Propagating Updates Through XML Views Using Lineage Tracing

Leonidas Fegaras

University of Texas at Arlington, CSE
Arlington, TX 76019, USA
fegaras@cse.uta.edu

***Abstract*— We address the problem of updating XML views over relational data by translating view updates expressed in the XQuery update facility to embedded SQL updates. Although our XML views may be defined using the full extent of the XQuery syntax, they can only connect relational tables through restricted one-to-many relationships that do not cause view side effects for a wide range of XQuery updates. Our approach is to use lineage tracing to propagate the necessary information about the origins of updatable data pieces through the query and the view code, to be used when these pieces are to be updated. Our system performs a compile-time analysis, based on polymorphic type inference and type usage, to detect the exclusive data sources, which are the table columns from the database that can be updated without causing side-effects to the view. The rest of the updates are associated with an update context in the form of a chain of tuples, which reflects the navigation path that was used to reach the update destination. At commit time, our system collectively considers all the compatible chains of all updates in the transaction and tries to relink them to new chains from the existing database whose tuples contain the updated data, so that the updates are reflected correctly without causing side effects to the other components of the view.**

I. Introduction

The widespread adoption of XML as a data representation and exchange format for Internet applications has rendered it as the language of choice for web data integration. In order for such integration systems to be effective, they must support both querying and updating. Although there is a large body of work on the relational view update problem, the problem of updating XML views has received very little attention. In this paper, we focus on virtual XML views defined on top of a relational database. Our goal is to translate XQuery updates over these XML views to embedded SQL updates over the base tables. Although the XQuery Update Facility (XUF) [25] is defined to operate on any kind of XML data, our goal is to handle only those updates whose destination has origins to updatable relational data. More specifically, we want to check whether a given view update can be traced back to its origin and, if it is, to generate the appropriate SQL updates so that the updated relational tables, when mapped through the view, would yield the same result as that we would have gotten if we had updated the view directly. To preserve the referential transparency of our transformation and optimization rules, we assume that XQuery updates have snapshot semantics, which indicates that the results of the updates are not visible during query evaluation (ie, updates cannot commit during querying).

To propagate the view updates back to their data sources, one needs to establish a link between the update destination and the data sources that were used to produce its value. This is called the data *lineage*, which is the description of the origins of a piece of data [10]. Thus, one needs to evaluate the view code in such a way that the lineage information is propagated from the data sources to the updates so that the lineage would precisely point to the place in the database where these updates should be directed to. In our framework, parts of the lineage information are statically inferred from the view code, while the rest annotate XML atomic data at run-time, requiring a minimal overhead.

An important goal of our framework is to prevent view side effects by restricting the way relational tables are joined in the XML view. Tables participating in a view can only be connected to each other through one-to-many relationships, using foreign keys to establish forward or backward links that form trees of table connections. Since our view updates are translated to base table updates, one important goal is to identify those columns $T.A$ associated with a relational table access in a view that may cause view side effects when updated. We call the column $T.A$ an *Exclusive Data Source (EDS)* if its values do not appear in the view output at all (which means that they are not updatable), or they appear in the view output only once, but they are not accessed elsewhere in the view code. In the former case, we cannot cause any side effect since it is impossible to update $T.A$; in the latter case, a view update on the component that correspond to $T.A$ can be safely translated to a base update over $T.A$ since it would only change this component, while leaving the rest of the view intact. One contribution of this paper is in the use of polymorphic type inference ([17] chapter 22) to check this semantic constraint. Although the XQuery Data Model (XDM) provides an extensive type system based on XML Schema [24], static type validation is optional and many implementations choose not to support it, while providing run-time checking instead. Given that XQuery does not require type annotations for variables and functions, static type validation is actually type inference, because it must be able to deduce the type of any XQuery expression. Considering the types involved, a view is a function from a database schema to some output type, in our case an XML type. Let this database schema be an aggregation of relations $T_i(A_{i1}: t_{i1}, \ldots, A_{in}: t_{in})$, where t_{ij} is an SQL base type. If the view code is well-typed, one may

infer the type of the view output using static type inference. There is no need for the XQuery type inference algorithm to be polymorphic, since XQuery does not support parametric polymorphism. But if we had used a Hindley/Milner-like polymorphic type inference system for XQuery, similar to that used by modern functional programming languages [17], we could have checked the EDS constraint by just looking at the view type. More specifically, instead of the SQL base type t_{ij}, we could have used a type variable a_{ij} for the type of column $T_i.A_{ij}$ in the view input. Let t be the inferred principal type of the view (a most general type that makes the view code typable). Then, a_{ij} must appear free in t in order for $T_i.A_{ij}$ to be EDS. The fact that a_{ij} is free in t indicates that the view is oblivious to the value of $T_i.A_{ij}$, which is passed to the view output as is (without analyzing its value). Although XQuery type validation is a very complex process, one can make it polymorphic quite easily: instead of type equality, the type inference algorithm must use type unification, which is similar to the unification in resolution theorem proving. The result of unification is a substitution list that binds type variables to types, which must be passed around and extended during type inference (both as a parameter and as a part of the result). Inferring that a_{ij} is free in t is a necessary but not a complete condition for $T_i.A_{ij}$ to be EDS; the $T_i.A_{ij}$ value must also appear at most once in the view output. In general, for higher-order types combined with lazy evaluation, one needs a complex usage analysis [23] to statically identify variables and expressions that will be evaluated at most once. But XQuery is first-order and strict (call-by-value), which makes easier to check the uniqueness of $T_i.A_{ij}$: we simply count how many times the type variable a_{ij} appears in the view output type t. This count should be at most one. There is a catch though. Even if the column $T_i.A_{ij}$ is used once in the output, a single row from this column may actually appear multiple times, as is apparent in forward references in the view tree, such as when along with each employee we embed her department name. To solve this problem, we consider as candidates for EDS only those columns that can be reached from a root of the view tree using backward links only. This is a conservative approach that, combined with our type and usage analysis, guarantees that each row from these columns may appear at most once in the output. Note that, the number of times $T_i.A_{ij}$ is used in the view code is not important, because we have already established that its value would not affect the view output (since a_{ij} is free). If $T_i.A_{ij}$, for example, were used in some aggregation that was part of the view output, then a_{ij} would have been bound to the int type. The proof that polymorphic type inference deduces the EDS constraint is given in the Appendix.

View updates on data that originate from an EDS column can directly propagate back to the underlying source data without side effects. To handle the other updates (over the non-EDS columns), we look at the total effects of a single transaction at commit time (at the end of a single XQuery). That is, instead of considering a single SQL update in isolation for side effects, as is done in related work, we look at the entire transaction, which allows a wider spectrum of updates. At run-time, each update is associated with an update context in the form of a chain of tuples, connected with foreign-to-primary key links (forward and backward), that reflects the navigation path in the table connection tree that was used to reach the update destination. This chain is truncated so that it always starts with a tuple (called the anchor) whose foreign or primary key used in the chain is an EDS and ends with the update destination tuple. At commit time, we collectively consider all the chains that start with the same anchor and we try to relink the anchor to a new chain from the existing database whose tuples contain the updated data, so that the updates are reflected correctly without causing any side effect to the other components of the view. For example, suppose that the view is a join between employees and departments so that, for each employee, we return the employee information along with the department name and phone number. The employee foreign key that links her to a department is an EDS. Suppose that we update both the name and phone number of the department of a particular employee. Neither of these columns is an EDS but this employee is the anchor for both updates. Then, if there is already an existing department under the updated name and phone number, this employee will be linked to it by updating its foreign key (which is permitted since it is an EDS); otherwise a new department will be inserted, which may violate the key constraint and may cause a run-time error.

The key contribution of our work is the development of a novel framework for updating virtual XML views that has the following characteristics:

- Unlike related work that focuses on restricted XML-to-relational mappings, such as view trees, our framework can handle XML views and integration mappings in which only the relational table connections are restricted to trees, while the actual view query can be any XQuery. That is, although we allow the full extent of the XQuery language when specifying views, in order to prevent side effects, we restrict the way relational tables are joined in the view.
- Our framework handles XQuery updates whose destinations have lineage that can be traced back to updatable relational tuples. Some of the lineage information is statically inferred with the help of the XQuery type inference algorithm.
- We facilitate the detection of view side effects by restricting the way relational tables are connected to form the view. Although our views are expressive enough to capture many common XML view scenarios, such as XML publishing and XML shredding using hybrid inlining or generic mapping, most side effects from view updates over our restricted views can be actually detected at compile-time using polymorphic type inference and a simple usage analysis. Based on these checks, updates that are certain not to cause side effects are accepted, incompatible updates or updates on data with no lineage are rejected, while the rest are tested at run-time, requiring a minimal run-time overhead.

