DTD-Directed Publishing with Attribute Translation Grammars

Michael Benedikt Chee Yong Chan Wenfei Fan Rajeev Rastogi
Bell Laboratories
{benedikt,cychan,wenfei,rastogi}@research.bell-labs.com

Shihui Zheng Aoying Zhou
Fudan University
{shzheng0,ayzhou}@fudan.edu.cn

Abstract

We present a framework for publishing relational data in XML with respect to a fixed DTD. In data exchange on the Web, XML views of relational data are typically required to conform to a predefined DTD. The presence of recursion in a DTD as well as non-determinism makes it challenging to generate DTD-directed, efficient transformations. Our framework provides a language for defining views that are guaranteed to be DTD-conformant, as well as middleware for evaluating these views. It is based on a novel notion of *attribute translation grammars* (*ATGs*). An ATG extends a DTD by associating semantic rules via SQL queries. Directed by the DTD, it extracts data from a relational database, and constructs an XML document. We provide algorithms for efficiently evaluating ATGs, along with methods for statically analyzing them. This yields a systematic and effective approach to publishing data with respect to a predefined DTD.

1 Introduction

XML [6] has become the primary standard for data exchange on the Web. To exchange data currently residing in relational databases, one needs to *publish it in XML*, i.e. to transform the data into an XML format. In practice, publishing of relational data is always done with a predefined type, typically a DTD. A community or industry agrees on a certain DTD, and subsequently all members of the community create XML views of their relational data that conform to the DTD [3]. This is common in, e.g., B2B applications and the health-care industry: a hospital needs to extract patient information from its relational store, convert it to an XML format , and send it to an insurance company, with the XML data generated conforming to a DTD defined by the insurance company.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002

The problem can be stated as follows: given a DTD D and a relational schema R, define a view σ such that for any instance I of R, $\sigma(I)$ is an XML document that conforms to D. We refer to this as *DTD directed publishing*. The goal is to provide a DTD-directed publishing system that captures transformations commonly found in practice.

DTD-directed publishing is rather challenging. The presence of disjunction in a DTD leads to difficulties in defining deterministic mappings based on the DTD, while recursion makes for a poor match with the querying facilities of standard relational databases. Recursive DTDs are commonly found in specifications of biomedical [5], protein [20] and chemical data [9], e.g., DNA is specified in terms of clone, clone has subelements gene and DNA, while gene is in turn specified with DNA. As a simple example, let us consider a mild variation of a fragment of the TPC-H relational schema [24] shown in Fig. 1 (with keys underlined). The schema, referred to as R_0, specifies parts, suppliers of those parts, and the composition of a part from other parts. Suppose that one wants to define an XML view that extracts information about parts with the brand "Acme" from the relational database. For each part the view provides the name, suppliers and moreover, the part-hierarchy composing it: its sub-parts, the sub-parts of those sub-parts, and so on. In addition, the XML document generated is to conform to a DTD D_0 given in Fig. 2 (here we omit the description of elements whose type is PCDATA). Observe that D_0 is recursive : part is defined in terms of itself. Moreover, the structure of the `address` is non-deterministic: if the supplier is "domestic", i.e., based in the US, its `address` is simply the `addr` attribute of the `Supplier` relation; otherwise, i.e., if it is "foreign", its `address` consists of the `addr` attribute and its `nation`. Given an instance of R_0, the goal is to generate an XML document of DTD D_0. In the document, parts are nested to an arbitrary level which is not known at compile time, but is rather data-driven, i.e. determined by the relational data. This is an instance of DTD-directed publishing.

```
Supplier (suppkey, name, addr, nationkey)
PartSupp (partkey, suppkey, availqty)
Part (partkey, name, mfgr, brand, size, retail)
MadeOf(partkey1, partkey2)
Nation (nationkey, name, regionkey)
...
```

Figure 1: A fragment of TPC-H relational schema, R_0

```
<!ELEMENT  db          (part*)>
<!ELEMENT  part        (pname, supplier*, part*)>
<!ELEMENT  supplier    (sname, address)>
<!ELEMENT  address     (addr | fornaddr)>
<!ELEMENT  fornaddr    (addr, nation)>
```

Figure 2: A predefined DTD D_0 for publishing data of R_0

To publish relational data with a predefined DTD, we need an XML view definition language that allows the DTD to guide view creation, as well as an efficient implementation of the language. Two of the well-known systems that have been developed for publishing relational data in XML are SilkRoute [13], which is based on the view definition language RXL (abstracted as TreeQL in [3]); and XPERANTO [7], which extends SQL by supporting XML constructors to specify views. However, none of these systems takes DTDs/types into account. There have also been several commercial systems [21, 19, 14] that specify XML views by embedding SQL queries within an XML document template. These can only produce mild variations of a fixed document, and thus cannot support data-driven transformations directed by a predefined DTD, especially when the DTD is recursive and/or non-deterministic. To the best of our knowledge, none of the existing systems is able to support the XML view described above.

Another approach to coping with a predefined DTD is by means of type checking [3]: simply define an XML view and then check whether the view conforms to the DTD. Unfortunately this is impractical since type checking of data-driven transformations, even for simple DTDs, is computationally intractable: co-NEXPTIME for extremely restricted view definitions, and undecidable for realistic views [3] (languages such as XQuery [8] implement only approximate type-checking). Worse still, any approach based on type-checking does not provide any guidance on how to define an XML view that does typecheck.

There has also been a line of work on automated inference of mappings from schema information (e.g., [2, 4, 17]). This approach works well when the source and target schemas involved in translations are similar to each other. If the schemas are dramatically different, or if the view mapping depends on the *application* rather than merely upon the schemas, this process cannot be fully automated.

In this paper, we provide the first systematic method for DTD-directed publishing of relational data in XML, by making three contributions. First, we introduce a novel notion of *attribute translation grammars* (*ATGs*). An ATG is an extension of a DTD by associating attributes and semantic rules (SQL queries) with element types. Given a relational schema R and a DTD D, one can define an ATG that, given an instance of R, generates XML data by first extracting data from the database using the rules, and then tagging the data to create XML elements following the element type definitions of D. ATGs facilitate data-driven transformation by using attributes to pass data (to be used, e.g., in grouping) as well as control down a partially-constructed tree. If the evaluation of the ATG terminates successfully, it yields an XML document that is *guaranteed* to conform to the DTD. As an example, the ATG in Fig. 3 defines the XML view of the TPC-H data described above (see Sec. 3). ATGs are inspired by attribute grammars (see, e.g., [10]), but have significant differences (see Sec. 7 for detailed discussion). To the best of our knowledge, they provide the first language that guides the user in the definition of DTD-conformant views.

Second, we establish results for static (compile-time) analyses of ATGs. These include termination analysis of ATG evaluations and the expressive power of ATGs.

Third, we provide efficient algorithms for evaluating ATGs. We introduce new techniques, based on dynamic programming, which combine query-partitioning with the materialization of intermediate results to generate evaluation plans based on estimates of query cost and data size.

Based on these we have implemented a middleware system, PRATA (Publishing Relational data using Attribute Translation grAmmars), for DTD-directed publishing from relational data to XML. We have been conducting experiments on data sets that include the variant of the TPC-H database mentioned earlier. Our experimental results demonstrate that our algorithms generate efficient evaluation plans.

Organization. Section 2 reviews DTDs. Section 3 introduces ATGs and provides static analyses of ATG evaluation. Sections 4 and 5 develop efficient algorithms for evaluating ATGs. Section 6 presents experimental results. Section 7 addresses issues for further work.

2 Background: DTDs

A Document Type Definition (DTD [6]) is typically represented as an extended context free grammar [15].

A *DTD* is a tuple $D = (Ele, P, r)$, where Ele is a finite set of *element types*; r is a distinguished element type, called the *root type*; P is a set of production rules that define the element types: for each A in Ele, $P(A)$ is a regular expression $\alpha ::= \mathsf{S} \mid B \mid \epsilon \mid \alpha{+}\alpha \mid \alpha,\alpha \mid \alpha^*$, where S denotes the *string* (PCDATA) type, B is a type in Ele, ϵ is the empty word, and "$+$", "," and "$*$" denote disjunction, concatenation and the Kleene star, respectively (here we use "$+$" instead of "|" to avoid confusion). We write $A \rightarrow P(A)$ and refer to it as the *production* of A.

