Incremental Evaluation of Schema-directed XML Publishing

Citation for published version:

Digital Object Identifier (DOI):
10.1145/1007568.1007625

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Publisher's PDF, also known as Version of record

Published In:
Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data

Publisher Rights Statement:
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Abstract

When large XML documents published from a database are
maintained externally, it is inefficient to repeatedly recompute them when the database is updated. Vastly preferable
is incremental update, as common for views stored in a data
warehouse. However, to support schema-directed publish-
ing, there may be no simple query that defines the mapping
from the database to the external document. To meet the
need for efficient incremental update, this paper studies two
approaches for incremental evaluation of ATGs [4], a formal-
ism for schema-directed XML publishing. The reduction ap-
proach seeks to push as much work as possible to the under-
lying DBMS. It is based on a relational encoding of XML trees
and a nontrivial translation of ATGs to SQL 99 queries with
reursion. However, a weakness of this approach is that it
relies on high-end DBMS features rather than the lowest com-
mon denominator. In contrast, the bud-cut approach pushes
only simple queries to the DBMS and performs the bulk of
the work in middleware. It capitalizes on the tree-structure
of XML views to minimize unnecessary recomputations and
leverages optimization techniques developed for XML pub-
lishing. While implementation of the reduction approach is
not yet in the reach of commercial mass, we have implement-
ed the bud-cut approach and experimentally evaluated its
performance compared to recomputation.

1. Introduction

XML publishing by middleware [11, 8, 16] or with direct
DBMS support [7] has been well studied, and techniques
from this work are rapidly being introduced into commercial
products [25, 28]. In some applications, small portions of
a database are extracted into “disposable” XML documents,
for example the messages needed to execute or respond to
requests using a web-services protocol. However, in many
applications including mediation, archiving and web site man-
agement, large XML documents may need to be exported. In
this case, the cached XML document can obviously be seen
as a view of the database instance. For all the reasons that
efficient incremental maintenance of views in the database
(see, e.g., [14]) is important, it may also make sense to in-
crementally update published XML documents, even when they
are externally cached by a middleware system. However, to

*Supported in part by an Earmarked Research Grant.
†Supported in part by NSF Career Award IIS-0093168, NSFC
60228006 and EPSRC GR/S63265/01.

Permission to make digital or hard copies of all or part of this work
for personal or classroom use is granted without fee provided that
copies are not made or distributed for profit or commercial advantage
and that copies bear this notice and the full citation on the first page.
To copy otherwise, to republish, to post on servers or to redistribute
to lists, requires prior specific permission and/or a fee.

Copyright 2004 ACM 1-58113-839-8/04/06 ...$5.00.
and for updating the external view $T$ with $\Delta T$.

Previous work [11, 8] for XML publishing has developed mappings from XML view queries to SQL queries that compute root-to-leaf paths in a relational encoding of the XML trees. These techniques however, cannot be directly applied to generating or updating AVG-generated documents due to a lack of support for recursion. In the context of XML shredding, the authors of [17] showed that linear recursion of SQL 99 is sufficient to support XQuery queries over shredded XML data, even when the shredding schema is recursive, and suggested that this approach could be applied for publishing queries. The reduction framework extends the prior work by showing the connection between the recursive XML views needed for schema-directed publishing and SQL 99 views.

Much as [11, 8] seek to push XML publishing work to the DBMS, a primary goal of the reduction framework is to push incremental work to the DBMS, thereby taking advantage of sophisticated capabilities for query optimization, execution and incremental view update. However, three practical issues complicate the use of DBMS resources in support of incremental work. The first issue involves the DBMS features required for the reduction approach. Middleware that seeks to work with a wide variety of products supporting the ODBC interface must take a "lowest common denominator" approach to the functionality required from the DBMS. But SQL 99 recursion and incremental update of views are separate, advanced features of only the most sophisticated commercial products; furthermore the reduction approach also depends on "incremental update of materialized views defined using SQL 99 recursion; in other words, a combination of both features. Second, to effectively push down the work required to incrementally update an external view, one must have access not only to a materialized view, but more importantly, to a stream of updates to that view. One way to obtain this functionality would be to define triggers on the materialized view, but this is disallowed by at least one commercial DBMS with materialized view support, and needs not be supported in general for the DBMS to function well.

Finally, if the publishing queries are even mildly complex, the combined recursive queries may become extremely complex. As a result, they may not be effectively optimized by all platforms supporting with...recursive for the same reasons that not all DBMS platforms can effectively optimize complex non-recursive publishing queries [11].

In response to this, our second contribution is to propose an alternative approach, referred to as the 'bud-cut approach' to incremental ATG evaluation, that requires less sophistication from the DBMS. Further, we develop certain optimization techniques which capitalize on the tree-structure of XML views. The bud-cut mechanism propagates relational changes to XML in three phases: generation, completion and garbage collection. The bud-cut generation phase determines the impact of $\Delta I$ on existing parent-child relations in the XML view, i.e., insertions (buds) and deletions (cubs), by evaluating a fixed number of materialized SQL queries. Following this, the bud completion phase iteratively computes newly inserted subtrees top-down by pushing SQL queries to the relational engine. Finally, deleted subtrees are removed by a garbage collection process.

The bud-cut approach has several properties. a) $T$ can be updated in parallel with ongoing computation of $\Delta T$ during bud completion. b) It minimizes unnecessary recomputations via a caching strategy not considered in prior work for maintaining recursive views such that each new subtree in the XML view is computed at most once no matter how many times it occurs in the XML view, and furthermore the computation maximally reuses subtrees of the old XML view. c) It incorporates optimization techniques that have proved effective in XML publishing but are not supported by DBMSs, e.g., query merging [11]. d) Since the tree is computed level-by-level in this phase, it is possible to return partial results to a user navigating the tree while computation is ongoing, and such computation can even be deferred according to a lazy evaluation strategy [7]. e) It does not require materialization of the view in the DBMS. Finally, of course the bud-cut approach does not require the DBMS to support either SQL 99 or incremental view updates.