- Instead of checking a single SQL update for side effects, as is done in related work, we check for side effects across a transaction (a single XQuery that may correspond to multiple SQL updates). This allows a wider spectrum of updates that would have been otherwise rejected.
- We provide a simple method to annotate XML atomic data with lineage information at run-time. The lineage annotations of the update destination are used to reflect the updates back to their data sources.
- Since the XQuery updates can be very complex and may correspond to multiple underlying SQL updates, we introduce a novel algorithm that generates these updates statically, guided by the type of the update destination.
- Finally, we report on a prototype XQuery engine that implements our view update framework and we present a performance study that assesses the feasibility of our approach.

The rest of the paper is organized as follows. Section II surveys related work. We describe our approach through two common examples of view mappings: A schema-based mapping (Section III) and a generic mapping (Section IV). Section V defines our update operations against the underlying database. Section VI describes how the lineage attributes are introduced by the relational data and how they are propagated at run-time. Section VII gives the rules for translating XQuery updates guided by the type of the update destination. Section VIII gives the XQuery optimization rules and the code folding rules that promote parts of the query evaluation onto the database engine. Finally, Section IX reports on a prototype XQuery engine that implements our view update framework.

II. Related Work

There is already a sizable body of work on the relational view update problem, since it is considered to be one of the most fundamental problems in relational databases. Although the source of view mappings are often normalized relational schemas, which do not have update anomalies, most virtual views generate data that are not normalized, since they often introduce data redundancies by joining tables together. Data redundancy does not affect view querying but it may cause view side effects when view updates are translated into base table updates [12]. When a user submits a view update and this update is mapped to base table updates, she expects that the results of these updates, when viewed through the view, would be the same as if she had updated the view directly. This means that, since a view is not normalized, a transaction that contains view updates that does not consistently update all replicas of an updated object in the view should be aborted, since it would violate the integrity constraints associated with the data redundancies in the view. Furthermore, there may be multiple translations of a view update that may cause the same view changes, and, often, a choice may depend on a particular application. There two main approaches for addressing this problem: schema-based and data-driven. The data-driven approaches are more expensive, since they require monitoring the actual updates to the base data to reject those that cause view side effects. One example of a schema-based approach is the work by Dayal and Bernstein [12] for checking the updatability of a relational view using FDs and view equality constraints. It works on single updates only, so that if the update destination does not correspond to a single source, as is deduced by the FD/view graph, then the update is outright rejected. This method is very hard to apply to XML views since it is not clear what an XML FD should look like and how one can derive equalities from complex XML views to construct the view dependency graph. Furthermore, this approach considers simple view updates in isolation, instead of considering the total effects of transactions, thus rejecting many common view updates that do not cause view side effects when combined with others in a transaction. The work by Bancilhon and Spyratos [1] gives a non-constructive proof of the existence of the inverse of a view update as long as it keeps the view complement invariant to the update. A compromise between fully materializing the view and using expensive tests to check for side effects is the work by Kotidis *et al* [15], that stores multiple versions of each value in the base tables, one for each value replica in a particular view.

Our view restrictions have been influenced by the work of Barsalou *et al* on object view updates [2]. They have restricted the structural schema of the underlying database to consist of three kinds of identifiable 1:N relationships only, the most important one being the ownership connection, where each tuple of the member table is connected to exactly one tuple in the owner table. View objects are defined by selectively removing some of the connections from the structural schema to form a tree. Their replacement updates are translated literally to base updates within a 'dependency island', which is a subtree of the object view that consists of ownership connections only. The rest, are either translated to insertions (under certain conditions), or rejected. In our framework, we provide a third alternative, in which certain objects are relinked to existing objects to reflect the updates.

Although there is a large body of work on the relational view update problem, the problem of updating XML views has not received the deserved attention. The strategy used in Pataxo' ([6], [7]) is to map XML views into relational views. Then, the XML update view problem is reduced to the relational view update problem, which is well studied and there are many existing solutions. As many other view querying frameworks, they use query trees to represent views, which are also used to derive the relational views. The work by Choi, et al [9] addresses insertion and deletion updates over simple XPath views that may be derived from recursive DTDs. They use a graph-theoretic, data-driven approach to map XML updates to SQL updates, which requires to maintain an auxiliary reachability matrix to hold all ancestor-descendant relationships, which is exponential in size. BEA's AquaLogic data services platform [5] automatically generates update maps for some update service data objects. It can handle limited 1:N (but not N:1) view trees that cannot cause a value in the source data services to be duplicated in the target service.

The closest work to ours is a hybrid framework, called

HUX ([21], [22]), that combines schema-centric and data-driven view update translations. The data-driven approach is monitoring the actual updates to the base data and reject those that cause view side effects. The authors introduced a semi-decidable method for determining the translatability of an update by examining the view schema. If the method fails, then the more expensive data-driven approach is used. Like our framework, each view element is assigned a *source*, which takes the form $T.i$, where T is a table and i is a row id. An update in their framework is acceptable iff it updates a clean extended source of a view element (ie, the element has an exclusive data lineage). They infer whether a source is clean or not from the relationship cardinalities used in the view joins. In our framework, these relationships are inferred from exclusive data sources (EDS). Furthermore, our algorithm can work on multiple layers of views and integration mappings.

Lineage tracking has also been used for consistent representation of uncertain data [3] and for propagating annotations in relational queries [4]. It is also related to data provenance, which is used to link components of the output to the originating components in the input ([8], [14]). Data provenance is defined to be the data origins and how it came to be included in a database [8]. Our work does not address the general lineage or provenance problem. Instead, it uses the concept of lineage tracing for a specific goal: to propagate updates through views. Contrary to the related work on lineage and provenance, our lineage annotations have two components: One related to the 'lineage kind', which is statically inferred so that it can be used to check the translatability of updates and generate the base updates at compile-time, and another needed to track the exact lineage at run-time.

Finally, our approach of using polymorphism for detecting side effects was highly influenced by the work of Voigtlander on bidirectionalization [20]. It assumes that views $S \to V$ are polymorphic of type $S(\alpha) \to V(\alpha)$, where the type parameter, α, encapsulates the data unit, which is the update granularity. To detect view side effects and map V updates to S updates, it lifts $S(\alpha)$ to $S(int)$ by enumerating the α values in S with unique numbers. Then, it applies the view to $S(int)$ to get a $V(int)$, which is compared with the updated view. This comparison yields bindings from int to values, which should be unique to be side-effect free. These bindings, when applied to $S(int)$, give the updated source. Although this method does not need to inspect the view code, requiring that views be polymorphic is restrictive because one cannot express content-based filters or embed constant values to the view. On the other hand, in our views, only the updatable view components that appear in the view output must be polymorphic, while the others may be freely used in filters. More importantly, their work requires the full comparison between the before and after view values, which can be proportional to the data size.

III. An XML View Example

In this section, we use an XML view example to describe our approach for propagating view updates to their source using lineage tracing. This example deals with XML publishing, where pre-existing relational data are exported in XML format. More specifically, consider a small part of the DBLP database described by the following relational schema (along with their associated type variables α_{ij}):

Inproceedings (key, title, year)	$T_1(\alpha_{11}, \alpha_{12}, \alpha_{13})$
Person (pid, name)	$T_2(\alpha_{21}, \alpha_{23})$
Author (keyref, pid)	$T_3(\alpha_{31}, \alpha_{33})$
Cite (keyref, citation)	$T_4(\alpha_{41}, \alpha_{43})$

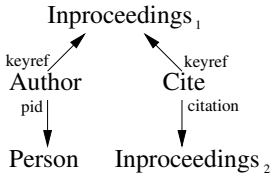
where Author.keyref, Cite.keyref, and Cite.citation are foreign keys that reference the Inproceedings.key, while the Author.pid is a foreign key that references the Person.pid. We assume that every tuple has a unique tuple id that is not visible to views (but, as we will see, it is accessible as a special system attribute).