An XML document is typically modeled as a node-labeled tree. An XML tree T *conforms to* a DTD D if its structure is constrained by D as follows: (1) there is a unique node in T labeled with r, namely, the *root*; (2) each node in T is labeled either with an element type A of Ele, called an A *element*, or with S, 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 $P(A)$; (4) each text node carries a string value (PCDATA) and is a leaf of the tree.

To simplify the discussion when it comes to defining ATGs, we do not consider attributes in this paper; but we allow entities (defined in the XML standard [6]) to be used in DTDs. An *entity* B is merely a macro (alias) of a regular expression $P(B)$.

Taking advantage of the notion of entities, we define a *DTD D in normal form* to be $(Ele,\ Ety,\ P,\ r)$, where Ety is a finite set of entities, P defines element types and entities such that for each A in $Ele \cup Ety$, $P(A)$ has the form:

$$\alpha \ ::= \ \mathtt{S} \ | \ \epsilon \ | \ B_1, \ldots, B_n \ | \ B_1 + \ldots + B_n \ | \ B^*$$

where B, B_i are in $Ele \cup Ety \cup \{\mathtt{S}\}$ and $n \geq 1$.

For example, the DTD D_0 given earlier can be converted to a DTD D_0' in the normal form by introducing entities $parts, suppliers$ and rewriting the production of $part$ as:

```
<!ELEMENT part   (suppliers, pname, parts)>
<!ENTITY  suppliers   ''supplier*''>
<!ENTITY  parts       ''part*''>
```

Abusing the notion of XML trees, we define a *virtual* XML tree of a DTD D to be an XML tree conforming to a DTD D' which is obtained from D by treating entities as element types. That is, we allow nodes in a virtual XML tree to be labeled with entities. A virtual XML tree V_T of D can be converted to an XML tree T in time *linear* in the size of V_T, by collapsing entity nodes, i.e., merging each entity node with its parent node until no node is labeled with an entity. We refer to T as a *parse tree of D* and say that it *conforms to D*.

We say that DTDs D and D' are *equivalent* if for any XML tree T, T conforms to D iff T conforms to D'. For example, the DTDs D_0 and D_0' are equivalent.

Proposition 2.1: *For any DTD D there exists a DTD D' in the normal form such that D and D' are equivalent. Moreover, D' can be computed from D in linear time.* □

By Proposition 2.1, in the sequel we shall only consider DTDs in the normal form.

A DTD is said to be *recursive* if it has some element type that is defined in terms of itself, directly or indirectly.

3 Attribute translation grammars

This section introduces ATGs and presents static analyses of ATG evaluation.

3.1 ATGs

Definition 3.1: Let $D = (Ele, Ety, P, r)$ be a DTD and R be a relational schema. An *attribute translation grammar* (*ATG*) σ from R to D, denoted by $\sigma : R \to D$, consists of:

- Grammar: the DTD D.
- Attribute tuples: a tuple of attribute members is associated with each $A \in Ele \cup Ety \cup \{\mathtt{S}\}$. The tuple is called the *attribute* of A and denoted by $\$A$. We will use $\$A.x, \$A.y \ldots$ to denote the members of $\$A$. For the root type r, $\$r$ is empty.
- Rules: a set of *semantic rules*, $rule(p)$, is associated with each production $p = A \to \alpha$ in P. For each $B \in Ele \cup Ety \cup \{\mathtt{S}\}$ that occurs in α, there is a rule that specifies how the values of $\$B$ are evaluated. This generally involves an SQL query on relations of R with $\$A$ as parameters.

We say that σ is *recursive* if D is recursive. □

In an ATG we combine a DTD D with database operations by defining semantic rules in terms of SQL queries [1]. Given a relational database, the evaluation of the ATG yields a parse tree of D, in which nodes carry attributes whose values are computed by the queries on the database. The tree is generated incrementally starting from the root downwards. The generation is data-driven: the children of a node v are populated based on the value of v's attribute.

Recall the XML view (part-hierarchy) for the TPC-H data described in Sec. 1. The ATG $\sigma_0 : R_0 \to D_0'$ depicted in Fig. 3 defines the view. In σ_0, the DTD D_0' is the one in normal form given in Sec. 2.

We next describe more precisely the definitions of semantic rules in an ATG $\sigma : R \to D$. For a production $p = A \to \alpha$ (with element types or entities $B_1, \ldots, B_n$ in α) and for each B_i, we have a function returning values for the attribute $\$B_i$ defined from the attribute $\$A$ of A. More specifically, they are computed by a function $f(y_1, \ldots, y_m)$ of one of the following forms:

$$f(y_1, \ldots, y_m) \ ::= \ (y_1, \ldots, y_m) \ | \ Q(y_1, \ldots, y_m)$$

where $y_1, \ldots, y_m$ are members of $\$A$, which are treated as constant parameters of atomic values; $(y_1, \ldots, y_m)$ simply constructs a single tuple using the parameter values; Q is an SQL query on relations of R, by treating y_i as a constant. For example, referring to σ_0 in Fig. 3, the rule associated with production $parts \to part^*$ defines $\$part$ with a query Q_5 on a TPC-H database, which treats $\$parts.partkey$ as a constant.

With these functions we define $rule(p)$ associated with each production $p = A \to \alpha$ in the DTD D. By Proposition 2.1, we can assume that all DTDs are in normal form. Thus, it suffices to consider p of the following cases:
(1) If α is $\mathtt{S}$ then $\$\mathtt{S}$ must consist of a single member, and $rule(p)$ is defined as

$$\$\mathtt{S} = f(\$A),$$

where f is a function of the form defined above.
(2) If α is $B_1, \ldots, B_n$, then $rule(p)$ consists of

$$\$B_1 = f_1(\$A), \ \ldots, \ \$B_n = f_n(\$A),$$

where f_i is a function as defined above for each $i \in [1, n]$.
(3) If α is $B_1 + \ldots + B_n$ then $rule(p)$ is defined as:

$$(\$B_1, \ \ldots, \ \$B_n) \ = \ \text{case} \ \ Q_c(\$A) \ \ \text{of} \ \ 1\text{: } f_1(\$A); \ \ \ldots; \ \ n\text{: } f_n(\$A),$$

[1]The term “attribute translation grammar” was first used to denote a class of attribute grammars for compiler constructions [16], which are quite different from ATGs.

```
Grammar: D'0
Attribute tuples:
$db        = ()
$part      = (partkey, name)
$parts     = (partkey)
$suppliers = (partkey)
$supplier  = (name, addr, nationkey)
$address   = (tag, addr, nation)
$fornaddr  = (addr, nation)
$sname     = $pname = $nation = $addr = (val)
$S         = (val)
Semantic rules:
db → part*
Q1: $part ← select p.partkey, p.name
            from Part p
            where p.brand = ''Acme''
part → suppliers, pname, parts
Q2: $suppliers = ($part.partkey),
Q3: $pname     = ($part.name),
Q4: $parts     = ($part.partkey)
parts → part*
Q5: $part ← select m.partkey2, p.name
            from Madeof m, Part p
            where m.partkey1 = $parts.partkey and
                  m.partkey2 = p.partkey
suppliers → supplier*
Q6: $supplier ← select s.name, s.addr, s.nationkey
                from Supplier s, PartSupp ps
                where ps.partkey = $suppliers.partkey
                      and ps.suppkey = s.suppkey
suppplier → sname, address
Q7: $sname = $supplier.name,
Q8: $address = select 1 as tag, null as nation,
                      $supplier.addr as addr
               from  Nation n
               where 'USA' = n.name and
                  n.nationkey = $supplier.nationkey
               union
               select 2 as tag, n.name as nation,
                      $supplier.addr as addr
               from  Nation n
               where 'USA' <> n.name and
                  n.nationkey = $supplier.nationkey
address → addr + fornaddr:
Q9: ($addr, $fornaddr) =
    case $address.tag of     /* Qc */
      1: (($address.addr), null)
      2: (null, ($address.addr, $address.nation))
fornaddr → addr, nation:
Q10:  $addr = $fornaddr.addr
Q11:  $nation = $fornaddr.nation
A → S  /* A is one of sname, nname, addr, nation */
   $S = $A.val
```

Figure 3: Example of an ATG, σ_0

where Q_c is a query that returns a value in $[1, n]$, and f_j's are functions as above. That is, $\$B_i$ is assigned with the value of $f_i(\$A)$ if $Q_c(\$A) = i$, and with $null$ otherwise. These are to capture the semantics of the non-deterministic production. We refer to Q_c as the *condition query* of the rule.

