Capturing both Types and Constraints in Data Integration

Michael Benedikt[†], Chee-Yong Chan[*], Wenfei Fan[†], Juliana Freire[‡], Rajeev Rastogi[†]
[†]Bell Laboratories, Lucent Technologies
[*]SoC, National University of Singapore
[‡]OGI, Oregon Health & Science University
{benedikt,wenfei,rastogi}@research.bell-labs.com, chancy@comp.nus.edu.sg, juliana@cse.ogi.edu

Abstract

We propose a framework for integrating data from multiple relational sources into an XML document that both conforms to a given DTD and satisfies predefined XML constraints. The framework is based on a specification language, AIG, that extends a DTD by (1) associating element types with semantic attributes (inherited and synthesized, inspired by the corresponding notions from Attribute Grammars), (2) computing these attributes via parameterized SQL queries over multiple data sources, and (3) incorporating XML keys and inclusion constraints. The novelty of AIG consists in semantic attributes and their dependency relations for controlling context-dependent, DTD-directed construction of XML documents, as well as for checking XML constraints in parallel with document-generation. We also present cost-based optimization techniques for efficiently evaluating AIGs, including algorithms for merging queries and for scheduling queries on multiple data sources. This provides a new grammar-based approach for data integration under both syntactic and semantic constraints.

1. Introduction

Data exchange applications frequently require enterprises to integrate data from different relational sources for export as an XML document. The integrated XML document is typically required to conform to a predefined "schema". Typically a schema consists of two parts: a type specification and a set of integrity constraints, e.g., a specification in XML Schema [28]; thus, the integrated data should both conform to the type and satisfies the constraints. Here we consider DTDs for XML types, and keys and inclusion constraints (a generalization of foreign keys) for XML constraints, which represent the schema features most common in current practice.

Example 1.1: Let us consider data exchange between an insurance company and a hospital. The hospital maintains four relational databases: one containing patient information, one indicating whether a treatment is covered by an insurance policy, one storing billing information, and one describing treatment procedures – a treatment may require a procedure consisting of other treatments. The databases are specified by the following schemas (with keys underlined):

DB_1: patient information
patient(SSN, pname, policy),   visitInfo(SSN, trId, date)
DB_2: insurance coverage
cover(policy, trId)
DB_3: billing information
billing(trId, price)
DB_4: treatment procedure hierarchy
treatment(trId, tname),   procedure(trId1, trId2)

The hospital sends a daily report to the insurance company that includes information on patients and their treatments for that day. The insurance company requires the report to be in an XML format conforming to a fixed DTD D (here we omit the definition of elements whose type is PCDATA):

```
<!ELEMENT  report   (patient*)>
<!ELEMENT  patient  (SSN, pname, treatments, bill)>

<!ELEMENT  treatments  (treatment*)>
<!ELEMENT  treatment   (trId, tname, procedure)>
<!ELEMENT  procedure   (treatment*)>

<!ELEMENT  bill        (item*)>
<!ELEMENT  item        (trId, price)>
```

An XML report conforming to D is depicted in Fig. 1. It contains information from the database DB_1 about `patients` treated on the day. For each `patient`, the `treatments` subtree describes the treatments covered by the patient's insurance policy as well as the procedure hierarchy of each of these treatment, using the data in DB_1, DB_2 and DB_4; the `bill` subtree collects the price for *each* treatment appearing in the `treatments` subtree, using DB_3, for the reference of the insurance company.

Observe the following. First, the integration requires *multi-source* queries, i.e., queries on multiple databases. For example, to find the treatments of a patient covered by insurance, one needs a query on both DB_1 and DB_2. Second, `treatment` is defined *recursively*, i.e., in terms of itself. Thus, a `treatment` subtree may have an unbounded depth that cannot be determined at compile-time but is rather *data-driven*, i.e., determined by the data in DB_4. Third, the construction of the `bill` subtree of a `patient` is *context-dependent*: it is determined by the `trId`s collected in the `treatments` subtree of the patient. In fact in many XML-based information integration tasks the natural flow of tree construction is not strictly top-down, but may rather require pushing information calculated during construction of one part of the tree over to another part. The arrows in Fig. 1 indicate the information flow for this example.

[*]Work done while the author was with Bell Labs.

SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06 ...$5.00.

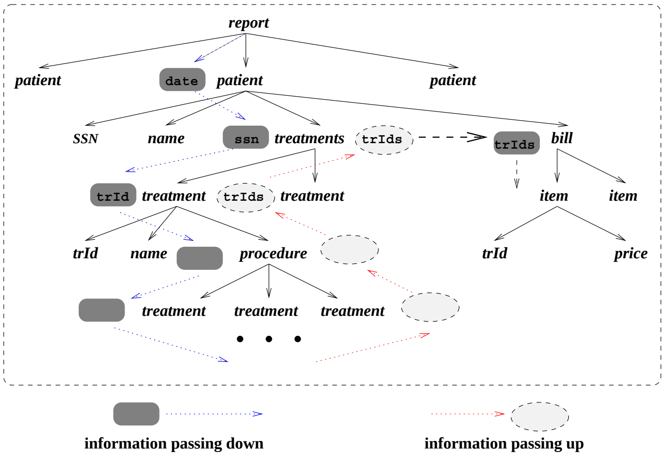


Figure 1: Example of XML report

Furthermore, the insurance company imposes the following constraints on the XML reports:

Key: `patient` (`item.trId` → `item`)
IC: `patient` (`treatment.trId` ⊆ `item.trId`)

The first constraint asserts that in *each* subtree rooted at a `patient`, `trId` is a *key* of `item` elements: for any two items, if their `trId` subelements have the same value, then the two must be the same element. That is, each treatment is charged only once for each `patient`. The second one is an *inclusion constraint*; it states that under each `patient`, for any `treatment` element there must be an `item` such that the values of their `trId` subelements are equal. This specifies an *inter-database* constraint: each treatment in DB_4 must have a corresponding price entry in DB_3. □

One goal of this work is to develop a language with which, given a collection R of relational databases (sources), a DTD D and a set Σ of XML constraints, one can specify an integration (mapping) σ such that $\sigma(R)$ is an XML document that both conforms to D and satisfies Σ. Furthermore, the language should be capable of specifying integration tasks commonly found in practice, such as those involving recursive DTDs, multi-source queries and the context-dependent constructions illustrated by the example.