A. Specifying the View

In multi-layer XML views on relational databases, all but the inner view layer can be defined in plain XQuery. The inner view layer though must access the relational database and convert relational data to XML. Consequently, the inner view must use two incompatible languages: SQL to extract the relational data and XQuery to convert this data to XML format. One way to glue these two languages together is to use a canonical view of relational data, as is done in SilkRoute [13] and many other systems. In our framework, this is done with the help of the special XQuery syntax, R**[** T, ... **]**, called a *table access*, where T is a relational table and '...' is an optional referential constraint. The referential constraint takes the form $A = e$, where A is either the primary key of T referenced by a foreign key accessed by e (a forward reference), or A is a foreign key of T that references the primary key of some table accessed by e (a backward reference). Given this notation, the XML view, $view, over the DBLP database may look like this:

```
<dblp>{
  for $i in R[ Inproceedings ]
  return <inproceedings>{
           $i/title,
           $i/year,
           for $a in R[ Author, keyref=$i/key ]
           return <author>{
                    R[ Person, pid=$a/pid ]
                      /name/data()
                  }</author>,
           for $c in R[ Cite, keyref=$i/key ]
           return <cite>{
                    R[ Inproceedings, key=$c/citation ]
                      /title/data()
                  }</cite>
         }</inproceedings>
}</dblp>
```

Note that the table Inproceedings appears twice in the view, so that for each article, we get the titles of the related citations. The table connections form the following view tree implicitly:



Given that access tables with backward connections return sequences while those with forward connections return optional elements (for clarirty, the optionality qualifier '?' is omitted), the XQuery type of $view is inferred to be:

```
element dblp {
  element inproceedings {
    element title { α12 },
    element year { α13 },
    element author { α22 }*,
    element cite { α12 }*
  }* }
```

where the type variables α_{12}, α_{13}, and α_{22} are associated with table columns Inproceedings.title, Inproceedings.year, and Person.name, respectively. The *ownership top* of a view consists of all the table accesses that can be reached from a view root through backward references only. For our view, these are the table accesses Inproceedings_1 (the root), Author, and Cite. Only the table columns from the ownership top can potentially satisfy the EDS constraint, since all the other columns may be replicated. For example, the Person.name cannot be EDS since the same person may be the author of multiple inproceedings. On the other hand, not all columns from the ownership top are necessarily EDS. We can see from the type of $view that Inproceedings.title, which corresponds to α_{12}, appears twice in the type, which indicates that it is not EDS. On the other hand, Inproceedings.year and Person.name are EDS. All the other table columns are not EDS because they are used in conditions. For example, because of the predicate keyref=$i/key, the type variables α_{31} and α_{11}, which correspond to Author.keyref and Inproceedings.key, will not be free since α_{31} is bound to α_{11}.

Note that our framework requires that the XQuery data model be extended with null values[1]. Elements and attributes with null content are removed during processing while operations on nulls return null. For example, if Inproceedings.year is null, then there will be no year child element in the view. This is actually the only way to remove the year element from the view.

B. Updating the View

Consider the following XQuery update:

```
for $i in $view//inproceedings[author="John Smith"]
where $i/title = "XML for Dummies."
return replace $i/year with 2009
```

which replaces the publication year of a certain article with the value 2009. The type of the update destination $i/year is inferred to be **element** year $\{\alpha_{13}\}$, where the type variable α_{13} is associated with the column Inproceedings.year. Given that the Inproceedings.year column is EDS, it can be updated directly without side-effects. But how do we map these updates back to the database? What is missing here is a link between the data generated through a view and the source that produced this data. This is called the *lineage*, which is the description of the origins of each piece of data [10]. In our framework, lineage comes in the form of special annotations attached to XML atomic types only (such as the element text content and attribute values), because these are the only XML components that may have been originated from the database. In the XQuery data model, these are the XML values of type xs:anyAtomicType. Although these annotations require that the XQuery data model be extended to incorporate them, it would not require extensive modifications to an existing XQuery interpreter, because lineage should be passed through all standard XQuery operations as is, while all newly constructed atomic values should be annotated with the empty lineage (the default) during XQuery processing. If, for example, $x in $x+1 is an xs:int that has a direct lineage to the database, then $x+1 should ignore the lineage and create a new integer value with empty lineage. The only places where lineage is taken into account are when it is embedded to atomic values (ie, when database data are converted to XML) and when it is used in XQuery updates. For direct updates, such as updating an EDS column, the only lineage information needed to reflect the previous update back to the database is the table name, the column name, and the tuple id. Recall that both the table and the column names are statically derived from the type of $i/year, which corresponds to Inproceedings.year. But, as we will see in Section V, to handle general updates, instead of a single tuple id, our framework annotates an atomic value with two lists: an id list, ids, and a list of foreign key names, path, which are used to define the update context.

IV. A Generic Mapping Example

Our second example is exporting shredded XML data from a generic relational storage that has the following fixed schema:

Elem (eid, parent, tag)
Text (eid, parent, text)

Note that there is no reason to use a special numbering scheme to capture parent-child relationships, such as pre-/post-order numbering, since our views are schema-based, which make explicit these relationships from the schema. For example, the view of the DBLP XML data can be defined as follows:

```
<dblp>{
  for $i in elem("inproceedings",0)
  return <inproceedings>{
    for $t in elem("title",$i/eid)
      return <title>{content($t)}</title>,
    for $y in elem("year",$i/eid)
      return <year>{content($y)}</year>,
    for $a in elem("author",$i/eid)
      return <author>{content($a)}</author>,
    for $c in elem("cite",$i/eid)
      return <cite>{ content($c) }</cite>
  }</inproceedings>
}</dblp>
```

which uses the following functions:

[1]The standard XQuery data model does not directly support null values but it provides a special attribute, xs:nil, to indicate that the content of an element is null.

elem($tag,$pid) = R[Elem, parent=$pid][tag=$tag]
content($e) = R[Text, parent=$e/eid]/text/data()

Unfortunately, none of the components of this view is EDS: The ownership top of the view tree covers all table accesses, five for Elem and four for Text. That is, the attribute Text.text appears four times in the view output, which violates the EDS constraint. Of course, it is obvious that all these table references to Elem and Text in the view generate disjoint sets of tuples, but this is impossible to prove in general. The root of the problem is that our trick of associating unique type variables to table columns works because we have a finite number of them in a relational database schema. We cannot do the same trick to analyze rows since we may have an arbitrary number of rows. We can specify this disjoint property though as follows: we use the syntax R$*$[$T, \ldots$], instead of R[$T, \ldots$], to indicate that this table access to T yields a set of tuples in the view output that is disjoint from the other table accesses in the view. The result of this annotation is that the type inference engine will assign different type variables to T columns in R$*$[$T, \ldots$], which is basically equivalent to treating this instance of the table T as a distinct table. This disjoint property cannot be statically deduced for a computationally complete language. Instead, our framework assumes that this property holds by definition; if it does not, there will be view side effects when a view update is mapped to base updates. Nevertheless, requiring this property for some views is a smaller annoyance than specifying FDs on the view result, which, in general, cannot be deduced from the base FDs.

