Incremental Evaluation of Schema-directed XML Publishing

Citation for published version:
Bohannon, P, Choi, B & Fan, W 2004, Incremental Evaluation of Schema-directed XML Publishing. in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data.* SIGMOD '04, ACM, New York, NY, USA, pp. 503-514. https://doi.org/10.1145/1007568.1007625

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

Incremental Evaluation of Schema-Directed XML Publishing

Philip Bohannon
Bell Laboratories
bohannon@research.bell-labs.com

Byron Choi*
University of Pennsylvania
kkchoi@gradient.cis.upenn.edu

Wenfei Fan†
University of Edinburgh & Bell Laboratories
wenfei@inf.ed.ac.uk

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 publishing, 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 formalism for schema-directed XML publishing. The *reduction* approach seeks to push as much work as possible to the underlying DBMS. It is based on a relational encoding of XML trees and a nontrivial translation of ATGs to SQL 99 queries with recursion. However, a weakness of this approach is that it relies on high-end DBMS features rather than the lowest common 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 publishing. While implementation of the reduction approach is not yet in the reach of commercial DBMS, we have implemented 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 management, 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 incrementally update published XML documents, even when they are *externally cached* by a middleware system. However, to

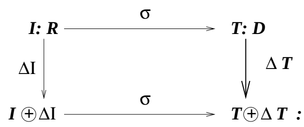

Figure 1: Incremental evaluation

our knowledge, no previous work has considered incremental update of XML documents published from relational data.

In this paper, we consider the particular case of incremental update of XML documents created by *schema-directed* XML *publishing* middleware. The idea of schema-directed publishing is to extract data from a relational database and construct an XML view that *conforms to a predefined schema*, in this case a DTD. The need for this is evident in practice: enterprises typically agree on a common schema for data exchange; thus, a specification for publishing views that ensures schema conformance is an obvious benefit for developers. In response to this need, the ATG (Attribute Translation Grammar) formalism has been developed [4]. An ATG is a mapping $\sigma : R \to D$ associated with a relational schema R and a predefined (possibly recursive) DTD D. Given an instance I of R, σ computes an XML view $T = \sigma(I)$ such that T conforms to D. To accomplish this transformation, D is treated as a set of productions. Each production is annotated by σ with a set of *semantic rules*, and a single *semantic attribute* is defined for each element type. These rules govern the production of child elements for each element based on the data present in the attribute.

To discuss incremental update, we assume that a set of changes to an XML tree T can be encapsulated as ΔT, and that an operator $\oplus$ represents the application of these updates. Given these assumptions, the *incremental evaluation* problem for ATGs can be stated as follows: given an ATG $\sigma : R \to D$, a relational instance I of R, the XML view $T = \sigma(I)$, and changes ΔI to I, compute XML changes ΔT to T such that $T \oplus \Delta T = \sigma(I \oplus \Delta I)$. These relationships are illustrated in Fig. 1. Note further that, if σ is correct, $T \oplus \Delta T$ is trivially guaranteed to conform to the predefined DTD D. In contrast to recomputing the new view from scratch, incremental evaluation of ATGs can, in principle, improve performance substantially by applying only the changes ΔT to the old view T. To realize the improvement requires a) an efficient algorithm that computes the XML changes ΔT in response to relational changes ΔI, and b) an efficient implementation of the tree-update operator $\oplus$.

Our first contribution is a *reduction framework* for incremental maintenance of XML views defined by ATGs. For a given ATG definition σ, the framework consists of a) a set of virtual relations associated with the semantic attributes of σ, b) queries defining the recursive relationship of these virtual relations to encode ATG-produced XML documents, c) a mapping based on these recursive queries from ATGs to SQL 99 queries with linear recursion [24], and d) mechanisms for computing XML changes ΔT from relational changes ΔI

*Supported in part by an Earmarked Research Grant.
†Supported in part by NSF Career Award IIS-0093168, NSFC 60228006 and EPSRC GR/S63205/01.

SIGMOD 2004, June 13-18, 2004, Paris, France.

and for updating the external view T with ΔT.

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 tree. These techniques, however, cannot be directly applied to generating or updating ATG-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 XPath 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 ΔI on existing parent-child relations in the XML view, *i.e.,* insertions (*buds*) and deletions (*cuts*), by evaluating a fixed number of incrementalized 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 ΔT 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 ΔI. Unfortunately but not surprisingly, the *reduction* approach is unrealizable 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 [3], 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(ssn, name),          enroll(ssn, cno),
prereq(cno1, cno2).
```

The database maintains `student` data, `enroll`ment records, course data classified into regular `courses` and `projects`, and a relation `prereq`, which gives the prerequisite hierarchy of courses where a tuple `(c1, c2)` in `prereq` indicates that `c2` is a prerequisite of `c1`.

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):

```
<!ELEMENT db       (course*)>
<!ELEMENT course   (cno, title, type, prereq, takenBy)>
<!ELEMENT type     (regular | project)>
<!ELEMENT prereq   (course*)>
<!ELEMENT takenBy  (student*)>
```

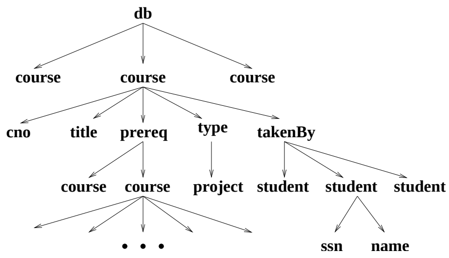

Figure 2: XML view