We have implemented the bud-cut approach. We use the implementation to investigate the impact of $|I|$ and $|\Delta I|$ on the performance of incremental update, as well as the improvement obtained over full recomputation for small updates. Further, we investigate the impact on performance of the subtree reuse optimization mentioned above, and find that its impact is greatest when only a portion of $I$ is published in $T$, and when there is a moderate degree of locality in the updates appearing in $\Delta I$. Unfortunately but not surprisingly, the reduction approach is unrealistic since current commercial relational systems do not implement incremental update of recursive queries.

The algorithms and the bud-cut mechanism can be extended to accommodate multiple data sources, i.e., for XML integration studied by [8], which is a generalization of ATGs by supporting multi-source SQL queries and XML constraints. They can also be used for incremental maintenance of XML views generated by other systems, such as [11, 8].

Organization. Section 2 reviews ATGs. Section 3 describes data structures for external XML views. Section 4 provides the reduction approach, followed by the bud-cut approach in Sect. 5. Section 6 presents experimental results. Section 7 addresses related work and Sect. 8 concludes the paper.

2. Background

In this section we first introduce a running XML publishing example used in the rest of the paper. We then review DTDs and present a refinement of ATGs as defined by [4].

Example 2.1: Consider a registrar database specified by the relational schema $R_0$ below (with keys underlined):

- course(cno, title, dept). project(cno, title, dept)
- student(sno, name)
- enroll(sno, cno).
- prereq(cno, cno2).

The database maintains student data, enrollment records, course data classified into regular courses and projects, and a relation prereq which gives the prerequisite hierarchy of courses where a tuple $(c_1, c_2)$ in prereq indicates that $c_2$ is a prerequisite of $c_1$.

The office of registrar maintains an XML view for the CS department, which contains data of CS courses registration, extracted from the registrar database. The view is required to conform to the DTD $D_0$ below (the definition of elements whose type is PCDATA is omitted):

```xml
<ELEMENT db (course*)>
<ELEMENT course (cno, title, type, prereq, takenby)>
<ELEMENT type (regular | project)>
<ELEMENT prereq (course*)>
<ELEMENT takenby (student*)>
```
The idea of attribute translation grammars (ATGs) is to treat the DTD as a grammar and recursively fire productions from the grammar to create an XML document. We now briefly review the syntax and semantics of ATGs (see [4]).

An ATG $\sigma : R \Rightarrow D$ specifies a mapping from instances of the source relational schema $R$ to documents of the target DTD $D$ as follows: a) For each element type $A$ of $D$, $\sigma$ specifies a semantic attribute $\$A$ whose value is a single relational tuple of a fixed arity and type; intuitively, $\$A$ controls the generation of $A$ elements in the XML view, and is used to pass data downward as the document is produced. b) For each production $p = A \Rightarrow \alpha$ in $D$, $\sigma$ specifies a set of semantic rules, $\text{rule}(p)$. These rules specify the computation of the $B$ children of an $A$ element for each type $B$ in $\alpha$.

Given a database $I$ of $R$, the ATG $\sigma$ is evaluated top-down starting at the root $r$ of $D$. A partial tree $T$ is initialized with a single node of type $r$, and this node is marked unexpanded; we refer to unexpanded nodes as buds. The tree $T$ is then grown by repeatedly selecting a bud $B$ (of some element type $A$), evaluating the semantic rules associated with $A$, and marking $B$ expanded.

Specifically, we find the production $p = A \Rightarrow \alpha$ in $D$, and generate the children of $B$ by evaluating $\text{rule}(p)$ and using the value of the attribute $\$A$ of $B$. The rules $\text{rule}(p)$ are defined and evaluated based on the form of $\alpha$ as follows:

1) If $\alpha$ is $B_1 : \ldots : B_n$, then a node tagged $B_i$ is created for each $i \in [1,n]$ as a child of $B$. The tuple value of $\$B_i$ associated with the new $B_i$ child is determined by projection from $\$A$. That is, $B_i = (\$A_i : \ldots : \$A_n)$ is in rule $(p)$ for $i \in [1,n]$, where $A_i$ is a field of the tuple $\$A$.

2) If $\alpha$ is $B_1 + \ldots + B_n$, then rule $(p)$ is defined by $(\$B_1, \$B_2, \ldots, \$B_n) = \text{case } f(\$A) \text{ of}$

$$
1: (\$A, \text{null}, \ldots, \text{null})
$$

$$
n: (\text{null}, \ldots, \text{null}, \$A)
$$

where $f$ is a function that maps $\$A$ to natural numbers in $[1,n]$. That is, based on the conditional test, a node is created for exactly one child, $B_i$. The value of the parent attribute $\$A$ is passed down to that child. No $B_i$ child is created if $i \neq j$, and $\$B_i$ (the special value null) is ignored. We assume that the function $f$ is simple enough to determine whether it is in the range $[1,n]$.

3) If $\alpha$ is $B^*$, then rule $(p)$ is defined by $\$B \leftarrow \text{Q}(\$A)$, where $Q$ is an $\text{xml}$ query over a database of $R$, and it treats $\$A$ as a constant parameter. For each distinct tuple $t$ returned by $Q(\$A)$, a $B$ child is generated, carrying $t$ as the value of its $\$B$ attribute. To help ensure that only finite documents are created, only references to attributes and constants, but not expressions are allowed in the select-list of $Q$.

4) If $\alpha$ is $\text{str}$, then the rule specifies formatting of the values of $\$B$ for presentation (string/PCDATA). Such rules are not shown or discussed further.

5) If $\alpha$ is $\epsilon$, then no rule is defined and no action is taken.

The element children of the node $b$ become new buds and are also processed. The process proceeds until the partial tree cannot be further expanded. i.e., it has no unexpanded node. The fully expanded XML tree do not include attribute values $\$A$, which are only used to control the tree generation.