V. The Database Interface

Consider the following relational schema:

Employee (ename, dno, salary)
Department(dno, dname, dphone)

where Employee.dno is a foreign key that references Department.dno. If we nest employees inside departments in an XML view, all table attributes would be EDS and can be safely updated without side-effects (as long as they satisfy the underlying database constraints). If we nest departments inside employees though, which corresponds to the relational view

EmpDept (ename, salary, dname, dphone)

derived by joining Department and Employee, only the employee attributes would be EDS, since the same department may be referenced by multiple employees. Consider now the update that replaces the dname of a tuple t in EmpDept with a new name n. We assume that t was produced by joining the tuples $te \in$ Employee and $td \in$ Department. Obviously, we should not replace td.dname with n because it may change other tuples in EmpDept (all those derived by joining with td). But suppose that there is already a department td' with the name n. Can we simply disconnect te from td and connect it to td' (by replacing te.dno with td'.dno)? The answer depends on the other attributes of td that are visible in the view. In our case, either td.dphone $= td'$.dphone or t.dphone should have been updated (in the same transaction) to be equal to td'.dphone, because, if not, the view will have a side effect on the value of t.dphone. This condition can only be checked at commit time by inspecting the data values of the tuples td and td'. In our framework, our view mapping algorithm generates for this update the following call:

db:replace(Department.dname,n,[td,te],[Employee.dno])

at compile-time to be executed at commit time. At commit time, there may be another db:replace call to update dphone on the same tuple id list, [td,te]. All these replace calls associated with the same id lists are considered and a new desired tuple td' is synthesized from the old td and the updated values. Then our system will locate a tuple from the table Department using the desired values, which can be done very fast because td' contains the primary key value (the dname n). If the key does not exist, then td' will be inserted in Department and will be connected to te. If the tuple with the desired key exists, if there is a mismatch with the other desired values, the transaction is aborted, otherwise the tuple is connected to te. Note that, although this method captures a very common scenario that avoids side effects, this is not the only scenario. If one replaces the dphone of *all* the tuples in EmpDept under a given dname, then the intention is probably to update the dphone of the Department under this name since this too will not cause any side effect. This case is very expensive to test on the data because it requires to access a large number of tuples. Such cases are not considered in our framework. Another issue not considered is garbage-collecting unreachable tuples.

To generalize this update, we define an ids list to be a sequence of tuple ids of type xs:int* and a path list to be a sequence of foreign key names of type xs:string*. A foreign key (FK) in a path list has the syntax "T.A" or "T.[A_1,...,A_n]", where A and A_i are attributes of the relational table T. In general, ids=[$t_0, t_1, \ldots, t_n$] and path=[$FK_1, \ldots, FK_n$], $n \geq 0$, where t_0 is the id of the update destination tuple and t_n is the id of the anchor tuple, which requires that FK_n be an EDS. We assume that we statically know the table names referenced by the FKs in the path list. Given the table name T of t_0 (which is a db:replace parameter), the table T_i of t_i is equal to the domain of FK_i (which is part of the FK_i string), if this domain is different from T_{i-1}, or, otherwise, to the table name referenced by FK_i.

A replacement update can take the general form

$$\text{db:replace}(T.A, v, ids, path)$$

where v is the updated value for the attribute $T.A$. At commit time, all the updates over the path $[t_i, \ldots, t_n], i \geq 0$ (ie, all those that start with the same anchor t_n) are considered to find a new chain of size n starting with t_0 that is 'compatible' to these updates. That is, all the visible columns of the new tuples must contain the updated values, or the old values if they have not been updated. As before, these tuples can be retrieved very fast since we know their primary keys. The special case of $n = 0$ requires that $T.A$ be EDS, since it is a direct update on $T.A$.

For example, consider the replacement update:

db:replace (T.A, v, [t0,t1,t2,t3,t4], [T.B,T2.C,T2.D,T4.E])

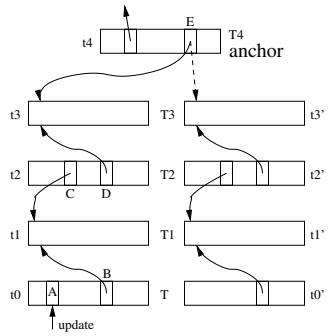


Fig. 1. An update on a non-EDS attribute T.A

shown in Fig.1. The tuples t0, t1, t2, t3, t4 come from the tables T, T1, T2, T3, T4, respectively, as it can be deduced from the foreign keys. At commit time, our system will collect all updates to the t0-t3 tuples and will try to find a new chain t0', t1', t2', t3', t4 that originates from the same anchor t4 that reflects these updates. Note that each tuple in t0'-t3' must be different from the corresponding tuple in t0-t3 because the only update allowed is on T4.E of t4, which is an EDS. If no existing chain is found, then four new tuples will be inserted in the tables T, T1, T2, T3, will be linked, and will be populated with the updated values, which may cause a run-time error if the key constraints are violated.

For deletions, we use db:replace($T.A$, NULL, ids, $path$). Setting the column $T.A$ of a tuple to NULL indicates that every XML element whose content is derived from this value will be deleted from the view, because of our XQuery null semantics. For example, when we delete T.A of the t0 tuple in Fig. 1, this column is set to NULL. At commit time, if all the visible columns of the tuples t0, t1, t2, t3, t4 have been set to NULL, then the foreign key of the anchor T4.E is set to NULL, thus disconnecting the chain from t4. Otherwise, the transaction is aborted. As before, the only database update is setting T4.E to NULL, without touching the other tuples. For an empty chain, when the update destination $T.A$ is an EDS, we just set it to NULL.

For simple insertions, where the destination is the content of a single element with direct lineage to a table column $T.A$, we use the following variation of db:replace:

$$\text{db:replace-null}(T.A, v, [t_0, \ldots], path) = \begin{cases} \text{db:replace}(T.A, v, [t_0, \ldots], path) & \text{if } t_0.A = \text{NULL} \\ \text{error} & \text{otherwise} \end{cases}$$

where v is the value to insert. The most common form of insertion though is the one in which the source becomes the child of the destination, such as **insert** <cite>A title</cite>**into** $i, where $i is an inproceedings element from the $view. Insertions like this require the creation of a new chain of empty tuples in the database, whose values must be replaced with those of the insertion source. The new tuples are inserted using the function db:insert(T), which inserts a new tuple in the table T and returns its tuple id. Our approach is to translate the insertion update to a replacement update over a new portion of the view that has direct lineage to the newly inserted tuples. This translation is described in detail in Section VII. Note that, even with snapshot semantics, db:insert causes side effects, since it always returns a new tuple id. Thus, it requires special attention during query normalization and optimization.

VI. Annotating Atomic Values

In this section, we explain our method for annotating atomic XML values with lineage attributes. At run-time, every XML atomic value (the text content of an XML element or an attribute value) is annotated with two system attributes, ids and path, of type xs:int* and xs:string*, respectively, which are the lineage chains used in updating: the list of strings in the path list is the foreign key names in the navigation path that reaches this atomic value, while the list of numbers in the ids list are the ids of the tuples participating in this path. The XQuery engine is oblivious to these lineage attributes, which are propagated through all standard XQuery operations as is. If an XQuery operation creates a new atomic value, its lineage is set to empty (empty ids and path lists). The only places where these annotations are used are when database data are converted to XML and when XQuery updates are translated to SQL updates based on the lineage of the update destination. Consequently, it is trivial to extend an existing XQuery processor to annotate atomic values with lineage, provided that when an atomic value is passed as an argument to a function call or is embedded into an element construction or attribute, it is not copied into a new value (since this will set its lineage to null). The following functions are used to access and set the lineage of an atomic value:

```
getIds ( x as xs:anyAtomicType ) as xs:int*
getPath ( x as xs:anyAtomicType ) as xs:string*
setLineage ( x as xs:anyAtomicType,
             ids as xs:int*, path as xs:string* )
```

We first describe how these lineage attributes are generated from a database. The simplest table access is R[T_i], where T_i is a relational table with schema $<A_{i1} : t_{i1}, \ldots, A_{in} : t_{in}>$. Recall that, in our framework, each SQL base type, t_{ij}, in the schema of T_i is generalized to a fresh type variable, α_{ij}. Then, the result of R[T_i] is an XML value of type:

$$\textbf{element}\ \text{row}\ \{\ \ldots,\ \textbf{element}\ A_{ij}\ \{\ \alpha_{ij}\ \},\ \ldots\ \}*$$

that is, it is a sequence of row elements, one row per tuple, where each row contains the tuple columns as elements tagged with the column name, A_{ij}. All atomic values whose type is a free type variable, such as those of type α_{ij} above, are always annotated with a non-null lineage. More specifically, the ids of the atomic values generated by R[T_i] is a singleton sequence that contains the tuple id, while their path is equal to the empty sequence. The table access R[$T_i, A_{ik} = e$] is valid if A_{ik} is a column of T_i and the inferred type of e is **element** A $\{\alpha_{jl}\}$, for some tag A. This indicates that e must have a direct lineage from a table attribute $T_j.A_{jl}$. Furthermore, either $T_i.A_{ik}$ must be a foreign key that references the primary key $T_j.A_{jl}$ of T_j (a backward reference), or $T_j.A_{jl}$ must be a foreign key that references the primary key $T_i.A_{ik}$ of T_i (a forward reference).