```
<!ELEMENT student (ssn, name)>
<!ELEMENT regular (empty)>    /* similarly for project */
```

An XML view conforming to D_0 is depicted in Fig. 2. It consists of a sequence of `course` elements, which represent all the CS courses and projects. Each `course` has a `cno` (course number), a course `title`, a `type` indicating whether it is a course or project, a prerequisite hierarchy, and all the students who have registered for the course.

The registrar database is updated constantly. Examples of (group) updates [14] include: (1) insertion of a new CS course to the `course` relation, along with insertions of its prerequisites to `prereq` and insertions to `enroll` for the students who have enrolled in the course, (2) deletion of a CS course from the `course` relation, along with deletions from `prereq` and `enroll` accordingly, and (3) updates of the name fields of some `student` tuples.

As will be seen shortly the XML view can be defined with an ATG that guarantees the view to conform to D_0. Incremental evaluation of the ATG is to update the materialized XML view in response to updates to the registrar database. Note that since the ATG is defined on a recursive DTD, relational updates may cause insertions, deletions and structure changes at an arbitrary depth of the XML view, which cannot be decided at compile time. □

2.1 DTDs

Without loss of generality, we formalize a DTD D to be (E, P, r), where E is a finite set of *element types*; r is in E and is called the *root type*; P defines the element types: for each A in E, $P(A)$ is a regular expression of the form:

$$\alpha \ ::= \ \texttt{str} \mid \epsilon \mid B_1, \ldots, B_n \mid B_1 + \ldots + B_n \mid B^*$$

where ϵ is the empty word, B is a type in E (referred to as a *child type* of A), and '+', ',' and '*' denote disjunction, concatenation and the Kleene star, respectively (we use '+' instead of '|' to avoid confusion). We refer to $A \to P(A)$ as the *production* of A. A DTD is *recursive* it has an element type that is defined (directly or indirectly) in terms of itself.

It has been shown in [4] that all DTDs can be converted to this form in linear time by introducing new element types and performing a simple post-processing step to remove the introduced elements. To simplify the discussion we do not consider XML attributes, which can be easily incorporated.

An XML document (tree) T *conforms to* a DTD D, if (1) there is a unique node, the *root*, in T labeled with r; (2) each node in T is labeled either with a type $A \in E$, called an A *element*, or with `str`, called a *text node*; (3) each A element has a list of children of elements and text nodes such that they are ordered and their labels are in the regular language defined by $P(A)$; and (4) each text node carries a string value (`str`) and is a leaf of the tree.

2.2 ATGs

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 $\sigma : R \to D$ 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, σ defines 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 $p = A \to \alpha$ in D, σ specifies a set of semantic rules, *rule*(p). These rules specify the computation of the B children of an A element for each type B in α.

Given a database I of R, the ATG σ 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 $p = A \to \alpha$ in D, and generate the children of b by evaluating *rule*(p) and using the value of the attribute $\$A$ of b. The rules *rule*(p) are defined and evaluated based on the form of α as follows:

(1) If α 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, $\$B_i = (\$A.a_i^1, \ldots, \$A.a_i^k)$ is in *rule*(p) for $i \in [1, n]$, where a_i^j is a field of the tuple $\$A$.

(2) If α is $B_1 + \ldots + B_n$ then *rule*(p) is defined by

```
($B1, $B2, ..., $Bn) =
  case f($A) of
       1: ($A, null,...,null)
          ...
       n: (null,...,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_j child is created if $i \neq j$, and $\$B_j$ (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 α is B^*, then *rule*(p) is defined by $\$B \leftarrow Q(\$A)$, where Q is an SQL 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 α is `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 α is ϵ, then no $r(p)$ 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 σ_0 given in Fig. 3 defines the XML view described in Example 2.1. Here *rule*(`course`),

```
db → course*
  $course ← Q1
  Q1:    select  distinct c.cno, c.title, 1 as tag
         from    course c
         where   c.dept = ''CS''
         union
         select  distinct p.cno, p.title, 2 as tag
         from    project:  p
         where   p.dept = ''CS''
course → cno, title, type, prereq, takenBy
  $cno = $course.cno,        $title = $course.title,
  $type = $course.tag,       $prereq = $course.cno,
  $takenBy = $course.cno
type → regular + project
  ($regular, $project) = case $type of
                         1: ($type, null)
                         2: (null, $type)
prereq → course*
  $course ← Q2($prereq)
  Q2(c1):    select  distinct c.cno, c.title, 1 as tag
             from    prereq p, course c
             where   p.cno1 = c1 and p.cno2 = c.cno
takenBy → student*
  $student ← Q3($takenBy)
  Q3(c):    select  distinct s.ssn, s.name
            from    enroll e, student s
            where   e.cno = c and e.ssn = s.ssn
```

Figure 3: Example ATG σ_0

rule(type) and *rule*(prereq) illustrate the cases (1), (2), (3) above. Given a registrar database, σ_0 computes an XML view as follows. It first generates the root element (with tag db), and then evaluates the query Q_1 to extract courses and projects of the CS department from the underlying database. For each distinct tuple c in the output of Q_1, it generates a course child v_c of db, which is a bud carrying c as the value of its attribute $course. 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 Q_2 to find prerequisites of the course, and again for each tuple in the output of Q_2 it generates a course node; similarly it constructs the takenBy subtree by evaluating Q_3 to extract student data. Note that Q_2 and Q_3 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 Q_2 returns empty at the prereq children of those course nodes. That is, ATGs 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 as defined refine 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, σ_0 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 $\sigma : R \to D$, whether σ 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 $p = A \to \alpha$, namely, $p' = A \to \alpha + \texttt{str}$. For example, we extend the production of course in the DTD D_0 to be

course → (cno, title, type, prereq, takenBy) + str

In order to ensure termination, we modify tree generation to stop expanding the tree if a newly created node lv has the same type and semantic attribute value as one of its ancestors, v. In this case, we simply emit the string value of lv's attribute as the contents of the new node. It should be mentioned that this does not lose information, since the subtree of lv, if constructed by following the original production $p = A \to \alpha$, 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 $\sigma : R \to D$ be an arbitrary ATG with exception handling. Then over all instances I of R, σ terminates and $\sigma(I)$ conforms to the DTD D.* □

Theorem 2.1 shows that an ATG $\sigma : R \to D$ is actually a total function: given a database I of R, $\sigma(I)$ computes an XML documents of D that is unique up to reordering of B-elements produced by productions of the form $A \to B^*$. 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 $\sigma : R \to D$ 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 $A at the root of T_A. Thus the ATG defines a function ST such that, given an element type A and a value t of A, ST(A, t)$ returns a subtree rooted at a node tagged A and carrying t as its attribute.