Example 2.2: The ATG $\sigma_0$ given in Fig. 3 defines the XML view described in Example 2.1. Here rule $(\text{course})$. 

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{xml_diagram.png}
\caption{XML view}
\end{figure}
db → course
$course ← Q1
Q1:  
\text{select distinct c.cno, c.title, 1 as tag from course c where c.dept = 'CS'}$
\text{union}
\text{select distinct p.cno, p.title, 2 as tag from project p where p.dept = 'CS'}
course → cno, title, type, prereq, takenBy
$\text{cno} = \text{Source.cno}, \text{title} = \text{Source.title},$
$\text{Stype} = \text{Source.tag}, \text{prereq} = \text{Source.cno},$
$\text{StakenBy} = \text{Source.cno}$
type → regular + project
$(\text{regular}, \text{Project}) = \text{case type of 1: (Stype, null)}$
$2: (null, Stype)$
prereq → course
$\text{source} ← Q2(\text{prereq})$
$Q2(c):  
\text{select distinct c.cno, c.title, 1 as tag from prereq p, course c where p.cno = c and p.cno = c.cno}$
takenBy → student
$\text{student} ← Q3(\text{StakenBy})$
$Q3(c):  
\text{select distinct s.ssn, s.name from enroll e, student s where e.cno = c and e.ssn = s.ssn}$

Figure 3: Example ATG $\sigma$
rule(type) and rule(prereq) illustrate the cases (1), (2), (3) above. Given a database, $\sigma$ computes an XML view as follows. It first generates the root element (with tag db), and then evaluates the query $Q1$ to extract courses and projects of the CS department from the underlying database. For each distinct tuple $c$ in the output of $Q1$, it generates a course child $v_c$ of $db$, which is a bud carrying $c$ as the value of its attribute $\text{cno}$. The subtree of the bud $v_c$ is then generated by using $c$. Specifically, it creates the cno, title, type, prereq and takenBy children of $v_c$, carrying c.cno, c.title, c.tag, c.cno and c.cno as their attributes respectively. It then proceeds to create a text node carrying c.cno as its PCDATA, as the child of the cno node; similarly for title. It determines the type of $v_c$ by examining c.tag; if it is 1 then a regular child of type is created; otherwise a project child is generated. It creates the children of the prereq node by evaluating the SQL query $Q2$ to find prerequisites of the course, and again for each tuple in the output of $Q2$ it generates a course node; similarly it constructs the takenBy subtree by evaluating $Q3$ to extract student data. Note that $Q2$ and $Q3$ take c.cno as a constant parameter. Since course is recursively defined, the process proceeds until it reaches courses that do not have any prerequisites i.e., when $Q2$ returns empty at the prereq children of those course nodes. That is ATG handle recursion following a data-driven semantics. When the computation terminates the ATG generates an XML view as depicted in Fig 2, which conforms to the DTD $D_0$ given in Example 2.1.

As observed by [4], ATGS are more expressive than the view definition languages of previous publishing systems [11, 8].

Exception Handling. ATGS are defined as defined in the original definition introduced in [4]. An ATG of [4] may abort over a relational database, i.e., it may terminate unsuccessfully as the XML view it generates may violate the given DTD. In contrast, our revised ATGS do not abort. However, even our revised ATGS defined over a recursive DTD may not terminate. For example, $\sigma$ of Fig. 3 may not converge if the relation prereq in the underlying database is cyclic, e.g., if a course is its own prerequisite. Worse still, it is undecidable [4] to determine at compile time for an arbitrary ATG $\sigma : R \rightarrow D$, whether $\sigma$ terminates on all databases of $R$ [4].

To cope with this we introduce an exception handling mechanism. For an element type $A$ defined recursively, we consider a mild extension of its production $p = A \rightarrow a$, namely, $p' = A \rightarrow a + \text{str}$. For example, we extend the production of course in the DTD $D_0$ to be $\text{course} \rightarrow (\text{cno, title, type, prereq, takenBy}) + \text{str}$

In order to ensure termination, we modify tree generation to stop expanding the tree if a newly created node $v_c$ has the same type and semantic attribute value as one of its ancestors. In this case, we simply emit the string value of $v_c$'s attribute as the contents of the new node. It should be mentioned that this does not lose information, since the subtree of $v_c$, if constructed by following the original production $p = A \rightarrow a$, is just a copy of the subtree at $v$ and does not introduce any new information. This process is referred to as exception handling. Although exception handling slightly modifies the DTD embedded in an ATG, it ensures termination of ATG evaluation without loss of information.

Theorem 2.1: Let $\sigma : R \rightarrow D$ be an arbitrary ATG with exception handling. Then over all instances $I$ of $R$, $\sigma$ terminates and $\sigma(I)$ conforms to the DTD $D$.

Theorem 2.1 shows that an ATG $\sigma : R \rightarrow D$ is actually a total function: given a database $I$ of $R$, $\sigma(I)$ computes an XML document of $D$ that is unique up to reordering of $B$-elements produced by productions of the form $A \rightarrow B^*$. In the sequel we consider only ATGS with exception handling and refer to them simply as ATGS. Note that our incremental techniques also work on ATGS without exception handling.

3. External Trees

Incremental update makes sense only in the context of an externally maintained tree available to the client. A variety of potential implementations for the external tree exist, from a native storage system like Berkeley XML DB to an in-memory implementation of DOM. While in the current work we have used our own implementation of an in-memory tree in C++, we expect the data structures and algorithms we propose to apply to other implementations.

We next describe the external data structures maintained by our middleware to accept and process changes.

Node Identity. We assume that we can associate a compact, unique value with each tuple value taken on by a semantic attribute in $\sigma(I)$. We abstract away the implementation of this identity value by assuming, without loss of generality, the existence of a Skolem function gen_id (see, e.g., [9]) that, given the tuple value of a semantic attribute $A$, computes id(A) that is unique among all identities associated with all semantic attributes (for example, it might encode the type and a unique value within that type).