(4) If α is B^*, then $rule(p)$ is defined as follows:

$$\$B \leftarrow Q(\$A),$$

where Q is a query as defined above. As will be seen shortly, the rule for B in fact introduces an iteration (loop), which implements the Kleene closure without using an unbounded number of attributes.

Observe that $rule(p)$ in cases (1) to (3) is built up from assignments of the form: $\$B_i = f(\$A)$. Here we require $f(\$A)$ to return a single tuple. In case (4), $rule(p)$ is defined with $\$B \leftarrow Q(\$A)$, where $Q(\$A)$ returns a set of tuples. It assigns each tuple in $Q(\$A)$ to $\$B$, i.e., $\$B$ ranges over each value in $Q(\$A)$. As will be explained shortly, for each $\$A$ tuple, the semantic rule for $\$B$ is triggered. For example, referring to Fig. 3, $\$supplier$ and $\$part$ are defined with the second form (with Q_1, Q_5, Q_6) while the rest are defined with the first form.

Next we give the semantics of the ATG σ by presenting a naive evaluation strategy. This strategy is only intended to give a conceptual view of the meaning of the ATG: practical techniques for evaluation will be discussed at length in Sections 4 and 5.

Given an instance I of the relational schema R, σ is evaluated following an *iterative* semantics. The iteration proceeds top-down: starting at the root type, evaluate semantic rules associated with each element type/entity encountered, and create nodes following the DTD to construct an XML tree. The iteration at each stage produces a (partial) XML tree T_n. At each iteration, we consider a particular leaf node lv tagged with A associated with value $\vec{a}$ from $\$A$. We find the corresponding production $A \rightarrow \alpha$, and trigger the rule associated with the production, substituting the value $\vec{a}$ for the parameters $\$A$ in their functions. The resulting functions compute $\$B_i$ for each B_i in α. For each B_i, its function generates a single tuple as the value of $\$B_i$ (or $null$ for some $\$B_i$ in the case of disjunction) except when the production is of the form $A \rightarrow B_i^*$. For each value in $\$B_i$, we create a B_i node and expand the tree T_n by appending these nodes to T_n as the children of lv. More specifically, we do the following:

(1) For a production $A \rightarrow \mathtt{S}$, recall that $\$\mathtt{S} = f(\$A)$ is its semantic rule. If f returns the empty set or multiple values, then the evaluation aborts; otherwise a text node is created as the only child of lv with $\$\mathtt{S}$ as its PCDATA.

(2) The semantic rules for a production $A \rightarrow B_1, \ldots, B_k$ are defined as: $\$B_1 = f_1(\$A)$, $\ldots$, $\$B_k = f_k(\$A)$. If one of the functions f_i returns the empty set or multiple values, then the evaluation aborts, while if each f_i returns a single value, then a single B_i node is created for each i, carrying the $\$B_i$ value. These nodes are treated as the children of lv, in the order specified by the production.

(3) For a production $A \rightarrow B_1 + \ldots + B_k$, recall that its semantic rule is defined with a case clause. The condition query in the clause is evaluated first, and based on its value, a particular B_i is selected and the corresponding function for computing $\$B_i$ is evaluated. A *single* B_i node is created as the only child of the node lv, carrying the $\$B_i$ value.

(4) For a production $A \rightarrow B^*$, recall that its semantic rule is defined as: $\$B \leftarrow Q(\$A)$. If the query Q returns empty, then no children are appended to lv; otherwise m nodes tagged with B are created, where m is the cardinality of

the output of Q, such that each B node carries a distinct value from the set $\$B$. These nodes are the children of lv.

(5) Nothing needs to be done for a production $A \to \epsilon$.

As a final step, we eliminate nodes tagged with an entity to construct an XML tree conforming to D, as described in Sec. 2. We use $\sigma(I)$ to denote the XML tree.

Observe that each step of the iteration expands the tree strictly following the DTD D. In particular, when D is recursively defined, the data-driven evaluation expands the tree to a level which is determined by the relational data and the semantic rules. It is easy to verify that if the evaluation of the ATG terminates successfully (without aborting), it generates an XML tree that conforms to D. This yields a systematic method of DTD-directed publishing.

Example 3.1: Given a database instance of the schema R_0, the evaluation of the ATG σ_0 in Fig. 3 generates an XML tree that conforms to DTD D'_0 as follows. It first creates the root of the tree, tagged with `db`. It then computes a set of tuples of the form (`partkey, name`) using the query Q_1. For each tuple, a distinct node labeled `part` is created, carrying the tuple as its attribute $\$part$. These nodes are the children of the root. For each `part` node, its `suppliers`, `pname` and `parts` children are created by using the queries Q_2, Q_3 and Q_4, respectively, and by treating members of the tuple in $\$part$ as parameters. Similarly, for each `suppliers` node, its subtree is constructed with the queries Q_6 to Q_{11}; each `pname` node in turn has a text node as its child, which carries $\$part$`.name` as its PCDATA; and at each `parts` node, the query Q_5 is executed with attribute $\$parts$`.partkey` as parameter, and its `part` children are generated, as long as the part identified by $\$parts$.partkey has sub-parts, i.e., Q_5($\$parts$`.partkey`) does not return an empty set. Thus the XML tree generated has an unbounded height determined by the relational data. Observe that the non-deterministic choice at `address` is handled by Q_9, which is specified with a condition query Q_c: "$\$address$`.addr`" that returns either 1 or 2. Also note that the evaluation aborts if the query Q_8 does not return a single tuple; but this will not happen as Q_8 is executed with a particular parameter: $\$supplier$`.addr` (treated as a constant) and $\$supplier$`.nationkey` (a key of the `Nation` relation). As the last step, nodes tagged with `suppliers` and `parts` (entities) are eliminated by merging these nodes with their parents. □

Example 3.1 demonstrates the following: (1) ATGs are capable of expressing recursive XML views. (2) They can also handle non-deterministic DTDs, namely, DTDs defined with disjunction. (3) ATGs capture group-by by passing attributes as parameters, e.g., in Q_5 for `part` and Q_6 for `supplier`, without introducing explicit constructs.

3.2 Static analyses

Correctness and termination of ATGs. The definition of ATGs easily ensures the basic correctness result:

Proposition 3.1: *For any ATG $\sigma : R \to D$ and for any database instance I of R, if σ terminates without aborting on I, then $\sigma(I)$ is an XML tree that conforms to D.* □

Of course, an ATG may not terminate, or may abort. The question thus arises whether or not termination of an ATG evaluation is decidable. The *termination problem for ATGs* is to determine, given any ATG $\sigma : R \to D$, whether σ terminates without aborting on all input database instances of R. Closely related is the *termination problem for ATGs on an individual instance*: given an instance I of R, whether σ terminates without aborting on input I.