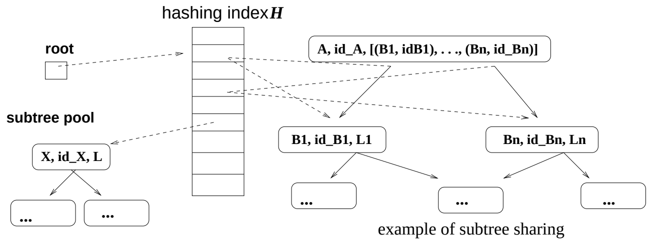

Figure 4: Middleware Data Structures

Since a subtree $\text{ST}(A, \$A)$ 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 $\text{ST}(A, \$A)$ 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, id_A, ptr), where A is an element type, id_A is the unique id of a value of $\$A$, and ptr is a pointer to the root node of the subtree $\text{ST}(A, \$A)$ in S. The subtree pool S consists of entries (A, id_A, L), where (A, id_A) represents a node v of T, and L is a list $[(B_1, id\_B_1), \ldots, (B_n, id\_B_n)]$ representing the children of v such that each (B_i, 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 ΔI to be any group of updates to the underlying DBMS that preserves the consistency of the database (integrity constraints). XML updates ΔT generated from ΔI 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 (id_A, id_B), where id_A and 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 (id_A, id_B) in E^+, 1) find the H entry (A, id_A) and the pointer to the node (A, id_A, L) in the subtree pool S, 2) insert (B, id_B) into L if it is not already in L, and 3) if H does not have an entry (B, id_B) in H, create an entry in H and a node $(B, id\_B, [\,])$ in S with an empty child-list $[\,]$. Note that if there is already an entry (B, id_B) in H, the new edge is actually a *cross edge* to the existing B node. A subtle issue concerns the order of the children list L when a B node is inserted into L. Note that L is constrained by the production $A \to \alpha$: if α is $B_1, \ldots, B_n$, the elements of L are in the same order as their element types in the production; and if α is B^*, L elements are ordered by a default order on the tuple values of the $\$B$ attributes of these elements.

The system carries out deletions similarly: for each tuple (id_A, id_B) in E^-, 1) find the node (A, id_A, L) in the subtree pool via the H-index entry $H(A, id\_A)$, and 2) *cuts* the edge (id_A, id_B) by removing id_B from the children list L. It should be noted that during the processing of relational updates ΔI, no entries are physically deleted from H because a) an entry may be shared by multiple subtrees of T, and b) as will be seen shortly, our incremental system minimizes unnecessary recomputations by reusing subtrees that have been disconnected from the tree by cuts.

```
Q_gen_prereq:
  select  gen_course.cno
  from    gen_course gc
Q_gen_course:
  select  distinct c.cno, c.title, 1 as tag  /* Q'_1 */
  from    course c
  where   c.dept = ''CS''
  union
  select  distinct p.cno, p.title, 2 as tag
  from    project p
  where   p.dept = ''CS''
  union     /* Q'_2: */
  select  distinct c.cno, c.title, 1 as tag
  from    prereq p, course c, gen_prereq g
  where   p.cno1 = g.cno  and p.cno2 = c.cno
```

Figure 5: Attribute relation generating queries.

Garbage Collection. Upon the completion of the computation of the new XML view $\sigma(I \oplus \Delta I)$, a garbage collection process runs in the background to remove H-entries and nodes in the subtree pool that are not linked to $\sigma(I \oplus \Delta I)$. Conceptually this can be done via a top-down traversal of the new XML tree, removing nodes that are no longer reachable from the root. In our implementation we associate with each H-entry a *count* keeping track of the number of nodes linked to it, and maintain the counters when processing insertions and deletions; the garbage collection process removes unused nodes, *i.e.*, those H-entries with $count = 0$ and their corresponding nodes in the subtree pool.

4. 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 ATG $\sigma : R \to D$ to a set $\mathcal{MQ}_\sigma$ of SQL 99 queries utilizing the `with...recursive` construct, which computes the attribute and edge relations of the XML views defined by σ.

Given this, the *reduction* approach to incrementally maintaining an XML tree T computed by ATG $\sigma(I)$ works as follows: 1) encode T with the attribute and edge relations, 2) map σ to SQL 99 queries $\mathcal{MQ}_\sigma$, 3) define an incrementally updated materialized view in the source DBMS for each query of $\mathcal{MQ}_\sigma$, 4) in response to relational updates ΔI, utilize the DBMS functionality to capture the incremental changes made to each of the materialized views which can be directly transformed into the XML changes ΔT (*i.e.*, E^+ and E^-), and 5) propagate the changes ΔT to the external tree, 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 and translation of 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.

```
Q_edge_course_title:
  select   gen_id(c), gen_id(c.title)
  from     gen_course c

Q_edge_prereq_course:      /* derived from Q'_2 */
  select   gen_id(gp), gen_id(c.cno, c.title, 1 )
  from     gen_prereq gp, prereq p, course c
  where    p.cno1 = gp.cno and p.cno2 = c.cno
```

Figure 6: Edge relation generating queries.