For backward references, the type of R[$T_i, A_{ik} = e$] is equal to the type of R[T_i], but for forward references, it is equal to **element** row $\{\ldots\}$? (ie, ? instead of $*$). The resulting XML elements from this table access must extend the id and path chains from the table access T_j. This is done with the help of the following XQuery function:

```
declare function extend ( $source as element(row),
                          $p as element(),
                          $fk as xs:string ) as element(row) {
<row>{
   for $c in $source/*
   return element {$c/name()} {
      setLineage( $c/data(),
                  (getIds($c/data()),getIds($p/data())),
                  ($fk,getPath($p/data())) ) }
}</row>}
```

Given that the lineage of the atomic values in $source is derived from R[T_i], getIds($c/data()) is a single tuple id. Thus, for each atomic value in $source, extend appends its tuple id and the foreign key $fk to the lineage of $p. The expression R[$T_i, A_{ik} = e$] is translated into the XQuery:

$$\textbf{for } \$x \textbf{ in } \mathcal{R}[\, T_i \,] \textbf{ where } \$x/A_{ik} = e \textbf{ return } Q$$

where Q depends on whether the reference is forward or backward and whether the foreign key is EDS:

1) **Backward, EDS:** if $T_i.A_{ik}$ is a FK that references the PK $T_j.A_{jl}$ and $T_i.A_{ik}$ is EDS, then $Q = \$x$.
2) **Backward, not EDS:** if $T_i.A_{ik}$ is a FK that references the PK $T_j.A_{jl}$ and $T_i.A_{ik}$ is not EDS, then $Q =$ extend($\$x, e,$ "$T_i.A_{ik}$").
3) **Forward:** if $T_j.A_{jl}$ is a FK that references the PK $T_i.A_{ik}$, then $Q =$ extend($\$x, e,$ "$T_j.A_{jl}$").

For example, consider the view, $view:

```
for $e in R[ Employee ]
return <emp>{ $e/name,
                 R[ Department, dno=$e/dno ]/dname
        }</emp>
```

which is translated to:

```
for $e in R[ Employee ]
return <emp>{ $e/name,
                 ( for $d in R[ Department ]
                   where dno=$e/dno
                   return extend($d,$e/dno,"Employee.dno")
                 )/dname
       }</emp>
```

It annotates every atomic value in $e with a singleton ids list that contains an Employee id and an empty path list, and annotates each atomic value in $d with an ids list of two ids, (a Department id along with its parent, an Employee id), and with a path equal to "Employee.dno".

VII. Translating the Updates

Figure 2 presents the rules for translating XQuery updates to SQL updates at compile-time, guided by the type of the update destination. We focus on general XQuery updates of the form **insert** s **into** d, **replace** d **with** s, and **delete from** d, although our framework can be easily extended to handle many other forms specified by the XQuery update proposal [25].

Function $\mathcal{U}[\![\, t \,]\!](e)$ used in Figure 2 translates the update e at compile-time, guided by the type t of the update destination. The type t must be explicitly passed as a parameter to handle insertions, as we will show in Rule (8). Rules (1)-(3) translate updates whose destination is an atomic value with a direct lineage to the database. They have been explained in detail in Section V. Rules (4)-(7) are some of the rules that simplify updates recursively based on the destination type. Rule (8), which handles insertions, applies when the insertion source, s, is an element tagged A to be inserted to the destination that expects multiple A children, that is, when the result of projecting the type t over A (denoted by t/A) takes the form t_2* for some type t_2 (this means that t_2 must also be an element type tagged A). Before we explain Rule (8), consider the update **insert** $ni **into** $view, where $ni is the following new inproceedings to be inserted in the view:

```
<inproceedings>
   <title>T</title><year>Y</year>
   <author>A</author><cite>C1</cite><cite>C2</cite>
</inproceedings>
```

Given that the outer tag of $ni is inproceedings, the type of $ni should be compatible to the type of $view/inproceedings, which is inferred to be:

element inproceedings {
  element title { α_{12} }, **element** year { α_{13} },
  element author { α_{22} }*, **element** cite { α_{12} }* }*

Based on this type, we can construct a new XML element $y:

```
<inproceedings>
   <title>null</title><year>null</year>
   <author>null</author><cite>null</cite>
</inproceedings>
```

that has singleton sequence elements and null atomic values. Then, the above update becomes **replace** $y **by** $ni, which replaces the null values in $y with the corresponding values from $ni and, based on Rule (6), it inserts the author and cites using this technique recursively. To make this translation work, the null values in $y must be annotated with the appropriate lineage. But the relational tuples associated with $y have yet to be inserted into the database. Our solution to this problem is to assume that there is at least one other $view/inproceedings element $x, say ($view/inproceedings)[1], whose content has a direct lineage to the database and use it as a guide for inserting new tuples in the database that have the same shape as those used in $x. Then, the lineage of the new tuples would be the same as that of the tuples in $x but with the old tuple ids replaced with the corresponding new ids. Requiring that such an element $x exists is a serious limitation of our method, which will not work for an empty database.

In general, Rule (8) retrieves an element $\$x$ from the sequence d/A. Given that d has type t, then d/A has type t_2*, which means that $\$x$ is an element tagged A. Both the shape and the lineage of $\$x$ are used for creating the replacement destination $\$y$. First, a binding list $\$rho$ is constructed, which maps tuple ids from $\$x$ to new tuple ids. The new tuples are created using the function db:insert(T_i), which inserts a new tuple into the table T_i and returns its tuple id. The binding list $\$rho$ is constructed by traversing both the destination type t_2

(1) $\mathcal{U}[\![\ \alpha_{ij}\ ]\!]$(**replace** d **with** s)
$=$ db:replace($T_i.A_{ij}$, s/data(), getIds(d), getPath(d))

(2) $\mathcal{U}[\![\ \alpha_{ij}\ ]\!]$(**delete** d)
$=$ db:replace($T_i.A_{ij}$, NULL, getIds(d), getPath(d))

(3) $\mathcal{U}[\![\ \alpha_{ij}\ ]\!]$(**insert** s **into** d)
$=$ db:replace-null($T_i.A_{ij}$, s/data(), getIds(d), getPath(d))

(4) $\mathcal{U}[\![$ **element** A $\{t\}$ $]\!]$(**replace** d **with** s)
$= \mathcal{U}[\![\ t\ ]\!]$(**replace** d/* **with** s/*)

(5) $\mathcal{U}[\![\ t_1, t_2\ ]\!]$(**replace** d **with** s)
$=$ ($\mathcal{U}[\![\ t_1\ ]\!]$(**replace** d[1] **with** s[1]),
$\mathcal{U}[\![\ t_2\ ]\!]$(**replace** d[2] **with** s[2]))

(6) $\mathcal{U}[\![\ t*\ ]\!]$(**replace** d **with** s)
$=$ ($\mathcal{U}[\![\ t*\ ]\!]$(**delete** d), $\mathcal{U}[\![\ t*\ ]\!]$(**insert** s **into** d))

(7) $\mathcal{U}[\![\ t*\ ]\!]$(**delete** d)
$=$ **for** $\$x$ **in** d **return** $\mathcal{U}[\![\ t\ ]\!]$(**delete** $\$x$)

(8) s : **element** A $\{t_1\}$ $\wedge$ $t/A = t_2*$ $\Rightarrow$
$\mathcal{U}[\![\ t\ ]\!]$(**insert** s **into** d)
$=$ **let** $\$x := (d/A)[1]$,
$\$rho := \mathcal{L}[\![\ t_2\ ]\!](\$x, ())$,
$\$y := \mathcal{D}[\![\ t_2\ ]\!](\$x, \$rho)$
return $\mathcal{U}[\![\ t_2\ ]\!]$(**replace** $\$y$ **with** s)