To this end we propose a formalism, called *Attribute Integration Grammar* (AIG), to specify data integration in XML. An AIG consists of two parts: a grammar and a set Σ of XML constraints. Inspired by attribute grammars [13], the grammar extends a DTD D by associating semantic attributes and semantic rules with element types. The semantic attributes, which can be either *synthesized* (evaluated bottom-up) or *inherited* (evaluated top-down), are used to pass data and control during AIG evaluation. The semantic rules compute the values of the attributes by extracting data from databases via multi-source SQL queries. A query for computing an attribute may take other attributes as parameters. This introduces a dependency relation on attributes, and thus the power to specify sophisticated control flow within the AIG evaluation as well as context-dependent construction of elements. The constraints Σ are compiled into relations on synthesized attributes and their satisfaction checking is embedded into the attribute evaluation. As a result, the XML document is constructed via a controlled derivation from the grammar and constraints, and is thus guaranteed to both conform to D and satisfy Σ.

As an example, Fig. 2 gives an AIG, σ_0, specifying the data integration described in Example 1.1. We defer the explanation of σ_0 to Section 3 where we formally define AIGs.

Based on AIGs, we develop a middleware system to integrate relational data. The system implements a variety of optimization techniques to evaluate AIGs efficiently. At compile time, it converts XML constraints into relations on synthesized attributes, reduces attribute dependencies to a minimum, and rewrites multi-source queries into single-source ones while remaining in the AIG framework. At run time, using basic database statistics, it generates execution plans that involve scheduling and merging single-source queries to maximize parallelism among underlying relational engines and to reduce response time.

Contributions. We propose a grammar-based approach to integrating relational data in XML under syntactic and semantic constraints.

- We introduce a specification language AIG, which provides a systematic method for supporting DTD conformant and context-dependent integration.
- We show that XML constraints can be captured in the same framework via constraint compilation.
- We show how to support multi-source queries within our specification and optimization framework.
- We adapt query scheduling and merging techniques for efficiently evaluating AIGs.

Related work. Little previous work has addressed the issue of DTD conformance and constraint satisfaction for XML-based data integration. Clio [23] derives schema and data mappings from inter-schema constraints; it is unclear how it can ensure DTD conformance automatically. Results on the existence of mappings satisfying those constraints are explored in [14]; this is inspired by Clio, but deals only with the relational setting. MIX [2] addresses DTD checking/inference for XML integration, but the inference process is expensive, and does not provide any guidance for how to define a mapping that type checks [22]; moreover, it does not consider constraints. XML publishing systems such as SilkRoute [15] and XPERANTO [27] consider only a single relational source and do not take DTDs and constraints into account. Earlier rule-based systems [10, 12, 20] rely on object identifiers to model recursive data structures; it is unclear how they can handle recursive DTDs and context-dependent integration of multiple databases. Data integration mappings can be specified in Turing Complete transformation languages such as XQuery [7] and XSLT [9]; but optimization for these languages still remains to be explored, and their complexity makes it desirable to work within a more limited formalism. Our contribution is complementary to work on schema mapping (see [24] for a survey), which focuses on schema translation rather than providing support for explicit specification and evaluation of data integration.

Closest to our work is [3], which presents a formalism for publishing relational data in XML with respect to a fixed DTD. AIGs extends the formalism of [3] in several nontrivial aspects. First, AIGs support multiple data sources. Second, AIGs support both synthesized and inherited attributes; this introduces the expressive power necessary for dealing

```
Semantic attributes:
   Inh(report)   = (date)
   Inh(patient)  = (date, SSN, pname, policy)

   Inh(treatments) = (date, SSN, policy)
   Syn(treatments) = Syn(treatment)
                   = Syn(procedure) = (set(trIdS))
   Inh(treatment)  = (trId, tname)
   Inh(procedure)  = (trId)

   Inh(bill) = (set(trIdS))
   Inh(item) = (trId, price)
   Inh(SSN)  = Inh(pname) = Inh(trId) = Inh(price)
             = Inh(tname) = Inh(S) = Syn(trId) = (val)

Semantic rules:
 report $\rightarrow$ patient*
  Inh(patient).date = Inh(report).date
  Inh(patient).(SSN,pname,policy) $\leftarrow$ $Q_1$(Inh(report))
  where $Q_1(v)$:
    select  p.SSN, p.pname, p.policy
    from    $DB_1$:patient p, $DB_1$:visitInfo i
    where   p.SSN = i.SSN and i.date = $v$.date

 patient $\rightarrow$ SSN, pname, treatments, bill
  Inh(SSN).val = Inh(patient).SSN,
  Inh(pname).val = Inh(patient).pname,
  Inh(treatments) = Inh(patient)(date,SSN,policy)
  Inh(bill).trIdS = Syn(treatments).trIdS

/* The treatments subtree of a patient */
 treatments $\rightarrow$ treatment*
  Inh(treatment).(trId,tname) $\leftarrow$ $Q_2$(Inh(treatments))
  where $Q_2(v)$:
   select t.trId, t.tname
   from   $DB_1$:visitInfo i, $DB_2$:cover c,
          $DB_4$:treatment t
   where  i.SSN = $v$.SSN and i.date = $v$.date and
          t.trId = i.trId and c.trId = i.trId
          and c.policy = $v$.policy
  Syn(treatments).trIdS = $\bigcup$ Syn(treatment).trIdS

 treatment $\rightarrow$ trId, tname, procedure
  Inh(trId).val = Inh(treatment).trId
  Inh(tname).val = Inh(treatment).tname
  Inh(procedure).trId = Inh(treatment).trId
  Syn(treatment).trIdS = Syn(procedure).trIdS
                         $\cup$ {Syn(trId).val}

 procedure $\rightarrow$ treatment*
  Inh(treatment).(trId, tname) $\leftarrow$ $Q_3$(Inh(procedure))
  Syn(procedure).trIdS = $\bigcup$ Syn(treatment).trIdS
  where $Q_3(v)$:
   select  p.trId2, t.tname
   from    $DB_4$:procedure p, $DB_4$:treatment t
   where   p.trId1 = $v$.trId and t.trId = p.trId2

 trId $\rightarrow$ S
  Syn(trId).val = Inh(trId).val
  Inh(S).val = Inh(trId).val

/* The bill subtree of a patient */
 bill $\rightarrow$  item*:
  Inh(item).(trId, price) $\leftarrow$ $Q_4$(Inh(bill).trIdS)
  where $Q_4(V)$:
    select trId, price
    from   $DB_3$:billing
    where  trId in $V$

 item $\rightarrow$  trId, price:
  Inh(trId).val = Inh(item).trId,
  Inh(price).val = Inh(item).tname


XML constraints:
  patient (item.trId $\rightarrow$ item)
  patient (treatment.trId $\subseteq$ item.trId)
```

Figure 2: Example AIG σ_0

with XML constraints and context-dependent information-passing during document generation. Third, AIG incorporates into the integration process XML constraints on the target document, which are not considered in [3]. Due to the increased expressiveness, the evaluation of AIGs demands techniques far beyond those reported in [3]: it requires the analysis of inter-query dependencies, constraint compilation, rewriting of multi-source queries into single-source ones, grouping and scheduling of queries based on the data sources; the top-down evaluation strategy of [3] no longer works in this context.