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 σ associates with each element type B a tuple-formatted variable $\$B$. For each such variable $\$B$, let 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 gen_B is populated with all the (non-`null`) values taken on by $\$B$ during an evaluation of σ on I.

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

(1) For productions of the form $A \to B^*$ with associated semantic rule $\$B \leftarrow Q(\$A)$, gen_B is the union, over all values of $\$A$ in 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 gen_A. This is accomplished, roughly, by a) adding gen_A to the `from` list of Q, and b) replacing references to $\$A$ in Q with the corresponding references to gen_A.

(2) If B appears as some B_i when α is $B_1, \ldots, B_n$, then Q' is a simple selection query from gen_A that projects fields according to $f_i(\$A)$ (see Section 2).

(3) If B appears as some B_i in α is $B_1 + \ldots + B_n$ then Q' can be written as `select * from` gen_A a `where` $f(a) = i$. Note that we assume further that $f(a)$ is computable in the dialect of SQL 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 σ. Suppose that it appears on the RHS of rule $p_1, p_2, \ldots, p_n$, and that Q'_i is defined per the discussion above. Then Q_{gen_A} is formed by taking the (distinct) union of all Q'_i queries.

For example, Fig. 5 shows the definition of two attribute-relation generating queries when the resulting construction is applied to the ATG σ_0 of Fig. 3.

We denote by $AR(\sigma)$ the set of all Q_{gen_x} queries for σ.

Maintaining Edges. We next describe how to capture the edge relations of the XML view. Let $edge\_A\_B$ 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 gen_id described in Sect. 3 to compute the unique id ia 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 $A \to B^*$, where $\$B \leftarrow Q(\$A)$ is the associated semantic rule. In this case, $edge\_A\_B$ is the set of pairs (ia, ib) such that $a \in gen\_A$

```
with QC1(cno,title,...) as (Q1 from Figure 3)
with QC2(idc, idp, pq_idp, pq_idc, srel)
     recursive as {
     select  gen_id(c.cno, c.title, 1) from QC1
     union
     select gen_id(gp), gen_id(c.cno, c.title, 1),
            null, null, ''course''
     from   QC2 gp, prereq p, course c
     where  p.cno1 = gp.cno and p.cno2 = c.cno
            and gp.srel = ''prereq''
     union
     select null, null, cno as pq_idc,
            cno as pq_idp , ''prereq''
     from   QC2 gc
     where  gc.srel = ''course''    }
select distinct cno, title, tag
from   QC2, course
where  QC2.srel=''course'' and QC2.idc=course.cno
```

Figure 7: `with...recursive` for recursive schema

and $b \in Q(a)$, where $ia = gen\_id(a)$ and $ib = gen\_id(b)$. To derive a query $Q_{edge\_A\_B}$ for an edge relation $edge\_A\_B$, 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 $Q_{edge\_A\_B}$ is similar for productions of other forms.

The set of all $Q_{edge\_A\_B}$ queries for σ is denoted by $ER(\sigma)$.

As an example, Fig. 6 shows two edge-relation generating queries derived from the σ_0 ATG of Fig. 3.

4.2 Computing Attribute and Edge Relations with SQL 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_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 a `not exists` clause in Q_gen_A. Second, note that references to two different attributes relations, for example B and C, must be made in two different subqueries of Q_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 SQL 99 `with...recursive` construct. Let $\mathcal{Q}_\sigma$ be a set of edge and attribute relation-generating queries from σ which either includes all of the attribute relations or all of the edge relations. Let G_Q be a graph in which there is a node for each query in $\mathcal{Q}_\sigma$ and an edge from Q_B to Q_A iff Q_B refers to the virtual relation gen_A in the from clause. Let $c^0{}_1, \ldots, c^0{}_n$ 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 $c_1, \ldots, c_n$. For each such component c_i, if it is acyclic, then a set of *merge-queries*,

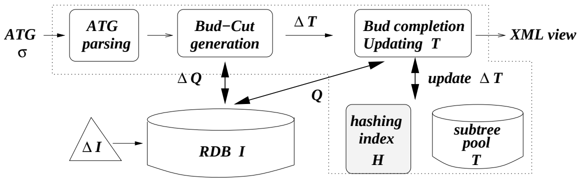

Figure 8: Middleware Architecture

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

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

(1) The schema of MQ_{c_i} is the union of all columns appearing in any query of c_i plus an additional `srel` attribute intended to encode which virtual relation a given tuple in MQ_{c_i} represents.

(2) The initial condition for MQ_{c_i} is the union of the queries for nodes with edges incident on c_i in G_Q.

(3) The query MQ_{c_i} is the `union` of all of the queries of c_i. In each subquery, if a mention to a virtual attribute relation *gen_A* appears, it is replaced with a reference to MQ_{c_i}, and a conjunctive condition is added to the where clause to ensure that `srel = ''A''`; similarly for edge relations.

Applying this algorithm to the ATG of Fig. 3, we get two components after merging. One component consists of $Q_{edge\_course\_prereq}$ and $Q_{edge\_prereq\_course}$, and is shown 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 MQ_{c_i} generated for an ATG σ as $\mathcal{MQ}_\sigma$. It is easy to verify the correctness of the mapping: for any ATG $\sigma : R \rightarrow D$ and any relational instance I of R, $\mathcal{MQ}_\sigma(I)$ computes *gen_A* and *edge_A_B* for all attributes $\$A$ and parent-child edges $(\$A, \$B)$ in the XML view $\sigma(I)$.