Theorem 3.2:

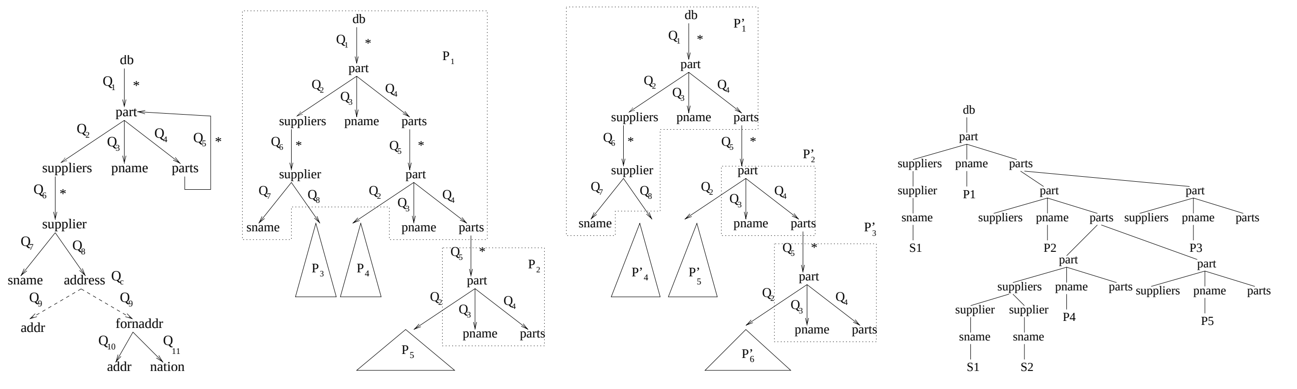
- *The termination problem is decidable for ATGs defined with unions of conjunctive queries and arbitrary DTDs in time exponential in the size of ATG.*
- *On an individual database instance I, the termination problem is decidable for arbitrary ATGs in time polynomial in the size of I.*
- *The termination problem becomes undecidable for ATGs defined with arbitrary SQL queries.* □

The first decidability result shows that it is possible to determine termination for an important class of ATGs. This can be proved by reduction to the satisfiability problem for Datalog programs (with equality and inequality). This result still holds in the presence of key constraints in the underlying relational schema with the same complexity. The second decidability tells us that when the input database instance is fixed, one can effectively determine the termination of the evaluation of an arbitrary ATG. This follows from the iterative semantics given in the last subsection. The final undecidability result says that the termination problem is beyond reach for ATGs defined with general SQL queries. This can be established by reduction from equivalence of SQL queries.

Expressiveness. ATGs are at least as powerful as the view definition languages of the existing publishing systems, namely RXL [13] (TreeQL [3]) in SilkRoute and the language of XPERANTO [7]. That is, for any XML view definable in RXL (TreeQL) or XPERANTO, it can also be expressed as an ATG (with some simple nonrecursive DTD). Moreover, there are ATGs that are not definable in RXL (TreeQL) and XPERANTO, e.g., recursive ATGs.

4 Overview of ATG evaluation

In this section, we provide an overview of an efficient algorithm for evaluating ATGs. As observed by [12], it is advantageous to extract all the relevant relational data first and then construct the final XML document at a later stage. Thus XML view evaluation consists of (1) a *tuple-generation phase* in which relational queries are generated and executed to produce an *output relation* – a relational representation of the view; and (2) a *tagging phase*, where the output relation is post-processed to produce the result XML tree. In systems such as [12, 22] a set of SQL queries



(a) G (b) T and its partition (c) another partition of T (d) XML Document

Figure 4: Example of ATG graph G, its partially unfolded ATG tree T, and the result XML document.

can be produced at compile-time that suffice to compute the output relation. In contrast, it may not be feasible to statically generate queries for recursive ATGs (DTDs). For ATGs we thus require an *iterative tuple-generation approach*: at run-time SQL queries are generated on-the-fly to construct the output relation incrementally; as the iteration proceeds, intermediate results required for later computation need to be maintained. To optimize the evaluation process we devise techniques for selecting certain intermediate results to *materialize* in temporary tables, while simultaneously *unfolding* the recursive rules in the ATG.

4.1 Generation of SQL queries

We illustrate the key ideas underlying our evaluation algorithm using the ATG σ_0 given in Fig. 3, which is an XML view of the TPC-H data (schema R_0). To do so we represent σ_0 as a multi-graph G depicted in Fig. 4(a), referred to as the *ATG graph* of σ_0, which can be easily derived from the DTD D_0' of σ_0. The ATG graph essentially contains a node for each element type/entity A. For each production rule $A \rightarrow \alpha$, there are labeled edges from A to every instance of element type/entity B in α. If $\alpha = B^*$, then the edge has a "$*$" as a label indicating that zero or more B elements can be immediately nested within an A element. Each edge is also labeled with the SQL query for computing the values of the attribute $\$B$ of B (defined using $\$A$). Finally, if α is a disjunction, then the A node is labeled with the condition query in the case clause (its outgoing edges are indicated by dashed lines to distinguish from the case of a concatenation). Note that, as shown in Fig. 4(a), the ATG graph for recursive DTDs contains cycles.

The ATG graph is useful for generating the ATG tree, which is essentially the template for the result XML tree. In the absence of recursion, the ATG tree is constructed by starting with the root node and moving downwards; at each node encountered it creates distinct children of the corresponding node in the ATG graph. For ATG graphs with cycles, this process would not terminate; as a result, when building an ATG tree in the presence of recursion, we only expand nodes to a bounded depth. For instance, Fig. 4(b) illustrates a partial ATG tree T (for the ATG graph in Fig. 4(a)) when `part` is expanded twice. (For simplicity we omit the `address` subtree under `supplier`).

Evaluating the ATG tree formed at a stage of the iteration, i.e., executing the SQL queries and computing attributes associated with the tree, will give a portion of the output relation. Our evaluation strategy, then, is to iteratively unfold the graph into an ATG tree and create SQL queries that append tuples to the output relation. This iteration continues until no leaves of the tree can contribute new tuples to the output relation, i.e., the entire relation has been generated. For example, we unfold and evaluate the ATG graph G of σ_0 until no `part` encountered has sub-parts.

We next consider how to generate, given a (partial) ATG tree T, SQL queries that return output relation tuples. A tuple contains information that can uniquely identify the position and content of a node in the output XML tree, namely, a coding of a root-to-leaf path, and string values for text contents of nodes on the path. This can be done in several ways by varying the sets of SQL queries to be generated. Similar to the approach adopted by SilkRoute [12], we generate queries by first partitioning T into a set of disjoint subtrees referred to as *P-members*, and then producing for each P-member P a single SQL query Q_P such that the composition of Q_P's along a path computes the portion of the output relation corresponding to that path. For example, the ATG tree T in Fig. 4(b) is partitioned into five P-members P_1 to P_5 (the last three P-members rooted at `address` and `suppliers` are not shown). The query S_1 generated for P_1 is given in Fig. 5, which computes one portion of the output relation[2]. In general, the query Q_P for a P-member P can be expressed as an outer union of subqueries corresponding to certain paths in the subtree; such a query is called a *sorted outer union query* in [22]. In particular, S_1 in Fig. 5 is an outer union of two subqueries for the paths from the root to `supplier` and the second `part` in P_1. The subqueries can be easily derived by com-

[2]To avoid cluttering the queries, we have omitted certain auxiliary attributes (that are used for sorting the output) from the select clauses.

```
S_1:
 select p.partkey as partkey, X.suppkey as suppkey,
        X.sname as sname, p.pname as pname,
        null as partkey2, null as pname2
 from   Part p left outer join
        ((select ps.partkey as partkey,
        s.suppkey as suppkey, s.sname as sname
        from PartSupplier ps, Supplier s
        where ps.suppkey = s.suppkey) as X)
        on p.partkey = X.partkey
 where  p.brand = ''Acme''
 union
 select p.partkey as partkey, null as suppkey,
        null as sname, null as pname
        X.partkey2 as partkey2, X.pname2 as pname2
 from   Part p left outer join
        ((select m.partkey1 as partkey1,
        m.partkey2 as partkey2, p2.pname as pname2
        from MadeOf m, Part p2
        where m.partkey2 = p2.partkey) as X)
        on p.partkey = X.partkey1
 where  p.brand = ''Acme''
 order by partkey, suppkey, partkey2
```

Figure 5: SQL query S_1 for P-member P_1 of T in Fig. 4(b).

```
S'_1:  Equivalent to the first subquery of S_1 (Fig.5)
       without null attributes partkey2 and pname
S_2:   select p.partkey, p2.partkey2, p2.pname
       from   Part p, MadeOf m, Part p2
       where  p.brand = ''Acme''
              and m.partkey1 = p.partkey
              and m.partkey2 = p2.partkey
       order by p.partkey, p2.partkey
S'_3:  select p.partkey, m.partkey2, m2.partkey2,
              p2.pname
       from   Part p, Part p2, MadeOf m, MadeOf m2
       where  p.brand = ''Acme''
              and p.partkey = m.partkey1
              and m.partkey2 = m2.partkey1
              and m2.partkey2 = p2.partkey
       order by p.partkey, m.partkey2, m2.partkey2
```

Figure 6: SQL queries for P'_1, P'_2, P'_3 of Fig. 4(c).

posing the SQL queries in the ATG tree[3]: the first subquery is generated by "composing" the queries $\{Q_1, Q_2, Q_3, Q_6, Q_7\}$, and the second one by composing $\{Q_1, Q_4, Q_5, Q_3\}$. The left-outer-join in S_1 ensures that tuples are generated for parts with no suppliers and zero sub-parts. The sorting in S_1 is to facilitate an efficient generation of the output XML data (to be explained shortly).

