their system cached all intermediate data and did not handle ordering. In subsequent work, however, they showed how to extend the basic system to handle ordering using a clever labeling scheme to encode node identity. There are several key differences between their system and ours. First, while the labeling scheme simplifies the translation rules for some operators by identifying the value affected by an update, it is not a panacea. Operators that change the order of items require additional annotations—e.g., the sequence operator, which can place items in the view in arbitrary order, has to store additional labels tracking the "overriding order". Thus, much of the same data about positions that is cached in our simple annotation scheme is ultimately tracked in their labeling scheme as well. Second, the maintenance scheme uses node identities. This works well when such identities are available. However, in systems where the source and view live on different hosts, this means that the identities need to be transmitted to the view. By contrast, our updates only assume a functional data model, with no additional metadata. Third, unlike our system which allows annotations to be tuned and selectively omitted, auxiliary data in their system cannot be omitted—the semantics of the evaluation and maintenance engines depend on it being present. Fourth, replacements and insertions in their update language carries data model values, not algebraic plans. This means that as an update is propagated through their system, the source must be immediately queried on every replacement. In ours, a URepl update carries an algebraic plan, which can be rewritten as it percolates up through the tree. Lastly, although XAT has similar expressive power to the tree algebra considered here, only the latter was designed with completeness in mind and it is being used as the compilation target for a reference implementation of XQuery 1.0.

8. Current and Future Work

Our work is ongoing. Most of our current efforts are focused on extending our prototype to handle a more complete set of operators. The biggest challenge in this area is developing annotation schemes and translation rules for the large number of built-in functions. However, since our system is compositional, extending the system only requires adding cases for the new operators. We are also exploring new alternatives for tuning annotation files.

There are also several areas where we would like to focus our efforts in the future. The first area is query rewritings. In most systems, rewritings are used to make query evaluation go fast. In view maintenance systems, the total cost of maintaining the view often far outweighs the initial cost of evaluating the query. We plan to explore query rewritings motivated with maintainability in mind. Second, our system currently only handles first-order queries; it would be interesting to investigate maintenance for recursive XQuery views. Third, we would like to explore extending our update language to carry metadata describing the values they apply to. For example, a node update could carry the name of the node that it applies to. This would simplify the maintenance of navigation operators, since the symbolic interpretation of a navigation step could be performed without fewer annotations (although propagating this metadata also complicates other rules.) A related idea is to investigate applications of provenance metadata to view maintenance [13, 10]. Fourth, we would like to optimize the queries that our update translator produces. Our update translator often produces queries of the form $p[i]$, which access the ith item from a sequence in the source (e.g., in Map), but the current implementation does not exploit this fact to streamline access to the source.

9. Conclusions

We have described a view maintenance system for XQuery. The system translates source updates through queries expressed as algebraic operators using auxiliary data as needed to guide the translation. Our approach is fully compositional and therefore easily extensible to new operators. Moreover, the amount of auxiliary data can be tuned by users. We have implemented a prototype and run experiments to confirm that our approach outperforms naive maintenance on simple examples.

Acknowledgements The authors wish to thank the anonymous referees and Dimitrios Vytiniotis for many helpful comments. This work was performed during Nathan Foster's summer internship.

References

[1] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C, Jan. 2007. Available from `http://www.w3.org/TR/xquery`.

[2] P. A. 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 *SIGMOD Conference*, Chicago, IL, USA, June 2006.

[3] D. Chamberlin, D. Florescu, and J. Robie. *XQuery Update Facility*. W3C, July 2006. Available from `http://www.w3.org/TR/xqupdate`.

[4] J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C, Nov. 1999. Available from `http://www.w3.org/TR/xpath/`.

[5] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD Conference*, pages 469–480, 1996.

[6] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *International Conference on Conceptual Modeling (ER), Chicago, IL*, volume 2813 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 2003.

[7] 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*. W3C, Jan. 2007.

[8] M. El-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An algebraic approach for incremental maintenance of materialized XQuery views. In *International Workshop on Web Information and Data Management (WIDM), McLean, VA*, pages 88–91, 2002.

[9] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*. W3C, Jan. 2007. Available from `http://www.w3.org/TR/xpath-datamodel`.

[10] J. N. Foster and G. Karvounarakis. Provenance and data synchronization. *IEEE Data Engineering Bulletin*, Dec. 2007. Invited paper for special issue on provenance. To appear after revision.

[11] G. Ghelli, N. Onose, K. H. Rose, and J. Siméon. A Better Semantics for XQuery with Side-Effects. In *Workshop on Database Programming Languages (DBPL), Vienna, Austria*, volume 4797 of *Lecture Notes in Computer Science*, pages 81–96. Springer, Aug. 2007.

[12] G. Ghelli, C. Re, and J. Siméon. XQuery!: An XML Query Language with Side Effects. In *Workshop on Database Technologies for Handling XML Information on the Web (DataX), Munich, Germany*, volume 4254 of *Lecture Notes in Computer Science*, pages 178–191. Springer, Mar. 2006.

[13] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. 2007.

[14] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD Conference*, pages 328–339, 1995.