In response to relational updates ΔI, each edge relation *edge_A_B* defined by SQL 99 queries $\mathcal{MQ}_\sigma$ 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 DBMSs, while middleware should depend on only the most common features. This observation motivates us to propose the *bud-cut approach* to incrementally evaluating ATGs $\sigma : R \rightarrow D$, 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 ΔI in three phases. The first phase, *bud-cut generation*, identifies the portions of the existing tree that will be affected by the updates ΔI, and propagates ΔI 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.

```
create view old_course as
  select * from course
  where not exists ( select 0 from Δcourse dc
                     where dc.cno = course.cno );

ΔQ_edge_prereq_course:
   select  p.cnt, gen_id(gp), gen_id(c.cno, c.title, 1),
           c.cno, c.title, 1 as tag
   from    gen_prereq gp, Δprereq p, course c
   where   p.cno1 = gp.cno and p.cno2 = c.cno
  union
   select  c.cnt, gen_id(gp), gen_id(c.cno, c.title, 1),
           c.cno, c.title, 1 as tag
   from    gen_prereq gp, old_prereq p, Δcourse c
   where   p.cno1 = gp.cno and p.cno2 = c.cno
```

Figure 9: Incremental query for bud generation.

While requiring several round-trips between the middleware and the DBMS, 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, $\text{ST}(A, t)$, 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 $\text{ST}(A, t)$ 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 ΔT, 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) SQL 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 $Q_{edge\_A\_B}$ in $ER(\sigma)$, we define a *bud generating* query formed by adding all the columns of the attribute $\$B$ to the select list of $Q_{edge\_A\_B}$. An incremental form of this bud-generating query, $\Delta Q_{edge\_A\_B}$, is then 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

```
ΔQ^Bud_edge_prereq_course(Bud_prereq):
  select  gen_id(gp), gen_id(c.cno, c.title, 1 as tag),
          c.cno, c.title, 1 as tag
  from    Bud_prereq gp, prereq p, course c
  where   p.cno1 = gp.cno and p.cno2 = c.cno
```

Figure 10: Incremental query for bud-completion.

the existence of a *change table* ΔI_i that holds ΔI restricted to R_i as well as a count `cnt` with the value either "+1" or "-1" to indicate an insert or delete. Further, we assume that ΔI_i has already been applied to I_i, the instance of R_i in I. In order to simulate the pre-image of I_i, we define a view old_R_i for each relation $R_i \in R$, which simply selects tuples of I_i that do not appear in ΔI_i. It can be implemented efficiently if ΔI_i is small and is indexed on the key of R_i.

For example, Fig. 9 gives the incremental bud-generating query for *edge_prereq_course*, which is derived from the query $Q_{edge\_prereq\_course}$ by using the counting method of [15]. It computes updates to *edge_prereq_course* in response to changes Δ`course` and Δ`prereq`. Observe that in addition to edges of *edge_A_B*, the query also returns the corresponding $\$B$ values. A query $\Delta Q_{edge\_A\_B}$ is typically a union of several queries each incorporating the effects of changes in one of the base relations referenced by $Q_{edge\_A\_B}$.

Incremental queries of the second form are a mild extension of $ER(\sigma)$. For each $Q_{edge\_A\_B}$ in $ER(\sigma)$, we define an incremental *bud-completing* query $\Delta Q^{\mathsf{Bud}}_{edge\_A\_B}(\mathsf{Bud}_A)$ to compute changes to *edge_A_B* in response to a set Bud_A of insertions to the attribute relation *gen_A*. The query is derived from $Q_{edge\_A\_B}$ by substituting Bud_A for *gen_A* and by adding all the columns of the attribute $\$B$ to the select list of $Q_{edge\_A\_B}$. It computes the edges of the newly inserted A-nodes in Bud_A. In contrast to the incremental bud-generating query $\Delta Q_{edge\_A\_B}$, the incremental query $\Delta Q^{\mathsf{Bud}}_{edge\_A\_B}(\mathsf{Bud}_A)$ takes Bud_A as a parameter and does not access *gen_A*; furthermore, it uses the new relation $I \oplus \Delta I$ without requiring the old I. For example, Fig. 10 shows the incremental bud-completing query of $Q_{edge\_prereq\_course}$, which assumes that `course` has been updated by Δcourse.

5.2 Bud-Cut Generation

Having derived the incrementalized SQL queries, this phase proceeds as follows:

(1) The incremental *bud-generating* queries $\Delta Q_{edge\_A\_B}$ are executed in the DBMS, yielding changes to the edge relations: a set of insertions E^+ and deletions E^-.

(2) The sets (E^+, E^-) are transmitted to the middleware and posted against the tree T, as described in Sect. 3. Moreover, the $\$A$-value of each new A-node added to T is also added to Bud_A, a set of buds of type A. An A-node is marked *unexpanded* if its $\$A$ value is in the Bud_A set.

(3) The bud-cut approach does not require the attribute relations to be materialized in the DBMS since they can be computed from the hash table and the subtree pool in the middleware. To speed up the response time with a tradeoff of space, we treat these relations as views maintained by the DBMS. In the latter case, the changes computed in step (1) above also update these views; furthermore, in step (2) tuples in Bud_A are sent to the DBMS and inserted into *gen_A*, for all element types A in the ATG.

It should be remarked that the bud-cut-generation phase updates the parent-child edges of all the *existing* nodes in the XML view T in response to relational changes ΔI. Thus *all* the edge *deletions* are handled in this phase, by evaluating the incrementalized SQL (bud-generating) queries of $ER(\sigma)$ *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 DTD 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 σ_0 defined in Fig. 3. Assume that an XML tree $\sigma_0(I)$ of a registrar database I is maintained 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 $\sigma_0(I)$ in this phase. Note that no buds are generated, *i.e.,* Bud_B 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 evaluating a fixed number of incrementalized SQL queries.