Note that the tuples computed by S_1 are relatively large (in terms of arity, i.e., the number of attributes) and thus may include many null values. Alternatively, one could reduce the arity of the output tuples by choosing a different partition that produces more subtrees of smaller sizes. To illustrate this, consider another partition shown in Fig. 4(c) which includes six P-members (subtrees, of which only the first three are shown). Here we further partition P_1 of Fig. 4(b) into P'_1 and P'_2. The queries generated for P'_1, P'_2 and P'_3 are given in Fig. 6. The results generated by

[3]By the syntax of ATGs one can show that any function in a semantic rule can be written as an SQL query.

Part

partkey	name	brand
p1	P1	Acme
p2	P2	Bar
p3	P3	Foo
p4	P4	Bar
p5	P5	Foo

MadeOf

partkey1	partkey2
p1	p2
p1	p3
p2	p4
p2	p5

PartSupp

partkey	suppkey
p1	s1
p4	s1
p4	s2

Supplier

suppkey	name
s1	S1
s2	S2

Figure 7: A database instance of the schema R_0

Output for S'_1

partkey	suppkey	sname	pname
p1	s1	S1	P1
p1	s2	S2	P1

Output for S'_2

partkey1	partkey2	pname
p1	p2	P2
p1	p3	P3

Output for S'_3

partkey1	partkey2	partkey3	pname3
p1	p2	p4	P4
p1	p2	p5	P5

Figure 8: Output relations of queries S'_1, S'_2, S'_3.

these queries on an instance (Fig. 7) of the schema R_0 is depicted in Fig. 8 (we show only the relevant relations and attributes of R_0). Observe that the tuples computed by S'_1 and S'_2 have smaller arities than those produced by S_1, and thus contain fewer null values. It should be mentioned that large partitions do not always outperform small ones. We will revisit this issue in Sec. 5, where we present heuristics for finding a good partition.

To correctly combine the results of various queries for the generation of the output XML document, the query for each P-member also needs to include the necessary key attributes along the path from the root of the ATG graph to the root of the P-member. For example, query S'_3 in Fig. 6 includes two additional key attributes (`partkey1` and `partkey2`) corresponding to the top two `part` nodes along the path from `db` to the bottom `part` node in T.

Once all the generated queries have been executed, the output XML document (shown partially in Fig. 4(d)) is generated by joining the output relations for partitions and tagging the resulting tuples based on their key values. This can be done via a simple sequential scan of each output relation outside of the relational database engine, following a top-down approach similar to the conceptual evaluation in Section 3.1.

4.2 Unfolding and materialization

In this subsection, we present two optimization techniques for evaluating ATGs that distinguish our framework from the existing systems. The importance of the proposed optimization is highlighted when ATGs are recursively defined. We should point out that although linear recursive query evaluation is supported by some commercial DBMSs, the

```
S''3:  select t.partkey, t.partkey2, m.partkey2,
              p.pname
       from   Part p, MadeOf m, Temp t
       where  t.partkey2 = m.partkey1
       and    m.partkey2 = p.partkey
       order by t.partkey, t.partkey2, m.partkey2
```

Figure 9: Rewriting of S'_3 using materialized result.

availability of this capability is not adequate to handle the forms of recursion that can arise in recursive DTDs.

The first technique, unfolding, is to address a natural question: How deep should we expand each leaf node in a partial ATG tree? Clearly, it is not practical to fully unfold a (cyclic) ATG graph since we do not know the final structure of the fully unfolded ATG tree in advance. To overcome this, we propose a simple solution of unfolding and partitioning in iterations as follows. Suppose that T is the partially expanded ATG tree at the start of an iteration. For each node in T, the SQL queries executed during the previous iterations generate a set of values in the output relation. We shall refer to those nodes of T with which some non-$null$ tuples are associated as *non-empty* nodes. Our unfolding scheme expands each non-empty leaf v in T up to a maximum depth of d; thus, the depth of the subtree T_v rooted at v does not exceed d, where d is a parameter. Each such subtree T_v (that results from expanding a non-empty leaf v) is then partitioned, and SQL queries for the partition are executed to generate values for nodes in T_v.

Note that there is a tradeoff involved in the number of levels d to unfold a non-empty node. The advantages of unfolding node v by a large number of levels are that the partitioning of T_v can be optimized better due to T_v's larger size. However, a danger with excessive unfolding is that many nodes in T_v may end up being empty, thus causing unnecessary computation. Thus, parameter d must be chosen with care to generate a good plan. Further experimentation is needed to obtain guidelines for the choice of d.

The second technique, query materialization, is to overcome two performance deficiencies. First, as illustrated by queries S'_2 and S'_3 in Fig. 6, there are often common subexpressions shared by the generated queries due to the need to include additional key attributes for sorting the query results. One obvious optimization is to materialize the results of S'_2 so that S'_3 can be rewritten to reference the materialized results. However, the materialized results for S'_2 contain many attributes irrelevant to S'_3. It is more efficient to materialize only a subset of the results of S'_2. For our example, if we materialize the projection on the first two output attributes of S'_2 in a temporary relation `Temp(partkey, partkey2)`, then S'_3 can be rewritten as the query S''_3 shown in Fig. 9 to save one join computation.

Second, as indicated earlier, the number of additional key attributes to be included in the output result for a subtree increases with the "depth" of that subtree. Clearly, this can result in very large composite keys, particularly for recursive DTDs. Thus, in addition to using query materialization to avoid redundant computations of common subexpressions, we can also materialize additional *mapping tables* to map large keys to more concise auxiliary

(a) Mapping table for S'_2

partkey1	partkey2	CKey
p1	p2	1
p1	p3	2

(b) Compressed output for S'_3

CKey	partkey3	pname3
1	p4	P4
1	p5	P5

Figure 10: Use of mapping tables to compress large keys.

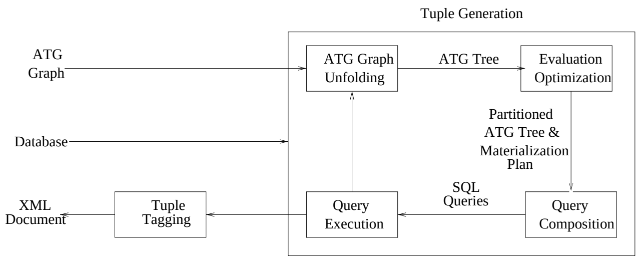


Figure 11: System architecture.

keys so as to improve both storage and processing efficiency. For example, Fig. 10(a) shows a mapping table that maps the composite key of S'_2 to a single auxiliary key attribute $CKey$; this is used to compress the output of S'_3, as illustrated in Fig. 10(b), by replacing the prefix of its composite key with the compressed key. The effect of key compression becomes more evident for long paths, e.g., a single key for $partkey_1, \ldots, partkey_n$ when n is large. Note that the mapping table can be computed as part of the materialized query for avoiding redundant join computations. The compressed keys are chosen to have the same order as the composite keys (e.g., by using a simple counter). This ensures that the inverse mapping from compressed keys to composite keys can be carried out in a single scan of the mapping tables when relations for partitions are joined.