Tree vs. Graph Representations. An important property of an ATG $\sigma : R \rightarrow D$ is that, for any database $I$ of $R$ and type $A$ of $D$, an $A$-element (subtree) $T_A$ in the XML view $\sigma(I)$ is uniquely determined by the value of the semantic attribute $A$ at the root of $T_A$. Thus the ATG defines a function $\text{str}$ such that, given an element type $A$ and a value $t$ of $\text{str}(A, t)$ returns a subtree rooted at a node tagged $A$ and carrying $t$ as its attribute.
Since a subtree \( st(A, S, A) \) may appear at different places in the XML view \( \sigma(I) \), if the middleware system is managing the external view, it may be more efficient to represent \( \sigma(I) \) as a graph such that a single copy of \( st(A, S, A) \) is stored and shared by its multiple occurrences. Indeed, if \( \sigma(I) \) is stored as a tree, for example by using an available implementation of DOM [2], the tree may be exponentially larger than the graph representation. In light of this the current implementation of ATGS adopts the graph representation, but supports client navigation on the graph as if it were a tree. The trade-off of the space efficiency is that the support of user navigation is complicated, as the path from a particular node to the root of the tree would be dependent on the route navigated; further, a mechanism must be provided to check for duplicates along this path to provide the exception handling semantics described above.

**Data Structures.** The external data structures used to represent the XML document are depicted in Fig. 4. The tree \( T \) is stored in a hash index \( H \) and a subtree pool \( S \). Each entry of \( H \) is of the form \(( A, \text{id}_A, \text{ptr} )\), where \( A \) is an element type, \( \text{id}_A \) is the unique id of a value of \( A \), and \( \text{ptr} \) is a pointer to the root node of the subtree \( st(A, S, A) \) in \( S \). The subtree pool \( S \) consists of entries \(( A, \text{id}_A, L )\), where \(( A, \text{id}_A \) represents a node \( v \) of \( T \), and \( L \) is a list \([ ( B_1, \text{id}_B_1 ), \ldots , ( B_n, \text{id}_B_n ) ]\) representing the children of \( v \) such that each \(( B_i, \text{id}_B_i )\) is an entry in \( H \). Observe that in the graph representation, there is a one-to-one mapping from \( H \)-entries to the nodes in \( S \).

**Handling Updates.** In this paper, we allow \( \Delta I \) to be any group of updates to the underlying rosses that preserves the consistency of the database (integrity constraints). XML updates \( \Delta I \) generated from \( \Delta I \) by one of the techniques described in the next two sections are represented as \(( E^+, E^-) \), where \( E^+ \) is a set of edges to be inserted into the tree \( T \), and \( E^- \) is a set of edges to be deleted from \( T \). The edges are represented as \(( \text{id}_A, \text{id}_B )\), where \( \text{id}_A \) and \( \text{id}_B \) are the ids of the parent \( A \)-element and the child \( B \)-element respectively.

The system processes insertions as follows for each tuple \(( \text{id}_A, \text{id}_B )\) in \( E^+ \): 1) find the \( H \)-entry \(( A, \text{id}_A, \text{ptr} )\) and the pointer to the node \(( A, \text{id}_A, L )\) in the subtree pool \( S \); 2) insert \(( B, \text{id}_B )\) into \( L \) if it is not already in \( L \); and 3) if \( H \) does not have an entry \(( B, \text{id}_B )\) in \( H \), create an entry in \( H \) and a node \(( B, \text{id}_B, [ ] )\) in \( S \) with an empty child-list \([ ] \).

**Pushing Incremental Work to the DBMS.**

In this section we present our reduction mechanism for incremental evaluation of ATGS. The approach is based on 1) a relational encoding of XML trees via a set of (interrelated) virtual attribute and edge relations and 2) a translation of an ATGS \( \sigma \) : \( R \rightarrow D \) to a set \( MQ_s \) of SQL 99 queries utilizing the with..recursive construct, which computes the attribute and edge relations of the XML views defined by \( \sigma \).

Given this the reduction approach to incrementally maintaining an XML tree \( T \) computed by ATGS \( \sigma(T) \) works as follows: 1) encode \( T \) with the attribute and edge relations, 2) map \( T \) to SQL 99 queries \( MQ_s \), 3) define an incrementally updated materialized view in the source DBMS for each query of \( MQ_s \), 4) in response to relational updates \( \Delta I \), utilize the newly functionality to capture the incremental changes made to each of the materialized views which can be directly transformed into the XML changes \( \Delta T \) (i.e., \( E^+ \) and \( E^- \)), and 5) propagate the changes \( \Delta T \) to the external trees as described in the previous section. This leads to a convenient approach to incremental evaluation of ATGS by pushing as much work as possible to the underlying DBMS.

We next focus on the relational encoding of XML trees using ATGS to SQL 99 queries.

### 4.1 A Relational Encoding of XML Trees

An XML view \( \sigma(I) \) is encoded via node (semantic attribute) and edge relations.
Attribute Relations. The attribute relations are to capture the values taken on by the semantic attributes defined in $\sigma(I)$. To avoid confusion, “attributes” of relational tables will be uniformly referred to as “columns”.

Recall that $\sigma$ associates with each element type $B$ a tuple

formalized variable $B$. For each such variable $B$, let $\text{gen}_B$ be a relation with columns matching the arity and type of $B$, along with a column for id$_B$ if an existing group of attributes does not serve this role. Further, in the context of a database instance $I$, assume that $\text{gen}_B$ is populated with all the (non-null) values taken on by $B$ during an evaluation of $\sigma$ on $I$.

We define each attribute relation, $\text{gen}_A$, in terms of a query $Q_{\text{gen}_A}$ involving the other attribute relations and the relations of $I$. To define $Q_{\text{gen}_A}$ we first rewrite sqs queries embedded in $\sigma$ to queries that take $\text{gen}_A$ instead of a single tuple $A$, as a parameter. Specifically, consider productions $A \rightarrow a$ in which $B$ appears on its right-hand side (rhs).