On the other hand, if ΔI is to update the name fields of some `student` tuples, then it involves both deletions and insertions, *i.e.,* $\mathsf{Bud}_{\texttt{student}}$ is nonempty and consists of `student` buds. The bud-cut-generation phase handles deletions, and the next phase, the bud-completion phase, proceeds to generate subtrees of new `student` buds. □

5.3 Bud Completion

The previous bud-cut-generation phase creates sets Bud_A 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 further generate new buds, but does not incur deletions.

The processing of bud completion is conducted by Algorithm `eval`, given in Fig. 11. The algorithm takes Bud_A's from the generation phase as input, and processes each nonempty Bud_A based on the production $p = A \to \alpha$ and its associated *rule*(p) (cases 1–4), generating children of these A-elements. For example, in case 3, it processes a given set Bud_A as follows. It first finds the edges of the A nodes in Bud_A that are to be inserted into *edge_A_B*, by evaluating the incremental *bud-completing* query $\Delta Q^{\mathsf{Bud}}_{edge\_A\_B}(\mathsf{Bud}_A)$ 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 Bud_B. This yields Bud_B, the set of new B-buds that will be further expanded at the next level of the tree. Note that to populate Bud_B 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 σ.
Output: completed subtrees for all nodes in Bud_A's.

```
repeat
  for all nonempty Bud_A with A in σ do
    case the production p = A → α of
    (1) A → B_1, ..., B_n:
        for each tuple t in Bud_A do
          for i from 1 to n do
            $B_i := f_i(t);    /* f_i in rule(p) for $B_i */
            process(B_i, $B_i);
          let v = H(A, t).ptr;
          v.L := (B_1, gen_id($B_1), ..., B_n, gen_id($B_n));
             /* gen_id: Skolem function */
        Bud_A := ∅;
    (2) A → B_1 + ... + B_n:
        for each tuple t in Bud_A do    /* f in rule(p)*/
          evaluate f(t) to find the unique $B_i ≠ null;
          process(B_i, t);
          let v = H(A, t).ptr;
          v.L := (B_i, t);
        Bud_A := ∅;
    (3) A → B*:
        X := ΔQ^Bud_{edge_A_B}(Bud_A); /* sending Bud_A to DBMS */
        /* executing the incremental edge-generating query*/
          for each tuple (id_A, id_B, $B) in X do
            let v = H(A, id_(A)).ptr;
            if (B, id_B) is not in v.L
            then insert (B, id_B) into v.L;
                 process(B, $B);
        Bud_A := ∅;
    (4) A → str:
        for each tuple t in Bud_A do
          let v = H(A, t).ptr;
          v.L := str (t);  /* computes PCDATA from t */
        Bud_A := ∅;
until Bud_A = ∅ for all A in σ;
```

Procedure $\texttt{process}(A, t)$
Input: element type A, tuple t as a value of $\$A$.
Output: modified index H, subtree pool and the sets Bud_A.

```
tid := gen_id(t);     /*gen_id: Skolem function*/
if (A, tid) is an entry of H
then return;
create a node v = (A, tid, [ ]) in the subtree pool;
create an entry (A, tid) in H;
H(A, tid).ptr := &v;     /*address of v*/
H(A, t).L := [ ];        /* empty list */
Bud_A := Bud_A ∪ {t};  /* newly created nodes */
return;
```

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 $\sigma(I \oplus \Delta I)$ can be exposed to the users in parallel with the computation of ΔT.

(2) It minimizes unnecessary computations via the procedure `process`: 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 $\$A$-values that $\Delta Q^{\mathsf{Bud}}_{edge\_A\_B}$ needs are in Bud_A, not in *gen_A*. Note that only case (3) in Algorithm `eval` needs to access the DBMS.

(4) The procedure `process` also ensures that algorithm `eval` always terminates, no matter whether the input ATG has exception handling or not. Specifically, if a descendant lv of an A-node v with attribute value $\$A$ is also an A node with the same $\$A$ value, the node lv is not created since there is already an entry for $(A, \$A)$ in the H-index; instead, any edge to lv is treated as a cross edge and linked to v. Thus no infinite computation is incurred.

Example 5.2: Consider again the ATG σ_0 and relational changes ΔI that update the name fields of some `student` tuples. The bud-cut-generation phase generates a set $\mathsf{Bud}_{\texttt{student}}$ consisting of these updated `student` tuples, generates new `student` buds for these tuples, and redirects affected edges to these new nodes. The bud-completion phase completes the subtree of these new `students` by creating new `name` fields and reusing `ssn` fields for the students. This is done in one iteration of the outer loop of Algorithm `eval`, although the `students` may appear at arbitrary levels of the XML view.

Now consider changes ΔI consisting of insertion of a new CS course to the `course` relation, along with insertions of its prerequisites to `prereq` and insertions to `enroll` for the students who have enrolled in the course. The bud-cut-generation phase generates a set $\mathsf{Bud}_{\texttt{course}}$ consisting of the newly inserted course tuple, creates a new node c representing the course, and adds an edge from the root `db` to c. The bud-completion phase takes $\mathsf{Bud}_{\texttt{course}}$ as input and constructs the subtree of c. Since all the prerequisites of c are already in the XML view, the subtree of c can be completed in two iterations of the outer loop of Algorithm `eval`. In the first iteration only $\mathsf{Bud}_{\texttt{course}}$ is nonempty, and the algorithm creates new `cno`, `title`, `prereq`, `type` and `takenBy` children of c. The second iteration completes their subtrees by reusing the existing `course` and `student` nodes. Since no new nodes are generated in the second step, the iteration terminates, although the subtree under c may have a depth greater than two due to its prerequisites hierarchy. □

5.4 Implementation Issues

Overlapping Phases. While the first and second phase have been presented as completely separate, in practice it may be advantageous to "pipeline" them. In particular, consider the case where ΔI affects base tables for both $\Delta Q_{edge\_A\_B}$ and $\Delta Q_{edge\_B\_C}$. It makes sense to compute $\Delta Q_{edge\_A\_B}$ first, allowing the A nodes in the tree to be expanded and thus *gen_B* to be incremented before $\Delta Q_{edge\_B\_C}$ is executed. This follows the observation in [15] that incremental queries should be evaluated according to their "stratum number", which reflects dependency relationship.