Organization. Section 2 reviews DTDs and XML constraints. Section 3 defines AIGs, followed by static analyses of AIGs in Section 4. Section 5 presents evaluation techniques for AIGs. Section 6 presents experimental results; and Section 7 identifies future research topics.

2. Schema Specification

Schema specifications for XML data normally make use of DTDs and XML constraints.

DTDs. A *DTD* (Document Type Definition [4]) can be represented as a tuple $D = (Ele, P, r)$, where Ele is a finite set of *element types*; r is a distinguished type in Ele, called the *root type*; P defines the element types: for each A in Ele, $P(A)$ is a regular expression and $A \rightarrow P(A)$ is called the *production* of A. To simplify the discussion we consider $P(A)$ of the form:

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

where S denotes the *string* (PCDATA) type, ϵ is the empty word, B is a type in Ele (referred to as a *subelement type* of A), and "+", "," and "$*$" denote disjunction, concatenation and the Kleene star, respectively (here we use "+" instead of "|" to avoid confusion).

Recall that an XML document (tree) T *conforms to* a DTD D if the following conditions are met: (1) there is a unique node, the *root*, in T labeled with r; (2) each node in T is labeled either with an Ele type A, 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 $P(A)$; and (4) each text node carries a string value (PCDATA) and is a leaf of the tree.

The simplification of the $P(A)$'s above does not lose generality because of the following facts, which are straightforward to verify: (1) by introducing entities [4] (macros of a regular expression), a DTD D_1 with general regular expressions can be converted to D_2 of the form given above in linear time; and (2) any XML document conforming to D_2 can be converted to one conforming to D_1 in linear time (by eliminating entities), and vice versa (by labeling entities).

To avoid overloading the notions used in AIGs, we do not consider DTD attributes in this paper; but it is trivial to extend our framework to incorporate them.

XML constraints. We consider XML keys and inclusion constraints of [1], defined in the presence of a DTD D as follows:

(1) *Key:* $C(A.l \rightarrow A)$, where C, A and l are element types such that in D, l is a unique string-subelement type of A, i.e., $P(l) = \mathtt{S}$ and l is unique in $P(A)$. An XML tree T *satisfies* the key iff in any subtree T_c of T rooted at a C element, for

any two A elements x, y in T_c, if the string values of their l subelements are equal, then x and y are the same node. That is, the l-subelement value of an A element a uniquely identifies a among all the A elements in T_c. We say that l is a *key of A elements relative to C elements*.

(2) *Inclusion constraint (IC):* $C(B.l_B \subseteq A.l_A)$, where C, A, B, l_A and l_B are element types of D such that l_A and l_B are of string type in $P(A)$ and $P(B)$, respectively. An XML tree T *satisfies* the IC iff in any subtree T_c of T rooted at a C element, for any B element b there exists an A element a in T_c such that the value of any l_B subelement of b equals the value of an l_A subelement of a.

We have seen XML keys and inclusion constraints in Example 1.1. A cursory examination of existing XML specifications reveals that most XML constraints are of this form. In particular, foreign keys are supported: a foreign key is a pair consisting of a key $C(A.l_A \to A)$ and an IC $C(B.l_B \subseteq A.l_A)$.

To simplify the discussion we consider XML constraints defined with single subelement. The same framework can be used to handle constraints in XML Schema [28].

3. Attribute Integration Grammars

This section defines the syntax and semantics of AIGs.

3.1 Definition of AIGs

Definition 3.1: Given a DTD $D = (Ele, P, r)$, a set Σ of XML constraints and a collection $\mathcal{R}$ of relational schemas, an *attribute integration grammar* (*AIG*) σ from $\mathcal{R}$ to D, denoted by $\sigma : \mathcal{R} \to D$, is defined to be a triple of:

1. Attributes: two disjoint tuples of attributes are associated with each $A \in Ele \cup \{\mathtt{S}\}$, called the *inherited* and *synthesized attributes* of A and denoted by $Inh(\mathtt{A})$ and $Syn(\mathtt{A})$, respectively. We use $Inh(\mathtt{A}).x$ (resp. $Syn(\mathtt{A}).y$) to denote a member of $Inh(\mathtt{A})$ (resp. $Syn(\mathtt{A})$).

   Each attribute member has either a *tuple type* (tuple of strings) $(a_1, \ldots, a_k)$ where the a_i's are distinct names, or a set type $set(a_1, \ldots, a_k)$ denoting a set of tuples with components a_i's.

2. Rules: a set of *semantic rules*, *rule*(p), is associated with each production $p = A \to \alpha$ in P. In *rule*(p), (1) there is a rule for computing the values of $Syn(\mathtt{A})$ by combining the values of $Syn(\mathtt{B_i})$ for each B_i in α. and (2) for each element type B that occurs in α, there is a rule for computing $Inh(\mathtt{B})$ by means of an SQL query on multi-databases of $\mathcal{R}$ and using $Inh(\mathtt{A})$ and $Syn(\mathtt{B_i})$ as parameters. Here B_i's are the element types other than B mentioned in α.

   The *dependency relation* of p is the transitive closure of the following relation: for any B, B' in α, B depends on B' iff $Inh(\mathtt{B})$ is defined using $Syn(\mathtt{B'})$. The dependency relation is said to be *acyclic* iff for any B in α, (B, B) is not in the dependency relation.

3. *Constraints:* XML keys and inclusion constraints of Σ are specified.

The AIG requires that the dependency relation of every production of D is acyclic. □

As an example, Fig. 2 gives an AIG σ_0. Observe that the dependency relations of all the productions of σ_0 are acyclic. In particular, although $Inh(\mathtt{bill})$ is defined using $Syn(\mathtt{treatments})$ in the production for `patient`, the dependency relation is not cyclic since $Inh(\mathtt{treatments})$ is not defined using $Syn(\mathtt{bill})$ directly or indirectly.