(1) For productions of the form $A \rightarrow B^+$ with associated semantic rule $B \leftarrow Q(A)$, $\text{gen}_B$ is the union, over all values of $A$ in $\text{gen}_A$ of $Q(A)$. In a manner similar to the

level-at-a-time processing of [4], this query can be rewritten as $Q^*$, which takes no parameters but additionally accesses $\text{gen}_A$. This is accomplished, roughly by a) adding $\text{gen}_A$ to the FROM list of $Q$, and b) replacing references to $A$ in $Q$ with the corresponding references to $\text{gen}_A$.

(2) If $B$ appears as some $B_i$ when $a$ is $B_1, \ldots, B_n$, then $Q^*$ is a simple selection query from $\text{gen}_A$ that projects fields according to $f_i(A)$ (see Section 2).

(3) If $B$ appears as some $B_i$ in $a$ is $B_1 + \ldots + B_n$ then $Q^*$ can be written as $\text{select} *$ from $\text{gen}_A$ where $f(a) = i$. Note that we assume further that $f(a)$ is computable in the dialect of sqs used to express $Q^*$.

It is now simple to generalize this construction to handle the case where $B$ appears on the rhs of multiple rules in $\sigma$. Suppose that it appears on the rhs of rule $p_1, p_2, \ldots, p_m$ and that $Q_i$ is defined per the discussion above. Then $Q_{\text{gen}_A}$ is formed by taking the (distinct) union of all $Q_i$ queries.

For example, Fig. 5 shows the definition of two attribute-relations generating queries when the resulting construction is applied to the rhs of $\sigma$.

We denote by $\text{AR}(\sigma)$ the set of all $Q_{\text{gen}_A}$ queries for $\sigma$.

Maintaining Edges. We next describe how to capture the edge relations of the.xml view. Let $\text{edge} \rightarrow B$ be a relation with two columns, id$_A$ and id$_B$. We create such a relation if $B$ appears on the rhs of some production for $A$. We overload the Skolem function $\text{gen}_A$ described in Sect. 3 to compute the unique id $a_i$ from a relational tuple $a$.

We now discuss how to derive queries to define these edge relations in terms of attribute relations and base relations. As before, consider first productions of the form $A \rightarrow B^+$, where $B \leftarrow Q(A)$ is the associated semantic rule. In this case, $\text{edge}_A B$ is the set of pairs $(a, b)$ such that $a \in \text{gen}_A$ with $Q_1(cno, title, ...) as (Q1 from Figure 3)

with $Q_2(idc, idp, pq_idc, pq_idc, srel)$

with $Q_3(idc, course, tag)$

and $b \in Q(a)$, where $ia = \text{gen}_A(a)$ and $ib = \text{gen}_A(b)$. To derive a query $Q_{\text{edge} A B}$ for an edge relation $\text{edge} A B$, we can employ the same rewriting as for the attribute relations, with the following change: the select list of the attribute relation is replaced with $(ia, ib)$. The definition of $Q_{\text{edge} A B}$ is similar for productions of other forms.

The set of all $Q_{\text{edge} A B}$ queries for $\sigma$ is denoted by $\text{ER}(\sigma)$. As an example, Fig. 6 shows two edge-relation generating queries derived from the $\sigma$ of Fig. 3.

4.2 Computing Attribute and Edge Relations with sqs 99 Linear Recursion

As the running example illustrates, the attribute relations are potentially mutually recursive, and the edge relations depend on the attribute relations. Furthermore, each attribute relation can be related to itself and other attribute relations through a variety of paths, raising the possibility that concluding evaluation of updates to the attribute relations will be excessively complex. Fortunately, this is not the case.

While the attribute relations are recursive there are substantial constraints on this recursion. Consider the references in the query $Q_{\text{gen}_A}$ to other attribute relations $B, C, \ldots$. First, observe that such a relationship cannot be made via negation (in the Datalog sense), that is the reference to $B$ cannot appear in $\text{not exists}$ clause in $Q_{\text{gen}_A}$. Second, note that references to two different attribute relations for example $B$ and $C$, must be made in two different subqueries of $Q_{\text{gen}_A}$, such that these subqueries are joined only by the top-level union operator introduced by the construction. Finally since expressions are not allowed in the select clauses of view-definition queries, aggregate expressions which depend on attribute relations will not appear.

Given these restrictions, the computation of the attribute and/or edge relations can be accomplished with the sqs 99 with recursive construct. Let $Q_0$ be a set of edge and attribute relation-generating queries from $\sigma$ which either includes all of the attribute relations or all of the edge relations. Let $GQ$ be a graph in which there is a node for each query in $Q_0$ and an edge from $Q_0$ to $Q_1$ if $Q_0$ refers to the virtual relation $\text{gen}_A$ in the from clause. Let $c_1, \ldots, c_n$ be the connected components of this graph. We now merge components and generate possibly recursive queries along the same lines as [17]. First, singleton components with only a single incoming edge is merged with the source component. Let the result of this process be $c_1, \ldots, c_m$. For each such component $c_i$, if it is acyclic then a set of merge-queries...
Figure 8: Middleware Architecture

$M_Q_{c_1}$, for the component can be defined. The number of queries produced will depend on the choices made for query merging [4, 11], and once structures can be handled with an embedded with clause as suggested by [17].

If the component is cyclic, a single recursive query $M_Q_{c_1}$ is defined to compute the outer union of all the virtual relations in $c_1$ with the following steps:

1. The schema of $M_Q_{c_1}$ is the union of all columns appearing in any query of $c_1$ plus an additionalarel attribute intended to encode which virtual relation a given tuple in $M_Q_{c_1}$ represents.

2. The initial condition for $M_Q_{c_1}$ is the union of the queries for nodes with edges incident on $c_1$ in $G_Q$.

3. The query $M_Q_{c_1}$ is the union of all of the queries of $c_1$.

In each subquery, if a mention to a virtual attribute relation gen$\_A$ appears it is replaced with a reference to $M_Q_{c_1}$, and a conjunctive condition is added to the where clause to ensure that $arel = \{'A'\}$; similarly for edge relations.