Query Merging. Query merging has been proved useful for systems to publish relational data in XML [11, 4]. The idea is to reduce the number of queries issued to the DBMS by merging multiple queries into a single, larger query via outer-join/outer-union. Query merging can help decrease the communication costs between the middleware and DBMS, while also potentially diminishing query processing time and execution overheads.

The generation of subtrees under buds is similar to the original process of ATG generation [4], and query merging can be used in the bud-completion phase to merge multiple edge-generating queries of the form $\Delta Q^{\mathsf{Bud}}_{edge\_A\_B}(\mathsf{Bud}_A)$ and $\Delta Q^{\mathsf{Bud}}_{edge\_B\_C}(\mathsf{Bud}_B)$ into one. However, the reuse of existing subtrees is an important difference, and direct application of the merging techniques for ATGs [4] may lead to unnec-

table	cardinality	tuple size
course	1	40 bytes
prereq	1-3/course	8 bytes
student	100	20 bytes
takenby	5-10/course	8 bytes
taughtby	1/2 courses	8 bytes

Figure 12: Table sizes

essary recomputations. Consider an edge (a, b) generated by $\Delta Q^{\mathsf{Bud}}_{edge\_A\_B}$. If node b already *exists*, then there is no need to recompute its subtree, and b should not be included in Bud_B for the computation of $\Delta Q^{\mathsf{Bud}}_{edge\_B\_C}(\mathsf{Bud}_B)$. This can be avoided by adding a condition in the `where` clause of $\Delta Q^{\mathsf{Bud}}_{edge\_B\_C}(\mathsf{Bud}_B)$, ensuring that b is not already in gen_B. With this mild extension, one can use the cost model and merging algorithms of [4] to determine what queries to merge before Algorithm `eval` is executed, and thus further optimize generation of subtrees under buds. The tradeoff is that the attribute relations need to be materialized in 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 `taughtby` 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 prerequesite will exist in the tree already. For the recursive part of the schema, between one and three random prerequesites with lower `cnos` 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 *bud-cut* to propagate the changes to the external tree. The first workload, $W1$, consists of executing such batched updates against our example ATG, σ_0. A second workload $W2$ modifies σ_0 to restrict prerequisites to courses taught by instructors from the computer science department. Thus, $W2$ introduces a join to `taughtby` in the recursive part of the view.

The experiments were run on a system with a 2GHZ Pentium 4 processor with 1G bytes of RAM; they were conducted with a *large* (256MB), *warm* DBMS 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 XML data up to a few tens of megabytes in size, given modern database configurations. Similarly, the external XML tree (see Sect. 3) is fully cached in RAM on the same system. Each experiment was run five times and the average

Symbol	Meaning	Default val.
S	the "universe" of the updated courses	300
f	the percentage of S in the published XML view	50%
w	a unit of work containing both insertions and deletions	$\|w\| = 100$
I	the database size	$\|I\| = 10$K

Figure 13: Table of symbols

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 $W2$, *bud cut* scales nicely with the size of the change in the source database, $|w|$. We note that the *bud-cut* approach is sensitive to the selectivity of the ATG 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 w. 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 *output*, the 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 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 *bud-cut* also behaves reasonably well with respect to output size, and further, that the *subtree caching* optimization (in combination with the graph-oriented storage model) tends to be more effective exactly when an incremental update to

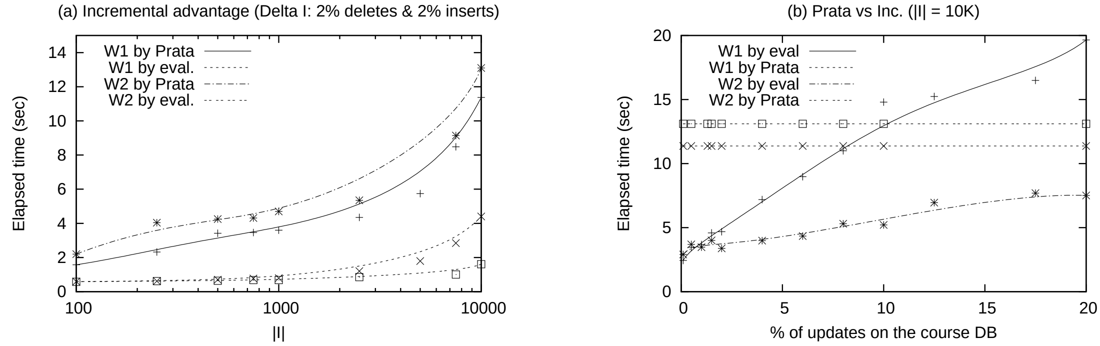

Figure 14: Performance of the *bud-cut* approach on a variety of database sizes $| I |$ and update sizes w