Roughly speaking, given a database instance I of $\mathcal{R}$, an AIG extracts data from I using the SQL queries in the rules for inherited attributes, constructs an XML tree with the data directed by the productions of its DTD, and checks whether the XML tree satisfies its constraints. The synthesized attributes collect data computed at each stage and pass it to SQL queries as parameters such that the XML tree can be constructed in a context-dependent and data-driven way. The dependency relations specify an ordering on the data and control flow, which are acyclic to assure that the computation can be carried out. The inherited attribute $Inh(\mathtt{r})$ of the root, referred to as *the attribute of the AIG*, is a global parameter that enables us to compute different XML trees for different input values. Thus an AIG is a mapping: it takes I and $Inh(\mathtt{r})$ as parameters, integrates the data of I following D, and produces an XML tree that both conforms to D and satisfies Σ. For example, given a `date` and DB_1, DB_2, DB_3 and DB_4 described in Example 1.1, the AIG σ_0 generates an XML report for the particular `date` by integrating the relational data.

We next define in detail the semantic rules in an AIG $\sigma : \mathcal{R} \to D$. Consider a production $p = A \to \alpha$ in D with element types $B_1, \ldots, B_n$ in α. We use the notation $\widehat{Syn(\mathtt{B})}$ to denote the vector containing all the synthesized attributes $Syn(\mathtt{B_i})$, and $\widehat{Syn(\mathtt{B_i})}$ to denote the vector of all except $Syn(\mathtt{B_i})$ for fixed i. We use two types of functions g and f for computing the synthesized attribute $Syn(\mathtt{A})$ and the inherited attributes $Inh(\mathtt{B_i})$, respectively:

$$g(Inh(\mathtt{A}), \widehat{Syn(\mathtt{B})}) ::= (x_1, \ldots, x_k) \,|\, \{x\} \,|\, \bigcup x \,|\, x_1 \cup \ldots \cup x_k$$

$$f(Inh(\mathtt{A}), \widehat{Syn(\mathtt{B_i})}) ::= (x_1, \ldots, x_k) \quad | \quad Q(x_1, \ldots, x_k)$$

where $x, x_1, \ldots, x_m$ are members of $Syn(\mathtt{B_j})$ for some j ($j \neq i$ in the second case) and $Inh(\mathtt{A})$, (.) and $\{.\}$ construct a tuple and a set of tuples, respectively; $\cup$ denotes set union, $\bigcup x$ builds a set by collecting all the tuples in x, and Q is an multi-source SQL query over databases of $\mathcal{R}$, which treats members of $Inh(\mathtt{A})$ and $Syn(\mathtt{B_j})$ as parameters (a temporary relation is created in the database if some member is a set). Type *compatibility* is required: the type of $Syn(\mathtt{A})$ must match that of g, i.e., if $Syn(\mathtt{A})$ is of a tuple type $(a_1, \ldots, a_k)$ then g must be defined using (.) and returns tuples of arity k, and if $Syn(\mathtt{A})$ is of type $set(a_1 \ldots a_k)$ then g is defined using $\{.\}, \cup$ or $\bigcup$ and returns a set of tuples of arity k. Similarly for $Inh(\mathtt{B_i})$ and f; in particular, $Inh(\mathtt{B_i})$ is of a set type iff f is defined with a query. It is easy to verify that type compatibility can be checked statically in linear time. Also note that while $Inh(\mathtt{B_i})$ can be defined with queries to extract data from the underlying databases, $Syn(\mathtt{A})$ uses simple tuple and set constructors to combine data computed previously.

More specifically, with these functions we describe *rule*(p) associated with each production $p = A \to \alpha$.

(1) If α is S then *rule*(p) is defined as

$$Syn(\mathtt{A}) = g(Inh(\mathtt{A})), \qquad Inh(\mathtt{S}) = f(Inh(\mathtt{A})),$$

where f, g are as defined above such that f must return a tuple of a single member, i.e., a string, which is treated as the PCDATA. This is one of the two cases where $Syn(\mathtt{A})$ can be defined using $\mathit{Inh}(\mathtt{A})$. An example of rules of this form is the one for $\mathtt{trId} \rightarrow \mathtt{S}$ in the AIG σ_0.

(2) If α is $B_1, \ldots, B_n$, then $\mathit{rule}(p)$ consists of

$$\begin{aligned} &\mathit{Syn}(\mathtt{A}) = g(\widehat{\mathit{Syn}(\mathtt{B})}) \\ &\mathit{Inh}(\mathtt{B_1}) = f_1(\mathit{Inh}(\mathtt{A}), \widehat{\mathit{Syn}(\mathtt{B_1})}) \\ &\ldots \\ &\mathit{Inh}(\mathtt{B_n}) = f_n(\mathit{Inh}(\mathtt{A}), \widehat{\mathit{Syn}(\mathtt{B_n})}) \end{aligned}$$

where g, f_i are as defined above, $\widehat{\mathit{Syn}(\mathtt{B_i})}$ is a list of $\mathit{Syn}(\mathtt{B_j})$'s, which does not include $\mathit{Syn}(\mathtt{B_i})$, and $\widehat{\mathit{Syn}(\mathtt{B})}$ is a list of all $\mathit{Syn}(\mathtt{B_j})$'s. This is the only case where the inherited attribute of B_i can be defined with synthesized attributes of B_i's siblings. For an example see the rules for the `patient` production in σ_0.

(3) If α is $B_1 + \ldots + B_n$ then $\mathit{rule}(p)$ is defined as:

$$\begin{aligned} \mathit{Syn}(\mathtt{A}) = \text{case } &Q_c(\mathit{Inh}(\mathtt{A})) \text{ of} \\ &1\text{: } g_1(\mathit{Syn}(\mathtt{B_1}));\ \ldots;\ \ n\text{: } g_n(\mathit{Syn}(\mathtt{B_n})) \\ (\mathit{Inh}(\mathtt{B_1}), \ldots, \mathit{Inh}(\mathtt{B_n})) = \text{case } &Q_c(\mathit{Inh}(\mathtt{A})) \text{ of} \\ &1\text{: } f_1(\mathit{Inh}(\mathtt{A}));\ \ \ldots;\ \ n\text{: } f_n(\mathit{Inh}(\mathtt{A})) \end{aligned}$$

where Q_c is a query that returns a value in $[1, n]$, referred to as the *condition query* of the rule, and f_j, g_i are as above. If $Q_c(\mathit{Inh}(\mathtt{A})) = i$, then $\mathit{Syn}(\mathtt{A})$ and $\mathit{Inh}(\mathtt{B_i})$ are computed with $g_i(\mathit{Syn}(\mathtt{B_i}))$ and $f_i(\mathit{Inh}(\mathtt{A}))$, respectively, otherwise they are assigned with *null* (or empty set depending on their types). These capture the semantics of the *nondeterministic* production in a data-driven fashion.