Applying this algorithm to the ATG of Fig. 3, we get two components after merging. One component consists of $Q_{edge\_prereq\_course}$ and $Q_{edge\_prereq\_course}$, and is defined as the linear-recursive QC2 in Fig. 7. All the remaining queries are in the other, which is not recursive.

We refer to the set of queries $M_Q_{c_1}$ generated for an ATG $\sigma$ as $M_Q$. It is easy to verify the correctness of the mapping: for any $\sigma : R \rightarrow D$ and any relational instance $I$ of $R$, $M_Q(I)$ computes $gen\_A$ and $edge\_A$ for all attributes $A$ and parent-child edges $(A,B)$ in the XML view $\sigma(I)$.

In response to relational updates $\Delta I$, each edge relation $edge\_A,B$ defined by sq. 99 queries $M_Q_{c_1}$ is updated via insertions and deletions. Insertions and deletions over all edge relations are collected into two sets $E^+$ and $E^-$, respectively. The two sets $E^+$ and $E^-$ are sent to the middleware maintaining the external tree, as described in the last section. As mentioned in the introduction, this approach assumes that the underlying DBMS incrementally computes edge changes $E^+$ and $E^-$, and, further that these incremental changes can be captured for example with triggers.

5. Bud-Cut Incremental Evaluation

As discussed in the introduction, the reduction approach is not practical for middleware-based XML publishing since it depends on a combination of features not yet found in even the most advanced commercial DBMS. While middleware should depend on only the most common features. This observation motivates us to propose the bud-cut approach to incrementally evaluating ATG $\sigma : R \rightarrow D$, which does not require the underlying DBMS to support with... recursive.

A middleware system based on the bud-cut mechanism is depicted in Fig. 8. The system interacts with an underlying DBMS and maintains a hash index $H$ and a subtree pool for the external XML view $T(\sigma(I))$ as described in Sect. 3. It responds to a relational update $\Delta I$ in three phases. The first phase, bud-cut generation, identifies the portions of the existing tree that will be affected by the updates $\Delta I$, and propagates $\Delta I$ to XML changes to the existing nodes in $T$. Nodes created in this phase are buds, i.e., they are marked unexpanded, and nodes to be removed are cuts. The second phase, bud completion, constructs subtrees under buds (and generates new buds), taking advantage of ATG properties to avoid recomputation and reusing existing subtrees when possible. The third phase, garbage collection, runs after bud completion is finished and removes unreachable subtrees.

While requiring several round-trips between the middleware and the nodes, this approach is able to exploit other optimizations by taking advantage of the specifics of ATG semantics and XML views. In particular, as observed in Sect. 3, since the value of a subtree $\sigma(A,I)$ is determined by the tuple value $t$ of the semantic attribute of the subtree's root node, changes to the children of the existing nodes in the tree can be computed by a fixed set of non-recursive queries. Furthermore, any new subtree $\sigma(A,I)$ can be reused and thus needs to be computed at most once. In addition, the bud-cut approach allows the update process of the external tree $T$ to run in parallel with the computation of XML changes $\Delta T$, thus improving the response time. This last point leads to a variety of options for lazy evaluation of incremental updates. That is, the middleware can optionally defer complete processing of updates until the subtrees affected by those updates are accessed. When the external view is materialized as an in-memory tree, it is possible to support tree navigation concurrent with processing [7, 23].

5.1 Implementation of Delta SQL Queries

Assuming the existence of virtual attribute relations, the first step is to derive two sets of incremental (nonrecursive) sq. queries from edge-generating queries $ER(\sigma)$. These two forms of incremental queries will be used in the bud-cut generation phase and the bud-completion phase, respectively.

Incremental queries of the first form are derived as follows. For each query $Q_{edge\_prereq\_course}$, in $ER(\sigma)$, we define a bud generating query formed by adding all the columns of the attributes $A$ of the select list of $Q_{edge\_prereq\_course}$. An incremental form of this bud-generating query, $\Delta Q_{edge\_prereq\_course}$ is thus created by using a counting method like [15]. In a nutshell, [15] associates a count with each tuple in a view to keep track of the number of alternative derivations of the tuple via the view, and computes updates to the view, namely, insertions and deletions of tuples, by incrementing or decrementing the counts of its tuples in response to changes to base relations.

Since the method of [15] is assumed to execute in the DBMS, it assumes access to both $I$ and $I \oplus \Delta I$. However, our middleware system is separate from the DBMS and can only access $I \oplus \Delta I$. To find $I$, for a given relation, $R_i$, we assume

Figure 9: Incremental query for bud generation.
the edge deletions are handled in this phase, by evaluating the 
incrementalized sq. (bud-generating) queries of ER(I) 
once. Note further that nodes disconnected by the edge 
cuts are not removed from H-index or the tree but instead remain 
in the subtree pool pending reuse during bud generation. 
Garbage collection of the nodes that have been disconnected 
from the tree will be handled by a background process after the 
bud-completion phase, as described in Sect. 3.

A complicated BTD may lead to a large number of 
incremental bud-generating queries. For an given small relational 
change ΔI, however, one only needs to evaluate those that 
refer to a relation affected by ΔI. Techniques [21, 6] for 
identifying queries irrelevant to ΔI can further reduce the 
number of bud-generating queries that need to be evaluated.

Example 5.1: Recall the ATG σ0 defined in Fig. 3. Assume 
that an XML tree σ0(I) of a relational database I is main-
tained by the middleware. Consider relational changes ΔI; 
deletion of a CS course from the course relation, along with 
deletions from prereq and enroll accordingly. Given this 
the bud-cut-generation phase computes only E+, which 
consists of deletions of edges (db.course), (course prereq), etc. 
These changes are made to the XML tree σ0(I) in this phase.