4.3 System architecture

We are now ready to give an overview of the architecture of our DTD-directed publishing system PRATA. As depicted in Fig. 11, it takes an ATG graph G (for $\sigma : R \to D$) and a database instance I of R as inputs and generates an XML document that conforms to D. It consists of a tuple generation phase (indicated by the large outer box) producing output relations, followed by the tuple tagging phase to generate the XML document from the output relations.

The tuple generation phase consists of an iterative sequence of four steps. The first step partially unfolds the ATG graph to an ATG tree T, as described earlier. To optimize the evaluation of T the second step then determines a partitioning of T as well as a set of intermediate queries to be materialized (see Sec. 5). The third step takes as input the partition of T and the materialization plan, and generates a set of SQL queries to evaluate T. The fourth step executes the generated queries to produce the materialized results and output relations. The system iteratively repeats the execution of these steps until the termination condition is met, as described earlier. Finally, the tuple tagging phase uses the output relations, the mapping tables for compressed keys and the DTD associated with the ATG graph to generate the XML document; during this phase additional checks are performed to see if the transformation needs to be aborted (we omit the discussion of these latter checks and the SQL query generation algorithm due to space constraints).

5 Plan generation

As described in the previous section, the evaluation of an ATG graph is carried out in iterations: in each iteration, non-empty leaf nodes of a *partially* expanded ATG tree T are expanded further to a certain depth. Further, for each newly-expanded subtree, an evaluation plan is generated and executed to produce the output tuples for the subtree. In this section we answer a central question for ATG graph evaluation in each iteration: How should we generate an optimal plan to evaluate each expanded subtree?

The goal of plan generation is, given a subtree, to compute a partition for the subtree and a set of nodes to materialize in the subtree such that the cost of executing the SQL queries corresponding to the partition (using the materialized tables) is minimum. Further, as in [12], we would like our plan generation algorithm to be loosely coupled with the underlying relational DBMS, only relying on it for coarse statistics like the cost of executing a query, the query execution plan and the size of the query result.

Clearly, a smaller partition (i.e., fewer P-members) enables the DBMS optimizer to generate better plans for the queries corresponding to the partition. However, as we saw before, with fewer P-members, the size of each query result increases due to the large number of null values for attributes (due to the outer union operation). Thus, our partitioning algorithm must balance the benefit of sharing query computation due to a smaller partition with the larger result sizes of a smaller partition. Similarly trade-offs need to be kept in mind when deciding which subtree nodes to materialize. While materializing intermediate results at a node can reduce the cost of executing queries for descendant P-members[4], there is an overhead with temporary tables (e.g., writing the materialized result to disk) that prevents us from materializing too many nodes. In this section, we present a greedy heuristic that balances the above trade-offs to compute a good partition along with the optimal nodes to materialize for evaluating P-members in this partition. Note that our plan generation algorithm differs from that of the existing systems, where materialization of intermediate results is not considered.

Before we present the algorithm, we formulate the precise optimization problem and develop the necessary notation. Let T denote the partial ATG tree, x be the node just expanded and T_x denote the subtree rooted at x that we want to partition. Consider a partition $\mathcal{P}$ of T_x. Let P_w denote a P-member in $\mathcal{P}$ that is rooted at the node w. We denote by Q_{P_w} the SQL query for P_w, and denote by Q_w the SQL query for materializing node w. For example, referring to Figs. 4(c) and 6, the SQL query for P-member P'_1 is S'_1, while the SQL query for the materialized node labeled `part` in P'_2 is S_2. Further, let $\mathsf{db\_cost}(Q)$ and $\mathsf{db\_card}(Q)$ denote, respectively, the cost for evaluating a query Q and the cardinality of the query result for Q returned by the DBMS optimizer. (Most commercial DBMSs provide support for such statistics). Also, let $\mathsf{num\_attr}(Q)$ denote the number of attributes in the result for Q. Thus, the total size of the result of a query Q is $\mathsf{num\_attr}(Q) \cdot \mathsf{db\_card}(Q)$. Since the estimated total cost of executing query Q_{P_w} involves running it at the DBMS and then retrieving the result tuples (possibly over a network), we model the overall cost without materialization as (similar to [12]):

$$\begin{aligned}\mathsf{tot\_cost}(Q_{P_w}) = \quad & w_1 \cdot \mathsf{db\_cost}(Q_{P_w}) + \\ & w_2 \cdot \mathsf{num\_attr}(Q_{P_w}) \cdot \mathsf{db\_card}(Q_{P_w})\end{aligned}$$

Above, w_1 and w_2 are weight parameters used to vary the trade-off between the cost of query evaluation and the cost of transferring the query result from the database server to the client.

We next compute, for a materialized node v and a descendant P-member P_w of v, the cost of executing Q_{P_w} when Q_{P_w} is rewritten in terms of the materialized table for v. We denote this cost by $\mathsf{tot\_cost}(Q_{P_w}/v)$. Note that the benefit of materializing v for P-member P_w is then given by $\mathsf{tot\_cost}(Q_{P_w}) - \mathsf{tot\_cost}(Q_{P_w}/v)$. As before, we model

$$\begin{aligned}\mathsf{tot\_cost}(Q_{P_w}/v) = \quad & w_1 \cdot \mathsf{db\_cost}(Q_{P_w}/v) + \\ & w_2 \cdot \mathsf{num\_attr}(Q_{P_w}/v) \cdot \mathsf{db\_card}(Q_{P_w}/v)\end{aligned}$$

The functions in the second term can be estimated fairly accurately. Specifically, $\mathsf{db\_card}(Q_{P_w}/v) = \mathsf{db\_card}(Q_{P_w})$ and $\mathsf{num\_attr}(Q_{P_w}/v)$ is essentially $\mathsf{num\_attr}(Q_{P_w}) + 1$ minus the number of ancestor nodes of node v in the ATG tree T. We subtract the number of intermediate nodes between the root and v (and add 1) since in the materialized table for v, all keys for nodes that are ancestors of v in T are replaced with a single auxiliary attribute CKey (due to key compression). Thus, to estimate $\mathsf{tot\_cost}(Q_{P_w}/v)$, we only need to get good estimates for $\mathsf{db\_cost}(Q_{P_w}/v)$. However, estimating $\mathsf{db\_cost}(Q_{P_w}/v)$ accurately is somewhat difficult since our plan generation algorithm is responsible for determining the nodes in T_x to materialize, and thus none of the nodes in T_x are materialized when our algorithm is invoked. A crude approximation that we found to work quite well in our experiments is to simply model $\mathsf{db\_cost}(Q_{P_w}/v)$ as $\mathsf{db\_cost}(Q_{P_w}) - \mathsf{db\_cost}(Q_v)$. This is because in some respect, Q_v is actually a subquery of Q_{P_w} (since P_w is a descendant of v, Q_{P_w} is obtained as a result of query composition with Q_v). A problem with this approximation, however, is that the query plan for Q_v may not match the one for the subquery Q_v in Q_{P_w}. To fix this problem, we add hints to Q_{P_w} so that the execution plan (specially, join order) for query Q_v is forced on the DBMS query optimizer when it generates a plan for Q_{P_w} (current DBMSs provide hooks for specifying hints for preferring certain join orders, e.g., the ORDER keyword in Oracle). This yields a fairly good estimate of $\mathsf{db\_cost}(Q_{P_w}/v)$.

We next turn our attention to the cost of materializing the query for a node v. The attributes for the materialized

[4] A P-member rooted at a node w is a descendant P-member of v iff w is a descendant node of v.

table of v essentially consist of: (1) a single auxiliary key attribute CKey that is a proxy for all the distinct combinations of key values for nodes in the path from the root to v in T, and (2) the attributes in the `select` clause for Q_v that are referenced in the queries relating v to its descendants. Let m_v denote the closest ancestor node of v that has been materialized. Then, we can model the cost of materializing the intermediate table for v using m_v as:

$$\begin{aligned}\mathsf{mat\_cost}(v/m_v) = \;& \mathsf{tot\_cost}(Q_v/m_v) + \\ & w_3 \cdot \mathsf{num\_attr}(Q_v) \cdot \mathsf{db\_card}(Q_v)\end{aligned}$$

where the first term is the cost of evaluating Q_v using the materialized result m_v, and the second term (weighted with another parameter w_3) models the cost of writing to disk (at the DBMS) the materialized table after key compression.