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

$$\mathit{Syn}(\mathtt{A}) = \bigcup \mathit{Syn}(\mathtt{B}), \qquad \mathit{Inh}(\mathtt{B}) \leftarrow Q(\mathit{Inh}(\mathtt{A})),$$

where Q is a query. The rule for $\mathit{Syn}(\mathtt{A})$ collects all the synthesized attributes of its children into a set. The rule for $\mathit{Inh}(\mathtt{B})$ introduces an iteration, which creates a B child for each tuple in the output of $Q(\mathit{Inh}(\mathtt{A}))$ such that the child carries the tuple as the value of its inherited attribute. See the rules for the `treatments` production for an example.

(5) If α is ϵ, then $\mathit{rule}(p)$ is defined by

$$\mathit{Syn}(\mathtt{A}) = g(\mathit{Inh}(\mathtt{A})),$$

where g is as defined above. This is the other case where $\mathit{Syn}(\mathtt{A})$ can be defined using $\mathit{Inh}(\mathtt{A})$.

Several subtleties are worth mentioning. First, we restrict data sources to be relational just to simplify the discussion. The same framework can be extended to integrate object-oriented, XML and other formats of data, by expressing queries in, e.g., OQL [6] or fragments of XQuery [7]. Second, AIGs are not a mild variation of attribute grammars (see, e.g., [13]) or their typical applications [21]; they are quite different in semantic definitions and evaluation strategies.

3.2 Conceptual Evaluation

We next give the semantics of an AIG σ by presenting a conceptual evaluation strategy. Here we focus on DTD-directed integration, deferring the discussion of the support for constraints and multi-source queries to Sections 3.3 and 3.4. Optimization techniques for efficient evaluation of AIGs will be discussed in Section 5.

Given database instances I of the schema $\mathcal{R}$ and a value v of the attribute $\mathit{Inh}(\mathtt{r})$ of the AIG, σ is evaluated depth-first, directed by its DTD and controlled by its dependency relation, using a stack. The root node r is first created and pushed onto the stack. For each node lv at the top of the stack, we compute the inherited attribute value $\vec{a}$ of lv, find the production $p = A \rightarrow P(A)$ in the DTD for the element type A of lv, and evaluate $\mathit{rule}(p)$ using $\vec{a}$ as follows:

(1) If $p = A \rightarrow \mathtt{S}$, recall $\mathit{Syn}(\mathtt{A}) = g(\mathit{Inh}(\mathtt{A}))$ and $\mathit{Inh}(\mathtt{S}) = f(\mathit{Inh}(\mathtt{A}))$. A text node is created as the only child of lv with $f(\vec{a})$ as its PCDATA. Then, $g(\vec{a})$ is computed as the synthesized attribute of lv.

(2) If $p = A \rightarrow B_1, \ldots, B_k$, recall $\mathit{Syn}(\mathtt{A}) = g(\widehat{\mathit{Syn}(\mathtt{B})})$ and $\mathit{Inh}(\mathtt{B_i}) = f_i(\mathit{Inh}(\mathtt{A}), \widehat{\mathit{Syn}(\mathtt{B_i})})$. We create a node tagged with B_i for each $i \in [1, n]$. These nodes are treated as the children of lv, in the order specified by the production. Since the dependency relation of p is acyclic, there is a topological order on $B_1, \ldots, B_k$ such that each B_i needs only the synthesized attributes of those preceding it in the order to compute its inherited attribute, while the first one needs $\mathit{Inh}(\mathtt{A})$ only. We push these nodes onto the stack in reverse-topological order, and proceed to evaluate them substituting $\vec{a}$ for $\mathit{Inh}(\mathtt{A})$. After all these nodes are evaluated and popped off the stack, we compute $\mathit{Syn}(\mathtt{A})$ of lv using the function g and the synthesized attributes of these nodes.

(3) If $p = A \rightarrow B_1 + \ldots + B_k$, we first evaluate the condition query, which takes $\mathit{Inh}(\mathtt{A})$, i.e., $\vec{a}$, as a parameter. Based on its value, a particular B_i is selected and the function for computing $\mathit{Inh}(\mathtt{B_i})$ is evaluated, a function depending on $\vec{a}$ only. A *single* B_i node is created as the only child of the node lv, and pushed onto the stack. We then proceed to evaluate the node. After the node is popped off the stack, $\mathit{Syn}(\mathtt{A})$ is computed by applying g_i to $\vec{a}$ and $\mathit{Syn}(\mathtt{B_i})$.

(4) If $p = A \rightarrow B^*$, recall $\mathit{Inh}(\mathtt{B}) \leftarrow Q(\mathit{Inh}(\mathtt{A}))$ and $\mathit{Syn}(\mathtt{A}) = \bigcup \mathit{Syn}(\mathtt{B})$. We first compute $\mathit{Inh}(\mathtt{B})$. If $\mathit{Inh}(\mathtt{B})$ is empty, then lv has no children and $\mathit{Syn}(\mathtt{A})$ is empty; otherwise m nodes tagged with B are created as the children of lv, such that each B node carries a tuple from the set $\mathit{Inh}(\mathtt{B})$ as its inherited attribute. The newly-created nodes are pushed onto the stack and evaluated in the same way. After all these nodes have been evaluated and popped off the stack, we compute $\mathit{Syn}(\mathtt{A})$ of lv by collecting the synthesized attributes of these nodes into a set.

(5) If $p = A \rightarrow \epsilon$, recall $\mathit{Syn}(\mathtt{A}) = g(\mathit{Inh}(\mathtt{A}))$. We simply compute $g(\vec{a})$ as the synthesized attribute of lv.

After $\mathit{Syn}(\mathtt{A})$ of lv is computed, we pop lv off the stack, and proceed to evaluate other nodes until no more nodes are in the stack. At the end of the evaluation, an XML tree is created, denoted by $\sigma(I, v)$.

The following should be remarked. First, each node is *visited* only twice: it is created and pushed onto the stack, and popped off the stack after its subtree is created and evaluated; it will not be on the stack afterward. Thus the evaluation takes *one-sweep*: it proceeds depth-first, following an order controlled by the dependency relations instead of left-right or right-left derivation. At each node, its inherited attribute is evaluated first, then its subtree, and finally, its synthesized attributes. Second, the evaluation is *data-driven*: the choice of a production in the nondeterministic case and the expansion of the XML tree in the recursive case are determined by queries on the underlying relational data. Third, *context-dependent* construction of XML trees is supported: synthesized attributes allow us to control the derivation of a subtree with data from other subtrees. Finally, each step of the evaluation expands the tree strictly following the DTD D. It is easy to show that if the AIG evaluation terminates, then it generates an XML tree that conforms to D. This yields a systematic method for DTD-directed integration.