[15] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):508–511, 1997.

[16] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[17] A. Gupta and I. S. Mumick, editors. *Materialized views: Techniques, Implementations, and Applications*. MIT Press, Cambridge, MA, USA, 1999.

[18] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD Conference*, pages 157–166, 1993.

[19] H. Liefke and S. B. Davidson. View maintenance for hierarchical semistructured data. In *International Conference on Data Warehousing and Knowledge Discovery (DaWaK), London, UK*, volume 1874 of *Lecture Notes in Computer Science*, pages 114–125. Springer, 2000.

[20] M. Onizuka, F. Y. Chan, R. Michigami, and T. Honishi. Incremental maintenance for materialized XPath/XSLT views. In *International World Wide Web Conference (WWW), Chiba, Japan*, pages 671–681. ACM, 2005.

[21] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[22] C. Re, J. Siméon, and M. F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *International Conference on Data Engineering (ICDE), Atlanta, GA*, page 14. IEEE Computer Society, 2006.

[23] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *International Conference on Management of Data (SIGMOD), Baltimore, MD*, pages 443–454. ACM, 2005.

[24] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *International Conference on Very Large Data Bases (VLDB), Hong Kong, China*, pages 974–985. Morgan Kaufmann, 2002.

[25] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *International Conference on Very Large Data Bases (VLDB), Mumbai, India*, pages 227–238. Morgan Kaufmann, Sept. 1996.

[26] L. Villard and N. Layaïda. An incremental XSLT transformation processor for XML document manipulation. In *International World Wide Web Conference (WWW), Honolulu, HI*, pages 474–485. ACM, 2002.

[27] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *International Conference on Data Engineering (ICDE), Orlando, FL*, pages 116–125. IEEE Computer Society, 1998.

A. XQuery Algebra Typing

XQuery types are as follows:

t	$::= tt \mid xt$	(types)
xt	$::= \{tt\} \mid \mathit{Item}$	(data model types)
r	$::= x_1 : tt_1; \ldots ; x_n : tt_n$	(tuple types)
tt	$::= \{xt\} \mid [r]$	(table types)

A type t either describes a set of data model values (tt) or tables (xt). A data model type tt describes a set of sequences ($\{tt\}$) or items (Item). For simplicity, we do not distinguish between the various sorts of items (elements, attributes, text nodes, etc.). Table types describe tables ($\{xt\}$)—i.e., ordered sequences of tuples—or individual tuples ($[r]$). Tuple types are written as finite lists of pairs of field names x_i and data model types tt_i and describe tuples that have a field x_i that leads to a value belonging to tt_i for every i. The fields mentioned in a tuple type must have distinct field names; when there are repeated names, the type is undefined. Tuple types are equivalent up to reordering of fields. We write $[r_1; r_2]$ for the tuple type with the union of fields from $[r_1]$ and $[r_2]$ (it is undefined if the intersection of the set of field names in $[r_1]$ and $[r_2]$ is non-empty). Finally, the type constructor for sequences and tables is idempotent: i.e., for every type t the types $\{\{t\}\}$ and $\{t\}$ are equivalent. The typing relation for the XQuery algebra is given by the following set of inference rules.

$$\mathsf{ID} : t \to t \qquad\qquad \mathsf{Empty}() : t \to \mathit{Item}$$

$$\frac{p_1 : t \to \{\mathit{Item}\}}{\mathsf{Elem}[qn](p_1) : t \to \{\mathit{Item}\}} \qquad \frac{p_i : t \to \{\mathit{Item}\}}{\mathsf{Seq}(p_1, p_2) : t \to \{\mathit{Item}\}}$$

$$\frac{p_1 : t \to \{\mathit{Item}\}}{\mathsf{TreeJoin}[s](p_1) : t \to \{\mathit{Item}\}}$$

$$\frac{p_1 : t \to \{\mathit{Item}\} \qquad p_2 : t \to t' \qquad p_3 : t \to t'}{\mathsf{If}(p_1)\{p_2, p_3\} : t \to t'}$$

$$\frac{\#x_i : [x_1 : tt_1; \ldots ; x_k : tt_k] \to tt_i}{\#x_i : t \to tt_i} \qquad \frac{p_1 : t \to tt}{[x : p_1] : t \to [x : tt]}$$

$$\frac{p_2 : t \to \{t_2\} \qquad p_1 : t_2 \to \{t_1\}}{\mathsf{Map}\{p_1\}(p_2) : t \to \{t_1\}}$$

$$\frac{p_2 : t \to \{[r_2]\} \qquad p_1 : [r_2] \to \{[r_1]\}}{\mathsf{MapConcat}\{p_1\}(p_2) : t \to \{[r_2; r_1]\}}$$

$$\frac{p_2 : t \to \{[r]\} \qquad p_1 : [r] \to \{\mathit{Item}\}}{\mathsf{Select}\{p_1\}(p_2) : t \to \{[r]\}}$$

$$\frac{p_i : t \to \{[r_i]\}}{\mathsf{Product}(p_1, p_2) : t \to \{[r_1; r_2]\}}$$