Note that no buds are generated, i.e., BudA is empty for 
all B in the ATG. In other words, although the relational 
changes have impact to the XML tree at an arbitrary level, 
they are captured in the bud-cut-generation phase by evalu-
ating a fixed number of incrementalized sq. queries.

On the other hand, if ΔI is to update the name fields of some 
student tuples, then it involves both deletions and in-
sertions, i.e., Budstudent is nonempty and consists of student 
buds. The bud-cut-generation phase handles deletions and 
the next phase, the bud-completion phase, proceeds to gen-
erate subtrees of new student buds.

5.3 Bud Completion

The previous bud-cut-generation phase creates sets BudA 
consisting of the A-attribute values of new A-nodes for each 
element type A in the ATG, and the bud-completion phase 
is to produce the subtrees of these buds. The process may 
then generate new buds, but does not incur deletions.

The processing of bud completion is conducted by Algo-
rithm eval', given in Fig. 11. The algorithm takes BudA's 
from the generation phase as input, and processes each non-
empty BudA based on the production p = A → a and its 
associated rel(p) (cases 1-4), generating children of these 
A-elements. For example, in case 3, it processes a given set 
BudA as follows. It first finds the edges of the A-nodes in 
BudA that are to be inserted into edge-AB, by evaluating the 
incremental bud-cutting query Qbud-AB(BudA) in the 
DBMS. It then updates the children lists of these A-nodes 
in the subtree pool T, taking advantage of the H-index and 
Id-A. Then, for each B child id-B of such an A-bud id-A, it 
invokes the procedure process(B, id-B) to inspect whether 
there already exists an entry for the node (B, id-B) in the 
hash index H. If so, it simply adds a cross edge (id-A, id-B) 
instead of recomputing the subtree of (B, id-B); otherwise, 
it creates a new H-entry for (B, id-B) and adds the attribute 
value $B of id-B to BudB. This yields BudB, the set of new 
B-buds that will be further expanded at the next level of 
the tree. Note that to populate BudB we need the attribute 
values $B of these newly inserted B nodes.

Observe the following properties of Algorithm eval.'
Input: newly inserted nodes $Bud_A$ for all element types $A$ in $\sigma$.

Output: completed subtree for all nodes in $Bud_A$.

1. repeat
2. for all nonempty $Bud_A$ with $A$ in $\sigma$
do
3. case the production $\rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \right

Figure 11: Algorithm eval

(1) At each iteration (level), any paths in the XML view that begin at the root and do not encounter a bud are guaranteed to be correct, and thus partial results of the new XML view $\sigma(\Sigma \oplus \Delta I)$ can be exposed to the users in parallel with the computation of $\Delta I$.

(2) It minimizes unnecessary computations via the procedure: it reuses subtrees that has been computed either by Algorithm eval or earlier for the old view $\sigma(I)$. Thus each bud is computed at most once. The reuse of the previous computations is possible since no nodes are removed from the $H$-index or subtree pool at this stage.

(3) The bud-completion phase does not need materialization of the attribute relations, since all the $\Sigma$-values that $\Delta Qedge_{Bud_A}$ needs are in $Bud_A$, not in $gen_A$. Note that only case (3) in Algorithm eval needs to access the DBMS.
6. Experimental Results

Our experimental evaluation focuses on the effect of database and update size on the performance of our approach and on the effectiveness of the proposed subtree caching strategy.

We build the source database based on the schema of Example 2.2, but with an additional taughby table giving instances of professors teaching courses. The database size is given in terms of the number of courses, with scaling factors for other tables given in Fig. 12. A smaller pool of \( f \% \) of the courses are used to initially build the tree. In effect, \( f \) controls the probability a new prerequisite will exist in the tree already. For the recursive part of the schema, between one and three random prerequisites with lower \( \text{core} \) from the pool are generated for each course. The maximum depth of this recursive part is limited to 8 levels of XML nodes.

An update consists of inserting or deleting a course and its associated prerequisites. Fixed size batches of meaningful updates, \( w = \Delta I \), are generated by ensuring that inserts are not duplicates and that deleted courses are present. Since deletes in our system generally execute much faster than inserts, \( w \) is constrained to have 50% inserts unless stated otherwise. The locality of updates is controlled by selecting the updates in \( w \) from a random pre-generated "universe," \( S \), of courses. When \( S \) is smaller, the updates in \( w \) are more related. The parameters and their default values are summarized in Fig. 13. Experiments are performed by running sequences of such batches for two workloads W1 and W2, and using \( \text{bud-cut} \) to propagate the changes to the external tree. The first workload, W1, consists of executing such batched updates against our example \( \text{atc} \), \( \sigma_0 \). A second workload W2 modifies \( \sigma_0 \) to restrict prerequisites to courses taught by instructors from the computer science department. Thus, W2 introduces a join to \( \text{taughby} \) in the recursive part of the view.

The experiments were run on a system with a 2GHz Pentium 4 processor with 1GB of \( \text{RAM} \); they were conducted with a large (256MB) \( \text{warn dmns buffer cache} \). While incremental update may reasonably avoid disk access since the relevant data was just updated, fully cached operation for document publishing is also reasonable, at least for published \( \text{xml} \) data up to a few tens of megabytes in size given modern database configurations. Similarly, the external \( \text{xml} \) tree (see Sect. 3) is fully cached in \( \text{RAM} \) on the same system. Each experiment ran five times and the average is reported here. All numbers obtained for each average are within 6% of the average value. Unless otherwise noted, we used the proposed caching strategy.

We compare the cost of incremental update against the cost of full tree recreation for a variety of database sizes. We scale the database by varying \( |I| \) from 100 to 10K, and setting \( |w| \) to 4% of \( |I| \). For this experiment, locality is not considered (\( S = I \), \( f = 100\% \)). Figure 14(a) shows the resulting time with \( |I| \) on a log-scale for workloads W1 and W2. Not surprisingly, incremental update far outperforms full reconstruction for small updates once the overall database size gets appreciably large. Note that incremental update for W2 is faster than for W1 since the tree is significantly smaller (and thus less update work in the tree is needed).