Example 3.1: Consider again the AIG σ_0 of Fig. 2. Given databases DB_1, DB_2, DB_3, DB_4 described in Example 1.1, and a value v of of *Inh*(`report`).date, the evaluation of σ_0 generates an XML tree T of the form depicted in Fig 1 as follows. It first creates the root of T, tagged with `report`. Following the rules for the `report` production, it computes *Inh*(`patient`) by extracting data from DB_1 via query $Q_1(v)$, which treats v as a constant. For each tuple in the output of $Q_1(v)$, it creates a `patient` element as a child of `report`, carrying the tuple as the value of its inherited attribute. For each `patient` node *pt*, it creates `SSN`, `pname`, `treatments` and `bill` subelements. The first two have their `S` subelements carrying the corresponding PCDATA from the *Inh*(`Patient`) value v' of *pt*. Now the dependency relation of the `patient` production decides that the `bill` subtree cannot be constructed before the `treatments` subtree. Thus it first evaluates the `treatments` child of *pt* by computing $Q_2(v')$, which is a multi-source query over three databases: DB_1, DB_2 and DB_4. Again for each tuple in the output of $Q_2(v')$ it creates a `treatment` subelement. The subtree rooted at each `treatment` node is generated similarly, using the rules for the `treatment` and `procedure` productions. Observe that `treatment` is recursively defined; thus, its subtree is expanded until it reaches `treatment` nodes whose `procedures` do not consist of other treatments, i.e., when Q_3 returns the empty set over DB_4 at `procedure` nodes. That is, AIGs handle recursion in a data-driven fashion. At this point *Syn*(`procedure`), *Syn*(`treatment`) and *Syn*(`treatments`) are computed bottom-up by collecting `trId`s from their children's synthesized attributes. Observe that the value v_s of *Syn*(`treatments`) cannot be computed before the construction of the `treatments` subtree is completed since the subtree has an unbounded depth. Next, v_s is passed to the `bill` child of the patient node *pt*, and the evaluation proceeds to generate the `bill` subtree using the data in DB_3 and v_s as *Inh*(`bill`). The integration is completed when all the `patient` subtrees are generated. □

3.3 Constraint Compilation

An AIG is pre-processed so that (1) enforcement of its XML constraints is done in parallel with document generation, and (2) multi-source queries are rewritten into single-source ones to be executed directly by underlying relational engines. The output of this step produces a *specialized AIG*, an AIG that may have extra semantic rules and internal computation states. The generation of specialized AIGs is automatic: no user intervention is needed.

We now describe how to pre-process constraints in an AIG σ to get a specialized AIG σ'. The AIG σ' extends σ with additional synthesized attributes, semantic rules and a *guard* construct. The synthesized attributes may have a *bag* type (set with duplicates), along with bag union operators "$\uplus, \biguplus$" analogous to set operators "$\cup, \bigcup$". A guard captures a constraint with a boolean condition on these attributes: if the condition holds then the evaluation proceeds, otherwise it *aborts*, i.e., it is terminated without success. The preprocessing step, referred to as *constraint compilation*, does the following:

```
Semantic attributes:
  Syn(patient) = Syn(bill) = ...   =
        (bag(B),set(S1),set(S2), ...)
Semantic rules:
 patient → SSN, pname, treatments, bill
  Syn(patient).B = Syn(SSN).B ⊎ Syn(pname).B
                 ⊎ Syn(treatments).B ⊎ Syn(bill).B
  Syn(patient).S1 = Syn(SSN).S1 ∪ Syn(pname).S1
                 ∪ Syn(treatments).S1 ∪ Syn(bill).S1
  Syn(patient).S2 = ...

  guard:  unique(Syn(patient).B)
  guard:  subset(Syn(patient).S1, Syn(patient).S2)

 treatments → treatment*
  Inh(treatments).B  = ⨄ Syn(treatment).B
  Inh(treatments).S1 = ⋃ Syn(treatment).S1
  Inh(treatments).S2 = ...

 item → trId, tname
  Syn(item).B = Syn(trId).val
  Syn(item).S1 = { }
  Syn(item).S2 = Syn(trId).val
  .  .  .
```

Figure 3: Constraint compilation in σ'_0

(1) For each key $k : C(A.l \rightarrow A)$, create an additional member of type $bag(l_k)$ in *Syn*(`X`) for every element type `X`. Add semantic rules such that (i) *Syn*(`A`).l_k is given the value of the `l` subelement of A elements, (ii) for any `X` not equal to A, *Syn*(`X`).l_k collects *Syn*(`A`).l_k's below it in its subtree, excluding the l subelement value of `X`; and (iii) add a guard *unique*(*Syn*(`C`).l_k) for the context type C, which returns `true` iff *Syn*(`C`).l_k contains no duplicates, i.e., the values of all the l subelements of A elements are unique within each C subtree. This automatically generates code in σ' for checking the key.

(2) Inclusion constraint $C(B.l_B \subseteq A.l_A)$ is treated similarly: create two additional members l_a, l_b of set type in *Syn*(`X`) and add associated rules using set operators. For the type C, a guard *subset*(*Syn*(`C`).l_a, *Syn*(`C`).l_b) returns `true` iff *Syn*(`C`).l_a is contained in *Syn*(`C`).l_b, i.e., the inclusion relation holds within each C subtree.

The evaluation of the AIG is aborted iff any of these guards is evaluated to `false`, i.e., any constraint is violated.

For example, the constraints of the AIG σ_0 are compiled into synthesized attributes and rules shown in Fig. 3.

Several remarks are worth mentioning. First, the semantic rules associated with constraints can be simplified statically. For example, the rule for *Syn*(`patient`).B in Fig. 3 can be rewritten to *Syn*(`patient`).B = *Syn*(`bill`).B. These rules need not be necessarily evaluated bottom-up: the relation on attributes imposed by constraints can be used to optimize the evaluation. Indeed, if at some point during generation, a bag-valued attribute (key) is found to have duplicates, evaluation is aborted immediately. Similarly, an inclusion constraint $C(B.l_B \subseteq A.l_A)$ suggests that $B.l_B$ is passed to the rules for A such that one can check whether there is a value of $B.l_B$ that is not in $A.l_A$ during the computation of $A.l_A$, and vice versa. This avoids unnecessary computation. Constraint repairing [19] can be incorporated into the framework, but to simplify the discussion, we focus on constraint checking in this paper.