Fig. 2. Type-Guided Update Code Generation

and the guiding value $\$x$:

$$
\begin{array}{lcl}
\mathcal{L}[\![\ a_{ij}\ ]\!](x,\rho) & = & \textbf{if } id \in \rho \textbf{ then } \rho \\
 & & \quad \textbf{else } \rho[id/\text{db:insert}(T_i)] \\
 & & \text{where } id = (\text{getId}(x))[1] \\
\mathcal{L}[\![\textbf{element } A\ \{t, \textbf{attribute } B\ \{t'\}\}\ ]\!](<A\ B{=}v{>}\{x\}{<}/A{>}, \rho) & & \\
 & = & \mathcal{L}[\![\ t'\ ]\!](v, \mathcal{L}[\![\ t\ ]\!](x,\rho)) \\
\mathcal{L}[\![\ t_1, t_2\ ]\!]((x_1,x_2),\rho) & = & \mathcal{L}[\![\ t_2\ ]\!](x_2, \mathcal{L}[\![\ t_1\ ]\!](x_1,\rho)) \\
\mathcal{L}[\![\ t*\ ]\!](x,\rho) & = & \mathcal{L}[\![\ t\ ]\!](x[1],\rho)
\end{array}
$$

where $\rho[id/\text{db:insert}(T_i)]$ extends the binding list ρ with a new binding from the old id to a new id returned by db:insert. To ease explanation, we listed few of the cases and assumed that an element construction, in addition to the element content, has one attribute. The actual replacement destination $\$y$ is synthesized by copying the shape of $\$x$ and by replacing the atomic values with null that have lineage derived from $\$rho$:

$$
\begin{array}{lcl}
\mathcal{D}[\![\ a_{ij}\ ]\!](x,\rho) & = & \text{setLineage}(null, \rho(\text{getId(x)}), \\
 & & \quad \text{getPath(x)}) \\
\mathcal{D}[\![\textbf{element } A\ \{t, \textbf{attribute } B\ \{t'\}\}\ ]\!](<A\ B{=}v{>}\{x\}{<}/A{>}, \rho) & & \\
 & = & <A\ B{=}\text{“}\{\mathcal{D}[\![\ t'\ ]\!](v,\rho)\}\text{”}{>}\{\mathcal{D}[\![\ t\ ]\!](x,\rho)\}{<}/A{>} \\
\mathcal{D}[\![\ t_1, t_2\ ]\!]((x_1,x_2),\rho) & = & (\mathcal{D}[\![\ t_1\ ]\!](x_1,\rho), \mathcal{D}[\![\ t_2\ ]\!](x_2,\rho)) \\
\mathcal{D}[\![\ t*\ ]\!](x,\rho) & = & \mathcal{D}[\![\ t\ ]\!](x[1],\rho)
\end{array}
$$

where $\rho(\text{getId(x)})$ replaces the old ids of x with the new ids from ρ. Note that the last rules for $\mathcal{L}[\![\ t*\ ]\!]$ and $\mathcal{D}[\![\ t*\ ]\!]$ recreate the x shape and replace its lineage with new ids.

VIII. Query Optimization

Although XQuery to SQL translation has already been addressed by others at different situations ([13], [11], [16], [22]), it is a very important component to our framework that enables our approach for update propagation through views. A view often transforms a large part of the underlying data source into XML, while a casual user query or update may access only a small part of this view. Normalization rules can eliminate those parts of the view that do not contribute to the query result. That is, normalization fuses the query with the view and eliminates most parts of the intermediate data so that the resulting query will work directly on the source data bypassing the view. Many XQuery optimizers use normalization rules, such as

$$
\begin{array}{rcl}
(<A>e</A>)/\text{child} :: B & = & e/\text{self} :: B \\
(<A>e</A>)/\text{self} :: A & = & <A>e</A>
\end{array}
$$

to partial evaluate operations on constructions and sequences, and eventually eliminate some of their components. When working with traditional databases, though, that do not provide XQuery functionality, we need to promote parts of the query evaluation onto the database engine. This is necessary because, even after normalization, the resulting query input will still remain the view input, which can be the entire database. While other systems use one-shot SQL derivations from view forests [13] or query trees [7], our approach is purely transformational. It starts from a query that works on the full content of relational tables (R[T] in our notation), promotes relevant SQL predicates into SQL queries, and fuses SQL queries in pairs using transformation rules. Obviously, these rules are heuristics and not all XQuery code can be promoted to SQL. In this section, we give some of these rules.

A function used by many of our SQL code folding rules is $\text{split}_E(p)$, which takes a binding list E that binds XQuery variables to SQL headers and an XQuery predicate p. For example, if E contains the binding from the XQuery variable v to the SQL header 't.A, t.B, s.C', where t and s are SQL variables, then $E[v/A] =$ t.A. The result of $\text{split}_E(p)$ is the pair (p_1, p_2), where p_1 is an SQL predicate and p_2 is an XQuery predicate. It conservatively promotes parts of the predicate p into the SQL predicate p_1, while leaving the rest to p_2. In the worst case, $\text{split}_E(p) = (\text{true}, p)$, that is, no part of p is promoted to SQL. Here are some cases of predicate promotion:

$$
\begin{array}{l}
v_i, v_j \in E \ \Rightarrow\ \text{split}_E(\$v_i/A \text{ cmp } \$v_j/B) \\
=\ (E[v_i/A]\ \text{cmp'}\ E[v_j/B], \text{true}) \\
\\
v \in E \ \Rightarrow\ \text{split}_E(\$v/A \text{ cmp const}) \\
=\ (E[v/A]\ \text{cmp'}\ \text{const}, \text{true}) \\
\\
\text{split}_E(p_1) = (p_1^1, p_1^2) \ \wedge\ \text{split}_E(p_2) = (p_2^1, p_2^2) \\
\Rightarrow\ \text{split}_E(p_1 \text{ and } p_2) \ =\ (\ p_1^1 \text{ and } p_2^1,\ p_1^2 \text{ and } p_2^2\ )
\end{array}
$$

where cmp' is the SQL equivalent of the XQuery comparison

operator cmp (eg, $<$ for lt and $<$).

Although we have already provided syntax for embedding table accesses in a view, we need a special XQuery syntax, $\mathcal{SQL}$**[ ]**, to embed more general SQL code inside an XQuery so that more complex SQL computations are pushed onto the database engine. For example, consider the following XQuery on $view:

for $i **in** $view//inproceedings[author="John Smith"]
where $i/title = "XML for Dummies." **return** $i/year

We would like to generate the XQuery **for** $i **in** $sql **return** $i/year, where $sql is equal to:

$\mathcal{SQL}$**[** year: i.year,
from (Inproceedings i **join** Author a **on** a.keyref=i.key)
join Person p **on** p.pid=a.pid
where i.title='XML for Dummies.'
and p.name='John Smith' **]**

The actual SQL header associated with this $sql expression is **select** ids: i.id, i.year, where id is the tuple id, while the result is implicitly converted into XML data of type:

element row { **element** year { xs:string } }*

where the atomic values (here the xs:string) have been annotated with the appropriate lineage (ie, an empty path and ids from the SQL result). Before we describe our optimization rules, we explain how these $\mathcal{SQL}$**[ ]** expressions are generated from table accesses in the view. After normalization, the resulting XQuery may contain multiple R[T] and R[$T, A = e$]. These expressions are translated to $\mathcal{SQL}$***[row: (t.*), from T t where t.A=e′]***, where e' is the SQL translation of e. In general, a generalized SQL query takes the form $\mathcal{SQL}$***[*** h, ***from*** f ***where*** p ***order by*** o ***]***, where h is the header that may implicitly nest the SQL result, f is the SQL 'from' part, p is the SQL predicate (equal to true, if missing), and o are optional order-by attributes. The result of this query is XML data, nested based on the header pattern. More specifically, given the header h, the XQuery type of the above query is $\mathcal{H}[\![\ t\ ]\!]$, defined as follows:

$$\begin{aligned} \mathcal{H}[\![\ A:(t)\ ]\!] &= \textbf{element}\ A\ \{\ \mathcal{H}[\![\ t\ ]\!]\ \}* \\ \mathcal{H}[\![\ t_1, t_2\ ]\!] &= \mathcal{H}[\![\ t_1\ ]\!], \mathcal{H}[\![\ t_2\ ]\!] \\ \mathcal{H}[\![\ x.A\ ]\!] &= \textbf{element}\ A\ \{\ \text{xs:string}\ \} \end{aligned}$$

Note that the actual data grouping is not done with an SQL group-by; instead it is done implicitly based on the header description. This grouping is done using streaming, without caching, because the unnested data coming from SQL are produced by left-outer joins ordered by the id of the left input.