Fixing \( |I| = 10K \), we vary \( |w| \) and study the behavior of incremental update vs. full tree recreation. The result is presented in Fig. 14 (b). As this figure shows, for W1 and W2, \( \text{bud cut} \) scales nicely with the size of the change in the source database. \( |w| \). We note that the \( \text{bud-cut} \) approach is sensitive to the selectivity of the \( \text{atc} \) queries, and low selectivity leads to substantial activity on the hash index in the subtree pool. Accordingly, the incremental update and the full tree recreation for W1 and W2 cross when 10% and 80% of the database is updated, respectively.

To investigate the impact of the subtree pool, we evaluate the performance impact of turning it off. We fix \( |w| = 100 \) and vary \( |S| \) from 100 to 600 (decreasing the locality of updates as \( |S| \) grows) for W1. (In this experiment, the percentage of inserts in \( w \) is allowed to vary.) Finally, we consider two values of \( f \), 50% and 80%, to control the probability that a given subtree will already appear in the tree. The results are shown in Fig. 15(a), where each point represents the average of 20 such experiments, with garbage collection fired after each. As expected, the impact of subtree caching is greatest with smaller values for \( |S| \) and \( f \). With \( f = 50\% \), the impact is substantial across the range. However, the curves for \( f = 80\% \) are rather flat, since most cached subtrees are small due to "natural caching" by the rest of the tree.

While in Fig. 14(b) we compare the size of the change in the source database to the time taken, it is also interesting to consider the size of the change in the \( \text{xml} \) tree, and compare this to the time taken for the incremental update. An experiment to investigate this relationship is shown in Fig. 15(b) (for workload W1). To capture this we output the \( \text{xml} \) tree before and after the incremental update \( w \). We then use X-Diff [32] to produce an edit script converting the old tree to the new tree and measure its size as an approximation of \( |\Delta T| \). Furthermore, \( |w| \) remains 100, \( f \) is 50% and \( |S| \) is set to 300. This experiment shows that \( \text{bud-cut} \) also behaves reasonably well with respect to output size, and further, that the \( \text{subtree caching} \) optimization (in combination with the graph-oriented storage model) tends to be more effective exactly when an incremental update to

![Figure 12: Table sizes](image)

![Figure 13: Table of symbols](image)
the database causes a large change in the output tree.

7. Related Work

Incremental computation has proved useful in many areas (see [27] for a survey). In particular, incremental view maintenance algorithms have been extensively studied for relational, datalog (see [13] for a collection of readings) and object-oriented views (see, e.g., [12, 18]). On the one hand, those algorithms are not directly applicable to incremental evaluation of XML publishing. Unlike traditional database views, XML views (ATGs) are defined by associating a collection of SQL queries with a (possibly recursive) DTD. These views are stored outside of DBMS, and their incremental maintenance is to propagate relational group updates to external XML trees. On the other hand, our incremental evaluation algorithms leverages previous work for traditional databases. Our reduction approach relies on the support of incremental maintenance of SQL views by the underlying DBMS, and our top-down mechanism makes use of the counting algorithm of [15] to incrementally evaluate SQL queries.

We now draw the analogy of our top-down ATG evaluation algorithm to incremental evaluation of recursive queries. A number of algorithms have been developed for evaluating recursive datalog queries and prolog programs notably [15, 29]. These algorithms typically involve a first phase where conservative deletions are conducted, followed by a second phase where insertions are performed, which may restore information deleted in the first phase. In contrast, our algorithm avoids recomputation between the two phases to an extent by reusing subtrees computed earlier and by deferring removal of disconnected subtrees. For example consider an extreme case where the conservative deletion involves deletion of the root of the tree, and thus the entire tree, whereas in the second phase the tree is reassembled with somewhat differing children of the root. While this may require recons-
The only work on maintaining XML views that we are aware of is [10]. The views considered in [10] are defined via a simple nonrecursive algebraic query over XML trees and as a result, maintenance of the views can be done via a bottom-up traversal of the source XML tree. These views are not capable of expressing ATGs, and the techniques of [10] cannot be applied to incremental evaluation of XML publishing of relational data. There has also been recent work on incremental XML validation [26], which differs from our work in that it focuses on validating XML documents in response to a single insertion or deletion of XML subtrees, rather than propagation of relational (group) updates to XML views.

Finally, the strategy commonly used by XML publishing middleware that pushes simple queries to the DBMS and conducts the rest of the work for composing a view at the middleware back to query processing for distributed relational databases [5]. Again recursive XML views introduce new challenges not encountered in the relational context.

8. Conclusion

We have proposed two approaches for incremental evaluation of ATGs [4]. The reduction approach is based on a non-trivial translation of ATGs to SQL 99 queries with recursion. Upon the availability of the support for incremental maintenance of SQL 99 views by commercial DBMS, the reduction approach leads to a convenient mechanism to maintain external XML views without requiring sophisticated middleware implementation. In the absence of high-end DBMS features, the bud-cut approach provides novel algorithms and a middleware system for efficient incremental maintenance of XML views. The algorithms and system minimize unnecessary recomputations by capitalizing on the ATG semantics and optimization techniques developed for XML publishing. We have implemented the bud-cut middleware, and our experimental results demonstrate that our approach, algorithms and optimization techniques are effective for maintaining XML views. To the best of our knowledge, our work yields the first effective framework for incremental evaluation of schema-directed XML publishing of relational data.

We plan to extend the current work in a number of directions. First, we note that our assumption of an in-memory hash table limits the technique for extremely large documents cached in middleware. We plan to address this by developing a streaming version of the incremental update so that an existing document can be read from disk and updated in a single pass. Second, the more complicated XML schema standard is gaining popularity, and our next step toward handing XML schema-directed publishing will be to extend our bud-cut algorithm to accommodate XML integrity constraints. Finally, we are planning to extend the present algorithms for use in schema-directed XML integration [5].

9. References