3.4 Multi-source Query Decomposition

We next show how to decompose multi-source SQL queries by introducing internal states into specialized AIGs.

```
Semantic attributes:
  $Inh$(St)   = (date, SSN, policy)
  $Inh$(St$_1$) = (set(trId, policy))
  $Inh$(St$_2$) = $Syn$(St$_1$) = (set(trId))
  $Syn$(St)   = $Syn$(St$_2$) = (set(trId, tname)), ...

Semantic rules:
 treatments → $St$, treatment*
  $Inh$(St) = $Q_2$($Inh$(treatments)),
  $Inh$(treatment).(trId,tname) ← $Syn$(St), ...
  where $Q_2(v)$:
   select i.trId, v.policy
   from   $DB_1$:visitInfo i
   where  i.SSN = $v$.SSN and i.date = $v$.date

 $St$ → $St_1$, $St_2$
  $Inh$(St$_1$) = $Q'_2$($Inh$(St)),     $Inh$(St$_2$) = $Q''_2$($Syn$(St$_1$))
  $Syn$(St) = $Syn$(St$_2$)
  where $Q'_2(v_1)$:
   select c.trId
   from   $DB_2$:cover c, $v_1$ T1
   where  c.trId = T1.trId and c.policy = T1.policy
  where $Q''_2(v_2)$:
   select t.trId, t.tname
   from   $DB_4$:treatment t, $v_2$ T2
   where  t.trId = T2.trId

 $St_1$ → $\epsilon$
  $Syn$(St$_1$) = $Inh$(St$_1$)      /* similarly for $St_2$ */
  .  .  .
```

Figure 4: Multi-source query decomposition in σ'_0

Recall that the query Q_2 in the AIG σ_0 is defined on multiple databases: DB_1, DB_2 and DB_3, which may have different systems and may even reside in different sites. It is desirable to shift work from the middleware to the source relational engine as much as possible; thus one wants to rewrite the query into sub-queries such that each of them can be sent to and executed at the underlying relational system. For this reason a specialized AIG supports a set of *states*, ST, to supplement the set Ele in the DTD. A state in ST behaves like a type in Ele: it has associated attributes and semantic rules, which are evaluated in the same way. The difference is that after the integration process terminates the nodes labeled with states in ST are removed from the resulting XML tree T. That is, these ST nodes only serve for computation purpose. As an example, Fig. 4 shows how the specialized AIG σ'_0 rewrites Q_2 by means of internal states St, St_1 and St_2.

A specialized AIG can be viewed as a two-way tree automaton [11] that can issue queries. It should be mentioned that multi-source query decomposition is conducted automatically. A left-deep query plan tree is first generated by means of an underlying relational optimizer, under the assumption that all the data lie on the same source. Internal states are introduced to represent the output of each node in the plan tree. The parent-child relationship follows the tree structure of the plan, the inherited attribute and semantic rule for a given state correspond to the output attributes and operator of the associated node in the query plan.

4. Static analyses of AIGs

An advantage of using a limited specification language is the ability to infer powerful static guarantees, and perform advanced program analyses. In contrast, expressive languages such as XQuery and XSLT can not allow such analyses: indeed XQuery and XSLT are Turing-complete, and hence termination issues and output guarantees are out of reach. Since a key goal of integrating data towards a target XML DTD is to ensure conformance of the output to a standard interface, static correctness assurance should be a central design goal.

The following results give a sample of the correctness checks that can be done statically on AIGs:

- Given an AIG σ without constraints and defined with conjunctive queries, one can decide whether σ will necessarily terminate on all instances.
- Given an AIG σ as above, one can decide whether σ will terminate on some instances.
- Given an AIG σ as above and an element type E in the DTD of σ, one can decide whether E can be reached, and whether E must be reached on any instance.

All of the above are proved by symbolic execution of σ, tracking along each path the conjunctive queries that are guaranteed to hold down that path. One can show that even in the case of recursive DTDs, one need only simulate execution down to a fixed depth to detect non-termination.

Of course, some limits of static analysis follow from the corresponding limits on relational queries:

- The above problems are undecidable for AIGs defined with arbitrary SQL queries, even when the underlying DTD is non-recursive.
- It is undecidable whether an AIG with inclusion and key constraints necessarily terminates (on some instance) even when the underlying DTD is non-recursive and the queries involved are conjunctive.

The first result follows directly from the undecidability of equivalence problem for SQL, while the second result makes use of undecidability results for the consistency of DTDs and (relative unary) XML constraints [1].

The restricted form of AIGs also allows certain kinds of optimizations to be done easily at compile time. An example is *copy elimination*. A semantic rule in a (specialized) AIG is classified as a *copy rule* (CSR) if its righthand side makes use only of functions of the form x_k or $\bigcup x$; it is referred to as a *query rule* (QSR) otherwise. For example, the semantic rule "Inh(treatments).trIdS = $\bigcup Syn$(treatment).trIdS" is a CSR. A *copy chain* is a maximal sequence of dependent CSRs $A_1 = f_1(A_0), \ldots, A_n = f_n(A_{n-1})$ followed by a QSR Q that references fields of A_n. In copy elimination, we replace any reference to fields of A_n in Q by the corresponding attributes of A_0. Note that in the resulting AIG, the semantic rule for the attributes of an element type E may now have parameters that are far away from E in the DTD. Copy-elimination is a kind of inlining – we will see a more general version of this in the merging algorithm of Section 5. It reduces dependencies among queries and thus allows more queries on different data sources to be executed parallelly.

5. Evaluating AIGs

In this section, we present a mediator system for evaluating AIGs whose goal is to generate the DTD-conforming XML document as fast as possible. Compared to traditional distributed query processing [18], the optimization and evaluation of AIGs present additional challenges. In AIGs, evaluation involves a *set* of queries that are related by *inter-query dependencies* (see the definition of query dependency graph below). Furthermore, since the output of

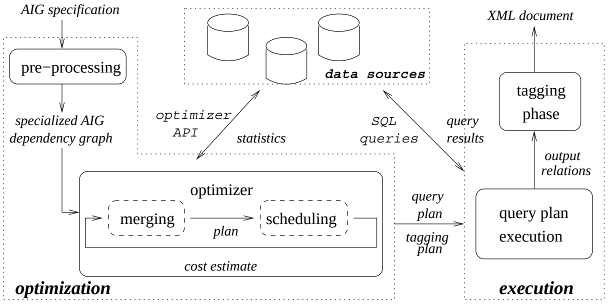


Figure 5: AIG Middleware Architecture

these queries is used to drive the generation of the XML document, as queries are transformed during optimization, this relationship must be dynamically maintained. The task to manipulate and combine queries is thus heavily constrained by both inter-query and query-to-data-source dependencies, leading to a novel optimization problem.