The following rewrite rules promote relevant predicates inside SQL queries and fuse two SQL queries. Consider first the following XQuery expression:

for $$v$ **in** $\mathcal{SQL}$***[*** h, ***from*** f ***where*** p' ***order by*** o ***]***
where p **return** e

Let header(h) be the function that returns the SQL 'select' header from h. For example, the SQL 'select' header of the previous generalized SQL is i.id, i.year, a.id. If $p_1 \neq$ true in $\text{split}_{[v=\text{header}(h)]}(p) = (p_1, p_2)$, then this expression can be rewritten to:

for $$v$ **in** $\mathcal{SQL}$***[*** h, ***from*** f ***where*** p' ***and*** p_1 ***order by*** o ***]***
where p_2 **return** e

which promotes p_1 inside the SQL 'where' condition.

We now focus on the following XQuery expression:

for $$v_1$ **in** $\mathcal{SQL}$***[*** h_1, q_1 ***]***
where p_1
return F(**for** $$v_2$ **in** $\mathcal{SQL}$***[*** h_2, q_2 ***]***
where p_2 **return** e_2)

where F is an arbitrary XQuery expression that contains this inner for-loop (it is not a function call). If $p_2^1 \neq$ true in $\text{split}_{[v_1=\text{header}(h_1), v_2=\text{header}(h_2)]}(p_2) = (p_2^1, p_2^2)$, then this expression can be rewritten into a for-loop over a single SQL query that uses, in general, a left-outer join, followed by an implicit grouping based on the header pattern:

for $$v_1$ **in** $\mathcal{SQL}$**[** row: (x.*, nest: (y.*)),
from s_1 x **left outer join** s_2 y **on** p_2^1
order by x.id **]**
where p_1
return F(**for** $$v_2$ **in** v_1/nest
where p_2^2 **return** e_2)

where s_1 and s_2 are the SQL queries for $\mathcal{SQL}$***[*** h_1, q_1 ***]*** and $\mathcal{SQL}$***[*** h_2, q_2 ***]*** respectively, without any implicit grouping, using header(h_1) and header(h_2) as SQL headers.

IX. Implementation

We have already built a prototype XQuery engine, called HXQ, that implements our view update framework. It is available at `http://lambda.uta.edu/HXQ/`.

HXQ is a fast and space-efficient translator from XQuery to embedded Haskell code. It takes full advantage of Haskell's lazy evaluation to keep in memory only those parts of XML data needed at each point of evaluation, thus performing stream-based evaluation for forward queries (queries that do not contain backward steps). This results to an implementation that is as fast and space-efficient as any stream-based implementation based on SAX filters or finite state machines. Furthermore, the coding is far simpler and extensible since it is based on XML trees, rather than SAX events. Since HXQ uses lazy evaluation, you get the first results of non-blocking queries immediately, while the non-streaming XQuery processors must first parse the entire input file and construct the whole XML tree in memory before they produce any output.

Lazy functional languages process lists in a stream-like fashion: list elements are often processed one-element-at-a-time, all the way from the point they are generated (such as when they are formed by a document parser) up to the point the result is printed. Computations that do not contribute to the final result are suspended as thunks in the heap and are garbage-collected. The HXQ coding combines a careful mix of strict and lazy evaluation, so that the large lists are unfolded lazily, while the code walks over them strictly. The result is a program that runs in constant space for many queries and is very fast. Although it is not that difficult to simulate lazy list processing in other programming languages, it is very hard to do so for tree-like data. Haskell can process XML data as effectively as regular lists because XML data are "mostly flat" trees (their nesting level is typically in the dozens while the total number of elements can be arbitrary large).

For instance, the following XQuery:

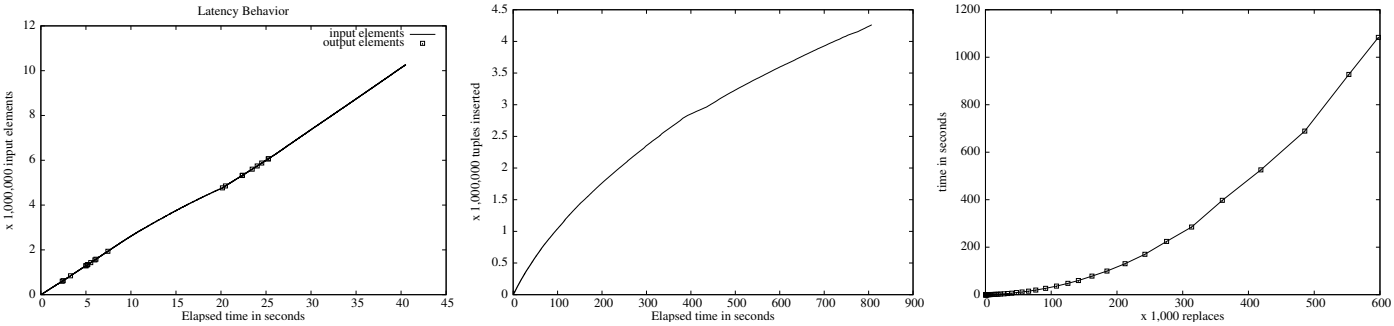


Fig. 3. A. Timeline of I/O events B. Tuples inserted when shredding DBLP C. Evaluating a large number of updates

```
<result>{ for $x at $i in doc('dblp.xml')//inproceedings
          where $x/author = 'John Smith'
          return <paper>{ $i, $x/booktitle/text(),
                          ':', $x/title/text()
                        }</paper>}</result>
```

which is against the DBLP XML document of size 420MB, runs in 36 seconds and uses a maximum of 3.2MB of heap space. (All results reported on this section are taken on a 2.2GHz Intel Core 2 Duo with 2GB memory running Haskell ghc-6.8.3 on a 32-bit Linux 2.6.27 kernel.) To contrast this, Qexo, which compiles XQueries to Java bytecode, takes 1 minute and 17 seconds and uses 1400MB of heap space for the same query, while XQilla, which is written in C++, takes 1 minute and 10 seconds and uses 1150MB of heap space. Figure 3.A shows the timeline of all input and output start-tag events, which illustrates the stream-like behavior of HXQ.

In addition to processing XML files, HXQ can store XML documents in a relational database (currently SQLite or MySQL through ODBC), by shredding XML into relational tuples, and by translating XQueries over the shredded documents into embedded optimized SQL queries. The mapping to relational tables is based on the document's structural summary, which is derived from the document data rather than from a schema. It uses hybrid inlining to inline attributes and non-repeating elements into a single table, thus resulting to a compact relational schema. For each such mapping, HXQ synthesizes an XQuery that reconstructs the original XML document from the shredded data. This XQuery is fused with the user queries using partial evaluation techniques and parts of the resulting query are mapped to SQL queries, using the methods described in this paper.

Figure 3.B shows the total number of tuples inserted during the shredding of the DBLP XML document into a MySQL database. Near the end of shredding, the insertion rate is 207K tuples per millisecond. After shredding the DBLP document and after creating a secondary index on author, the previous XQuery takes about 90 milliseconds in MySQL.

To evaluate the scalability of our updates through views, we run the following update:

```
for $i in $view//inproceedings
where $i/year lt Y
return replace $i/year with $i/year
```

for each year Y ranging between 1963 and 2008. Of course, the same update can be done in SQL using:

```
update Inproceedings set year=year where year <Y
```

because this is a special query that does not need any XQuery functionality. In fact, our framework will generate the following XQuery:

```
for $i in $sql
return db:replace( "Inproceedings.year", $i/year/data(),
                   getIds($i), getPath($i) )
```

where $sql is

```
SQL[ row: (i.year),
     from Inproceedings i where i.year <Y ]
```

We can see that, for a particular year Y, if there are n inproceedings with a publication year <Y, then there will be one query that retrieves all n ids followed by n updates, one for each id. Figure 3.C shows that the performance of our system degrades very fast when the number of updates increases, which indicates that this approach of handling each update destination id separately is not suitable for massive updates. On the other hand, in order to handle general updates, where the update source can be any XQuery expression, we have to do the updates one tuple at a time.