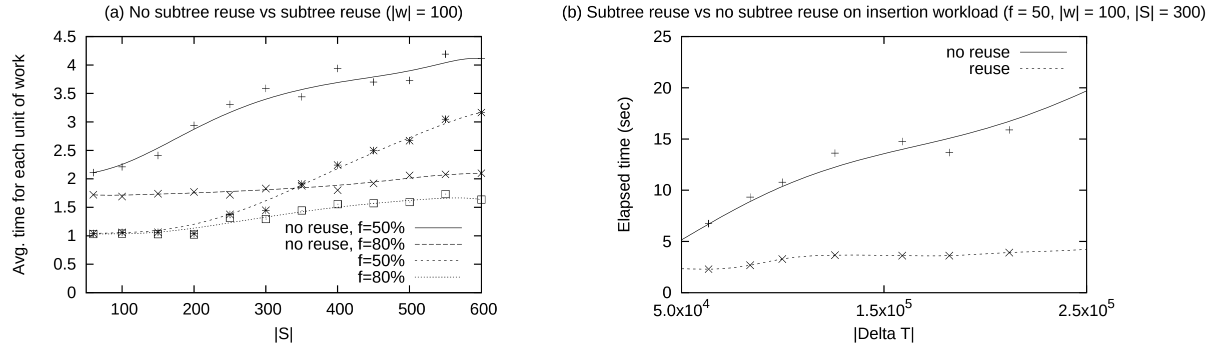

Figure 15: Effectiveness of the subtree pool on related update sequences and the change of the view $| \Delta T |$

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 99 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 reconstruction of the entire tree for the previous algorithms [15], our algorithm is capable of making maximum reuse of existing subtrees without doing unnecessary recomputations. Another difference between our algorithm and previous ones is that the conservative deletion phase repeatedly executes deletion operations for an unbounded number of times [15, 29], whereas by taking advantages of the semantics of XML trees, our bud-cut-generation phase does all the deletions by evaluating a fixed number of incrementalized SQL queries.

It is worth remarking that the possibility of encoding ATGs with SQL 99 queries was first suggested by [17], but [17] did not show how the encoding should be defined. It should be mentioned that there is a big gap between XML trees computed by an ATG and their relational encoding: while the former can be exponentially large in the size of the relational database, the latter is bounded by a polynomial in the database size. Thus an SQL 99 encoding of an ATG alone does not suffice to compute XML trees defined via ATGs.

There have also been incremental maintenance algorithms for semistructured views [1, 31, 33]. These algorithms are developed for views defined with a nonrecursive query over graph structures, and cannot be applied to ATG evaluation.

Incremental update of external views has been studied for data warehousing [22, 30, 34] and web site maintenance [19, 20]. Techniques developed in this work are complementary to ours, addressing *e.g.,* compensating queries to cope with updates to decoupled sources, maintaining mediated views via constrained rules, scheduling updates for a group of views to maximize data quality, and deciding where to materialize web views with respect to access and update patterns.

There has also been a host of work on the self maintainability of relational and datalog views (*e.g.,* [21, 6]). We plan to study techniques for detecting updates irrelevant to XML views in order to optimize XML view maintenance.

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 traces 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 nontrivial 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 [3].

9. References

[1] S. Abiteboul, J. McHugh, M. Rys, and J. L. W. Vasilis Vassalos. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[2] V. Apparao et al. Document Object Model (DOM) Level 1 Specification. W3C Recommendation, Oct. 1998.

[3] M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In *SIGMOD*, 2003.

[4] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-directed publishing with attribute translation grammars. In *VLDB*, 2002.

[5] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie. Query processing in a system for distributed databases (SDD-1). *TODS*, 6(4):602 – 625, 1981.

[6] J. A. Blakeley, N. Coburn, and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *TODS*, 14(3):369–400, 1989.

[7] P. Bohannon, S. Ganguly, H. Korth, P. Narayan, and P. Shenoy. Optimizing view queries in ROLEX to support navigable result trees. In *VLDB*, 2002.

[8] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *VLDB*, 2000.

[9] S. Davidson and A. Kosky. WOL: A language for database transformations and constraints. In *ICDE*, 1997.

[10] K. Dimitrova, M. EL-Sayed, and E. Rundensteiner. Order-sensitive view maintenance of materialized XQuery views. In *ER*, 2003.

[11] M. F. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD*, 2001.

[12] D. Gluche, T. Grust, C. Mainberger, and M. H. Scholl. Incremental updates for materialized OQL views. In *DOOD*, 1997.

[13] A. Gupta and I. Mumick. *Materialized Views*. MIT Press, 2000.

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

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[16] Intelligent Systems Research. XML from databases: ODBC2XML. `http://www.intsysr.com/odbc2xml.htm`.

[17] R. Krishnamurthy, R. Kaushik, J. Naughton, and V. Chakaravarthy. Recursive XML schemas, recursive XML queries, and relational storage: XML-to-SQL query translation. In *ICDE*, 2003.

[18] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in multiview: Strategies and performance evaluation. *TKDE*, 10(5):768–792, 1998.

[19] A. Labrinidis and N. Roussopoulos. "WebView Materialization. In *SIGMOD*, 2000.

[20] A. Labrinidis and N. Roussopoulos. "Update Propagation Strategies for Improving the Quality of Data on the Web". In *VLDB*, 2001.

[21] A. Y. Levy and Y. Sagiv. Queries independent of updates. In *VLDB*, 1993.

[22] J. J. Lu, G. Moerkotte, J. Schu, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *SIGMOD*, 1995.

[23] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. Navigation-driven evaluation of virtual mediated views. In *EDBT*, 2000.

[24] J. Melton, A. Simpson, and J. Gray. *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, 2001.

[25] Oracle. Using XML in Oracle internet applications. `http://technet.oracle.com/tech/xml/`.

[26] Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *ICDT*, 2003.

[27] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.

[28] M. Rys. Bringing the internet to your database: Using SQLServer 2000 and XML to build loosely-coupled systems. In *ICDE*, 2001.

[29] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *ICLP*, 2003.

[30] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *VLDB Journal*, 1996.

[31] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *VLDB*, 1996.

[32] Y. Wang, D. DeWitt, and J. Cai. X-Diff: An effective change detection algorithm for XML documents. In *ICDE*, 2003.

[33] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, 1998.

[34] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, 1995.