We are now in a position to define the cost of evaluating SQL queries for a partition of T_x when certain nodes in T_x are materialized. For a partition $\mathcal{P}$ of T_x and a set of materialized nodes $\mathcal{M}$ in T_x, we define the cost of evaluating T_x as:

$$\mathsf{cost}(T_x, \mathcal{P}, \mathcal{M}) = \sum_{P_w \in \mathcal{P}} \mathsf{tot\_cost}(Q_{P_w}/m_w) + \sum_{y \in \mathcal{M}} \mathsf{mat\_cost}(y/m_y)$$

Our objective is to compute a partition $\mathcal{P}$ of T_x and a set of nodes $\mathcal{M}$ to materialize in T_x such that $\mathsf{cost}(T_x, \mathcal{P}, \mathcal{M})$ is minimum. Unfortunately, this problem can be shown to be NP-hard (reduction from Set Partition).

In the following, we present a greedy heuristic, Procedure PARTITION, that, given T_x, attempts to find a partition $\mathcal{P}$ of T_x and a set of nodes $\mathcal{M}$ to materialize such that $\mathsf{cost}(T_x, \mathcal{P}, \mathcal{M})$ is small. The heuristic (Fig. 12) starts with each node of T_x as a separate P-member, and in each iteration of the while loop in Step 3, merges a pair of neighboring P-members in $\mathcal{P}$ such that the cost of evaluating the resulting P-members (after merging) is minimum. Of course, for a partition, the cost of evaluation depends on the set of nodes materialized in T_x. Thus, in each iteration, we would like to merge the pair of P-members such that for the resulting partition $\mathcal{P}$, if the optimal set of nodes in T_x are materialized, then the cost of evaluating $\mathcal{P}$ is minimum. In order to determine this optimal set $\mathcal{M}$ of nodes to materialize for a partition $\mathcal{P}$ so that $\mathsf{cost}(T_x, \mathcal{P}, \mathcal{M})$ is minimized, Procedure PARTITION invokes Procedure MATERIALIZE (explained below). Note that Procedure PARTITION terminates once the cost for $\mathcal{P}$ cannot be further reduced by merging the P-members in it.

We now describe the key ideas underlying Procedure MATERIALIZE (given in Fig. 13). Suppose for a partition $\mathcal{P}$ and a node v in T_x, mCost$[v, w]$.mSet denotes the optimal set of nodes to materialize in the subtree T_v rooted at node v, where $m_v = w$. Also, with the nodes in mCost$[v, w]$.mSet materialized, let mCost$[v, w]$.cost be the cost of evaluating the P-members in $\mathcal{P}$ that are (completely) contained within T_v. Let child(v) denote the children of node v. Then, it is possible to compute mCost[.] for v in terms of mCost[.] for its children.

```
procedure PARTITION($T_v, v, u$)
begin
1.  $\mathcal{P} := \{\{x\} : x \in T_v\}$
2.  benefit := $1$
3.  while benefit $> 0$
4.      benefit := $0$
5.      [cost, mSet] := MATERIALIZE($v, u, T_v, \mathcal{P}$)
6.      for each pair of P-members $Q, Q' \in \mathcal{P}$ connected by an edge in $T_v$
7.          $\mathcal{P}' := (\mathcal{P} - \{Q, Q'\}) \cup \{Q \cup Q'\}$
8.          [cost', mSet'] := MATERIALIZE($v, u, T_v, \mathcal{P}'$)
9.          if (benefit < cost - cost')
10.             benefit := cost - cost'
11.             pp := $(Q, Q')$
12.     if benefit $> 0$
13.         $Q :=$ pp$[1] \cup$ pp$[2]$
14.         $\mathcal{P} := \mathcal{P} - \{$pp$[1]$, pp$[2]\}$
15.         $\mathcal{P} := \mathcal{P} \cup \{Q\}$
16. $[\mathcal{C}, \mathcal{M}]$ := MATERIALIZE($v, u, T_v, \mathcal{P}$)
17. return $[\mathcal{P}, \mathcal{M}]$
end
```

Figure 12: Partitioning algorithm

$$\mathsf{mCost}[v,w].\mathsf{cost} = \begin{cases} \min\{\sum_{y \in \mathsf{child}(v)} \mathsf{mCost}[y,w].\mathsf{cost}, \\ \mathsf{mat\_cost}(v/w) + \sum_{y \in \mathsf{child}(v)} \mathsf{mCost}[y,v].\mathsf{cost}\} \\ \qquad \textbf{if } v \textbf{ is not the root of a P-member } P_z \in \mathcal{P} \\ \min\{\mathsf{tot\_cost}(Q_{P_z}/w) + \\ \quad \sum_{y \in \mathsf{child}(v)} \mathsf{mCost}[y,w].\mathsf{cost}, \\ \mathsf{tot\_cost}(Q_{P_z}/v) + \mathsf{mat\_cost}(v/w) + \\ \quad \sum_{y \in \mathsf{child}(v)} \mathsf{mCost}[y,v].\mathsf{cost}\} \\ \qquad \textbf{otherwise} \end{cases}$$

In the two equations above, the two terms in each $\min\{.\}$ expression correspond to the two cases in which v is not materialized (in which case w stays the closest materialized ancestor for each child y of v) or v is materialized (in which case v becomes the new closest materialized ancestor for each child y). Note that for the case when v is materialized using w, an additional cost of $\mathsf{mat\_cost}(v/w)$ is added to the sum of the costs for all children y. Further, if v is the root node for a P-member $P_z \in \mathcal{P}$ (second equation), then we also need to include the cost of evaluating P-member P_z, $\mathsf{tot\_cost}(Q_{P_z}/w)$ and $\mathsf{tot\_cost}(Q_{P_z}/v)$ for the two cases when v is not materialized and materialized, respectively (since this cost is not included in the costs for v's children). Comparing the costs in the two cases, it is possible to determine whether or not to materialize v. Procedure MATERIALIZE uses the above equation to recursively compute mCost$[\cdot]$.mCost and mCost$[\cdot]$.mSet, and returns these to Procedure PARTITION.

Our evaluation procedure will thus proceed by iteratively unfolding the ATG graph and calling Procedure PARTITION for each non-empty leaf v (using T_v and the nearest previously materialized ancestor u as arguments). The worst-case time complexities of Procedures MATERIALIZE and PARTITION are $O(n^2)$ and $O(n^4)$, respectively, where n is the number of nodes in T_v.

6 Experiments

In this section, we present experimental results on the performance of our ATG evaluation algorithms. One of the novel aspects that distinguishes our ATG-based XML publishing approach from previous work (e.g., [12]) is the ma-

```
procedure MATERIALIZE($x$,$y$,$T$,$\mathcal{P}$)
begin
1.  if mCost[$x, y$].computed = true
2.      return [mcost[$x, y$].cost, mcost[$x, y$].mSet]
3.  cost1 := 0
4.  mSet1 := $\emptyset$
5.  cost2 := mat_cost($x/y$)
6.  mSet2 := $\{x\}$
7.  for each child $w$ of node $x$ in tree $T$
8.      [cost, mSet] := MATERIALIZE($w$,$y$,$T$,$\mathcal{P}$)
9.      cost1 := cost1 $+$ cost
10.     mSet1 := mSet1 $\cup$ mSet
11.     [cost, mSet] := MATERIALIZE($w$,$x$,$T$,$\mathcal{P}$)
12.     cost2 := cost2 $+$ cost
13.     mSet2 := mSet2 $\cup$ mSet
14. if $x$ is the root of a P-member, say $P_Z \in \mathcal{P}$
15.     cost1 := cost1 $+$ tot_cost($Q_{P_z}/y$)
16.     cost2 := cost2 $+$ tot_cost($Q_{P_z}/x$)
17. if (cost1 $<$ cost2)
18.     mCost[$x, y$].cost := cost1
19.     mCost[$x, y$].mSet := mSet1
20. else
21.     mCost[$x, y$].cost := cost2
22.     mCost[$x, y$].mSet := mSet2
23. mCost[$x, y$].computed := true
24. return [mcost[$x, y$].cost, mcost[$x, y$].mSet]
end
```

Figure 13: Procedure MATERIALIZE

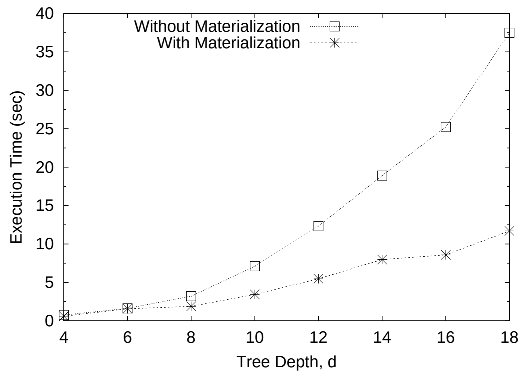
terialization of queries for intermediate nodes of an ATG tree. As described in Sec. 4.2, such materialization has the potential to improve overall system performance, since queries for descendant partitions of a node can be rewritten in terms of the materialized table for the node. Thus, the computation (e.g., joins) performed in materializing the table for a node is shared among the node's descendants.