X. Conclusion and Future Work

The most important characteristic of our view update approach is that most work is done at compile-time. Only the propagation of some lineage annotations is done at runtime, requiring a minimal overhead. Nevertheless, there is much room for improvement. Our framework (as well as the related work) is based on the assumption that we need to first retrieve the data to be updated and then reflect the updates to these data back to the source using the data ids. Although the data retrieval can be done using just one SQL query in most cases, the actual updates must be done one-tuple-at-a-time. Consequently, as it was apparent from our evaluation study, some XQuery updates may result to a large number of SQL updates that could have been expressed as one if they were written by hand. In the future, we would like to find optimization rules that will fuse these SQL updates with the SQL retrieval queries, resulting into a single query (when this is possible). In addition, we would like to apply our framework to other data types. Relational views expressed in SQL do not fit easily in our framework because SQL allows unrestricted view graphs, which do not provide an update context, as navigation paths in view trees do. XML views from native

XML storage to XML, on the other hand, may be very well-suited to our framework as long as there is an XML Schema that describes the XML storage.

REFERENCES

[1] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *TODS'81*, 6(4).

[2] T. Barsalou, A. M. Keller, N. Siambela, and G. Wiederhold. Updating Relational Databases through Object-Based Views. In *SIGMOD'91*.

[3] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *VLDB'06*.

[4] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB'04*.

[5] M. Blow, *et al*. Updates in the AquaLogic Data Services Platform. In *ICDE'09*.

[6] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. PATAXO': A Framework to Allow Updates Through XML Views. In *TODS'06*, 31(3).

[7] V. P. Braganholo, S. B. Davidson, and C. A. Heuser. From XML View Updates to Relational View Updates: old solutions to a new problem. In *VLDB'04*.

[8] P. Buneman, J. Cheney, and S. Vansummeren. On the Expressiveness of Implicit Provenance in Query and Update Languages. In *ICDT'07*.

[9] B. Choi, G. Cong, W. Fan, and S. D. Viglas Updating Recursive XML Views of Relations. In *ICDE'07*.

[10] Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. In *TODS'00*, 25(2).

[11] A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *SIGMOD'99*.

[12] U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. In *TODS'82*, 7(3).

[13] M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A Framework for Publishing Relational Data in XML. In *TODS'02*, 27(4).

[14] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *VLDB'07*.

[15] Y. Kotidis, D. Srivastava, and Y. Velegrakis. Updates Through Views: A New Hope. In *ICDE'06*.

[16] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? In *VLDB'04*.

[17] B. C. Pierce. Types and Programming Languages. The MIT Press, 2002.

[18] J. Simeon, and P. Wadler. The essence of XML. In *POPL'03*.

[19] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD'01*.

[20] J. Voigtlander. Bidirectionalization for Free! In *POPL'09*.

[21] L. Wang, M. Jiang, E. A. Rundensteiner, and M. Mani. An Optimized Two-Step Solution for Updating XML Views. In *DASFAA'08*.

[22] L. Wang, E. A. Rundensteiner, M. Mani, and M. Jiang. HUX: Handling Updates in XML. In *VLDB'06*.

[23] K. Wansbrough, and S. P. Jones. Once Upon a Polymorphic Type. In *POPL'99*.

[24] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics. `http://www.w3.org/TR/xquery-semantics/`, 2007.

[25] W3C. XQuery Update Facility 1.0. W3C Candidate Recommendation 1. `http://www.w3.org/TR/xquery-update-10/`, 2008.

APPENDIX

A proof that polymorphic type inference deduces the EDS constraint: To simplify the proof, we assume that the view is parameterized by only one database column domain. That is, the type of the $view$ function is $\forall\alpha.\ \mathrm{DB}(\alpha) \rightarrow \mathrm{XML}(\alpha)$, where both the database schema, $\mathrm{DB}(\alpha)$, and the view output type, $\mathrm{XML}(\alpha)$, are parameterized by this column domain. Any polymorphic function satisfies a parametricity theorem (aka, theorem for free), derived exclusively from the type signature of the function (Section 23 in [17]). For $view$, the theorem is:

$$\forall f:\ \mathrm{map}_X(f) \circ view = view \circ \mathrm{map}_D(f) \qquad (1)$$

where $\circ$ denotes function composition, and map_X of type $\forall\alpha,\beta.\ (\alpha \rightarrow \beta) \rightarrow \mathrm{XML}(\alpha) \rightarrow \mathrm{XML}(\beta)$ and map_D of type $\forall\alpha,\beta.\ (\alpha \rightarrow \beta) \rightarrow \mathrm{DB}(\alpha) \rightarrow \mathrm{DB}(\beta)$ are the map functions for the $\mathrm{XML}(\alpha)$ and $\mathrm{DB}(\alpha)$ types, respectively. Eq. (1) indicates that it does not matter if we change the α values before or after the view since the $view$ is oblivious to these values. Consider now the following two functions:

$$\begin{aligned}&\mathrm{enum}_D:\ \forall\alpha.\ \mathrm{DB}(\alpha) \rightarrow \mathrm{DB}(\mathrm{int} \times \alpha)\\ &\mathrm{enum}_X:\ \forall\alpha.\ \mathrm{XML}(\alpha) \rightarrow \mathrm{XML}(\mathrm{int} \times \alpha)\end{aligned}$$

that assign unique integers to the α values. Note that this value labeling is purely theoretical, since the value domains of DB and XML cannot be replaced by pairs of value-integers. In our framework, these unique integers are part of the lineage (the tuple ids) that annotate atomic values of type α, but any labeling with unique integers would be sufficient. Irrespectively of its functionality, enum_X satisfies a parametricity theorem:

$$\forall f:\ \mathrm{map}_X(\mathbf{1} \times f) \circ \mathrm{enum}_X = \mathrm{enum}_X \circ \mathrm{map}_X(f) \qquad (2)$$

where $\mathbf{1}$ is the identity function. Consider a database update that replaces an α value, v, given its integer labeling, id:

$$\mathrm{update}_D(id, v) = \mathrm{map}_D(U) \qquad (3)$$

where $U(i, x) = (i, \textbf{if } i = id \textbf{ then } v \textbf{ else } x)$. Consider also an XML update that replaces the α value in the XML tree that is numbered pos in XML and id in DB (if exists):

$$\mathrm{update}_X(pos, id, v) = \mathrm{map}_X(U') \qquad (4)$$

for $U'(j, (i, x)) = (j, (i, \textbf{if } i = id \wedge j = pos \textbf{ then } v \textbf{ else } x))$. To prove the EDS constraint, we need to prove $\forall pos, id, v$:

$$\begin{aligned}&\mathrm{update}_X(pos, id, v) \circ \mathrm{enum}_X \circ view \circ \mathrm{enum}_D\\ &= \mathrm{enum}_X \circ view \circ \mathrm{update}_D(id, v) \circ \mathrm{enum}_D\end{aligned}$$

that is, a single update on the XML tree on the XML position pos with a DB id, id, causes the same results as a single update in the database on the DB id, id. To prove it, we rewrite the right hand side:

$$\begin{aligned}&\mathrm{enum}_X \circ view \circ \mathrm{update}_D(id, v) \circ \mathrm{enum}_D\\ &= \mathrm{enum}_X \circ view \circ \mathrm{map}_D(U) \circ \mathrm{enum}_D && \text{from (3)}\\ &= \mathrm{enum}_X \circ \mathrm{map}_X(U) \circ view \circ \mathrm{enum}_D && \text{from (1)}\\ &= \mathrm{map}_X(\mathbf{1} \times U) \circ \mathrm{enum}_X \circ view \circ \mathrm{enum}_D && \text{from (2)}\end{aligned}$$

Given Eq. (4), we need to prove:

$$\begin{aligned}&\mathrm{map}_X(U') \circ \mathrm{enum}_X \circ view \circ \mathrm{enum}_D\\ &= \mathrm{map}_X(U'') \circ \mathrm{enum}_X \circ view \circ \mathrm{enum}_D\end{aligned} \qquad (5)$$

where $U'' = \mathbf{1} \times U$, that is:

$$U''(j, (i, x)) = (j, (i, \textbf{if } i = id \textbf{ then } v \textbf{ else } x))$$

Comparing it to the U' in Eq. (4), we can see that the theorem is true $\forall pos, id, v$ if every position pos in the XML tree has a unique id. This is exactly the uniqueness property, which is detected by the usage analysis in our type inference.