The results of our experiments presented in this section support the above thesis, and demonstrate that judiciously materializing a few selected internal nodes of the ATG tree can indeed significantly reduce evaluation time. This is most noticeable for ATG trees that are deep, which is frequently the case with recursive ATGs. Thus, we expect that existing XML publishing systems like SilkRoute [12] (that partition the *view* tree[5], but do not materialize intermediate results) can benefit from incorporating materialization.

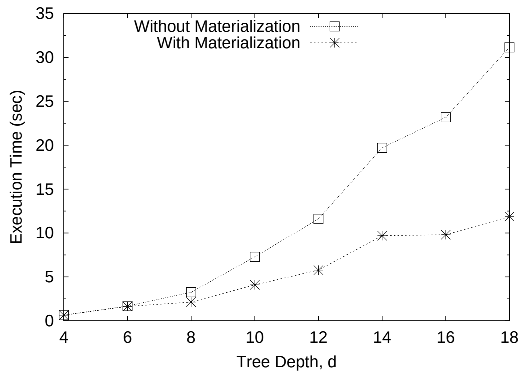
In our experiments, we used a variant of the TPC-H relational schema presented earlier in Fig. 1. Except for the table `MadeOf`, the rest of the tables are generated with TPC's `dbgen` utility using a scale factor of 0.1, where the cardinalities of the `Part`, `Supplier`, and `PartSupp` tables are $20K$, $1K$, and $80K$, respectively. The `MadeOf` table, which has a cardinality of about $80K$, is generated randomly such that each part has at most four sub-parts and the maximum height of each part hierarchy is at most 10. We measured the query execution time to generate the output relations for the portion of the ATG shown in Fig. 3 that involves only elements `part`, `supplier`, `pname` and `sname`. Thus, the ATG graph is essentially identical to the one shown in Fig. 4(a) without `address` and its subelements.

Our experiments were conducted with a database client consisting of a simple embedded SQL program, submitting queries to a database server (via a JDBC interface) on a $1.4GHz$ Pentium IV machine with $256MB$ of main memory running Windows 2000[6].

Fig. 14 depicts the impact of materialization as a function of the tree depth d for two different weight configurations. For each depth value d, we first created a ATG tree with d levels by unfolding the ATG graph $(d-4)/2$ times and then evaluated the ATG tree both with and without materialization to compare the benefits of materialization. The evaluation time measures both the time to materialize intermediate results (for the case with materialization) as well as the time to execute the queries. The results in the figure indicate that materialization can speed up the evaluation by a factor of up to almost 3.5. Furthermore, as we expected, the benefit of materialization generally increases as the tree depth increases: the materialized nodes in a larger ATG tree can benefit more P-members (i.e., the precomputed results are reused more often) thus resulting in more significant improvements. We have also explored the sensitivity



(a) $w_1 = 100, w_2 = 1, w_3 = 10$



(b) $w_1 = 100, w_2 = 100, w_3 = 100$

Figure 14: Benefits of Materialization.

[5]The view tree is similar to our ATG tree, except that it is not dynamically expanded in [12].

[6]Due to licensing restrictions, we are not permitted to identify the commercial product used.

of our plan generation algorithms to the various parameters (e.g., weights w_1, w_2, w_3). As indicated by Fig. 14, the benefits of our evaluation algorithms are rather robust to the changes of these parameters.

7 Conclusion

In this paper we have proposed a formalism, ATGs, for publishing relational data in XML with respect to a predefined DTD, and we have given efficient algorithms for evaluating ATGs. The middleware we have developed, PRATA, is to our knowledge the first system guaranteeing DTD-conformance. Our experimental results indicate that the optimization techniques introduced for PRATA are not only effective in speeding ATG evaluation, but are also useful in the context of existing publishing systems.

There are key differences between ATGs and traditional attribute grammars (AGs, see, e.g., [10]). A traditional AG is defined with a context free grammar (without Kleene star) and more complicated attributes (synthesized and inherited). It takes a string as an input, parses the string with the grammar, and computes attributes. In contrast, it is not possible to "parse" a relational database with a DTD; thus an ATG extracts relevant data from the database via queries, and then constructs a parse tree of the DTD using the data. There have also been applications of AGs to databases, e.g., for constructing query automata [18] and for querying text files [1]. These are mild variations of traditional AGs and are quite different from ATGs.

It is straightforward to extend our framework to handle DTD-directed transformations from Object-Oriented databases to XML, and XML-to-XML transformations. There are extensions that are more involved, and which are the subject of ongoing work. One involves supporting the synthesized attributes found in traditional attribute grammars. The extra expressive power of this extension needs to be examined, as well as its impact on ATG evaluation. We are also studying the extension of this technique from DTDs to XML Schema [23]. A specification (schema) in XML Schema typically consists of a type and a set of integrity constraints. In this context, schema-directed mapping is to define an XML view of relational data such that the XML documents generated both conform to the type and satisfy the constraints. Unfortunately, it is impossible even to decide whether or not a schema is consistent [11], i.e., there is any document satisfying it, due to the interaction between integrity constraints and types in XML Schema. We are working on identifying practical restrictions on XML Schema for effective schema-directed publishing. Another topic is to explore the evaluation of XML queries (e.g. XQuery [8]) against ATG-defined views. Finally, we are also studying methods for capturing information-preserving transformations via ATGs.

ACKNOWLEDGMENT: We thank Mary Fernández for answering several inquiries regarding SilkRoute. Wenfei Fan is on leave from Temple University and is supported in part by NSF Career Award IIS-0093168.

References

[1] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *VLDB*, 1993.

[2] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *ICDT*, 1997.

[3] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. Typechecking XML views of relational databases. In *Proc. of Logic in Computer Science (LICS)*, 2001.

[4] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *ICDT*, 1999.

[5] BIOML. http://www.bioml.com/BIOML.

[6] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C, 1998.

[7] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB*, 2000.

[8] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001. `http://www.w3.org/TR/xquery`.

[9] CML. `http://www.xml-cml.org`.

[10] P. Deransart and M. Jourdan (eds). Attribute Grammars and their Applications. *LNCS 461*, 1990.

[11] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. In *PODS*, 2001.

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

[13] M. F. Fernandez, W. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. In *WWW*, 2000.

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

[15] P. Kilpelainen and D. Wood. SGML and exceptions. In *PODB*, 1996.

[16] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed translations. *JCSS*, 9(3):279–307, 1974.

[17] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[18] F. Neven and J. V. den Bussche. Extensions of attribute grammars for structured document queries. *JACM*, 49(1):56–100, 2002.

[19] Oracle. Using XML in Oracle internet applications. `http://technet.oracle.com/tech/xml/info/htdocs/otnwp/about_xml.htm`.

[20] ProML. http://cartan.gmd.de/promlweb.

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

[22] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2-3):133–154, 2001.

[23] H. Thompson et al. XML Schema. W3C Recommendation, May 2001. `http://www.w3.org/XML/Schema`.

[24] Transaction Processing Performance Council. TPC-H Benchmark. `http://www.tpc.org`.