5.1 Overview

The architecture of the middleware system is shown in Fig. 5. The system takes an AIG specification σ as input and evaluates it in four phases to produce an XML document that conforms to the DTD and satisfies the constraints embedded in σ. In the *pre-processing* phase, it rewrites σ into a specialized AIG with single-source parameterized queries and relations encoding the constraints (see Sections 3.3 and 3.4); and performs copy elimination. The *optimization* phase produces a customized application, consisting of 1) a set of non-parameterized queries for each source, along with their input and output schemas; 2) a *query plan* giving an ordering for the queries among the various data sources; and 3) a tagging plan for generating the final document tree. The *execution phase* performs the runtime evaluation of the AIG. The query plan is executed to produce a set of *output relations* – a relational representation of the XML document. More specifically, at each source, the unprocessed query that is lowest in the plan's ordering is selected for execution as soon as its inputs are available. When the query is completely evaluated, its results are then shipped (via the mediator) to every dependent site. Finally, in the *tagging phase*, the tagging plan is applied to these relations to produce the final output document. Note that while query plan evaluation involves both processing on the sources and shipping of data over the network, tagging is done completely within the middleware.

To simplify the discussion, in the bulk of this section we will focus on the case of non-recursive AIGs. As an example, we use a variation of the AIG σ_0 of Fig. 2, by unfolding the recursion only once and assuming that the `procedure` leaf has no children. For non-recursive AIGs, the first three phases can be done entirely at compile time. We discuss how our proposed techniques can be extended to handle recursive AIGs in Section 5.5.

The pre-processing of the AIG yields a specialized AIG σ_0' by decomposing multi-source queries, as illustrated in Fig 4 (to simplify the discussion we ignore the rules for constraints). The graph representation of σ_0' after copy elimination is shown in Fig. 6. In this graph, edge labels represent queries for computing inherited attributes (*e.g., Q2, Q2'*); node labels represent queries for computing synthesized attributes (*e.g.,* ST, ST_1); and the dashed edges indicate the

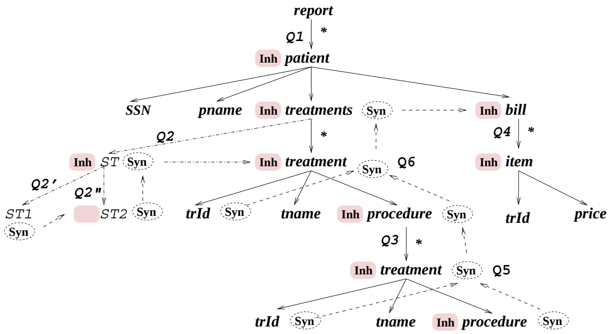


Figure 6: Example: a specialized AIG

flow of information. The results of the queries that compute inherited attributes are shipped to the mediator (viewed as a special data source `Mediator`), cached in temporary tables, and used to construct the final XML document in the tagging phase. The synthesized attributes are computed at the mediator using the cached tables. Observe that copy elimination removes the queries and attributes that are defined via copy rules (with simple projections) and are not referenced by other attributes. In the tagging phase, the values of these attributes are simply extracted from the cached tables when they are needed.

To construct the final XML tree, the queries in an AIG need to be rewritten such that the output relation of each query contains information that can uniquely identify the position of a node in the XML tree, *i.e.,* a coding of the path from the root to the node. Furthermore, as opposed to the conceptual evaluation strategy, for each query Q that takes a single tuple of the output *tmp* of another query Q' as an input, we convert Q to one that takes the entire set *tmp* of tuples as an input. This is done by replacing the tuple parameter in Q with the temporary table that contains the output of Q'. In our example, the parameterized query $Q_2(v)$ in Fig. 4, where v is a single tuple in Inh(`patient`) (the result of the query Q_1):

```
select  i.trId, v.policy
from    DB1:visitInfo i
where   i.SSN = v.SSN and i.date = v.date
```

is transformed into $Q_2(T_{\text{patient}})$:

```
select  i.trId, v.policy, v.SSN
from    DB1:visitInfo i, Tpatient:v
where   i.SSN = v.SSN and i.date = v.date
```

where T_{patient} is a temporary table storing Inh(`patient`). Observe that $Q_2(T_{\text{patient}})$ is executed once when T_{patient} is available, instead of being executed for each tuple in T_{patient}. Note that the output of $Q_2(T_{\text{patient}})$ includes an extra field `SSN`, which encodes a path from the root to a `patient` node; this, together with the `trId` field, uniquely identifies the position of the `ST` nodes (in fact, `treatment` nodes) in the final document. Given this, the final XML document can be generated by simply sort-merging the cached temporary tables. This query rewriting process is done in the optimization phase via an iterative process in which the scalar parameters of query Q are replaced by temporary tables, one per output of each query that Q depends upon. The optimization phase also generates the tagging plan, which will produce the tree in a top-down fashion, associating each node of element type E with a key path in the table for Inh(`E`) — internal states (e.g., ST, ST_1, ST_2) in the specialized AIG are eliminated in the tagging phase.

To capture the producer-consumer dependencies among the queries generated by the process above, we define the *query dependency graph* G of the AIG. The graph G contains a node for each query, and a directed edge from a node Q_1 to node Q_2 iff the result of Q_1 is used in Q_2, denoted by $Q_1 \rightarrow_G Q_2$. The query dependency graph for the AIG of Fig. 6 is shown in Fig. 7(a). In this graph, each query is associated with the data source where it is evaluated. Note that G is a DAG (directed acyclic graph): the DAG structure reflects the fact that an AIG generally specifies sharing of a query output among multiple further queries. This is in sharp contrast both to traditional distributed query processing and to XML publishing formalisms such as [15, 3].

We next consider query plan generation in the optimization phase. Different factors contribute to the total execution time of an AIG specification, most notably: *communication costs* – data must be transferred between data sources as well as between the mediator and data sources; *query execution overheads* – in addition to the cost of sending queries to data sources (*i.e.,* opening a connection, parsing and preparing the statement, etc.), temporary tables may have to be created and populated with inputs to a query; *query execution costs* – the total execution time of a query in a given data source; *parallel execution* – queries in different sources may be executed in parallel. Consequently, the main goal of the AIG optimizer is to minimize these costs and overheads by exploiting inter-query parallelism. In the remainder of this section, we discuss optimization in detail.

The AIG optimizer reduces the number of queries issued to data sources by *merging* queries that are processed at the same source into a single, larger query. Query merging can help decrease the communication costs between the mediator and data sources, while also potentially diminishing query processing time and execution overheads. However, query merging may involve outer-joins or outer-unions which increase the arity of the query's result table and hence the output size. Furthermore, injudicious query merging may increase processing time and lead to unnecessary delay of further query executions. Section 5.4 outlines a cost-based query merging algorithm that iteratively applies a greedy heuristic to select pairs of queries to merge, where the cost function estimates the best query schedule (*i.e.,* with the smallest total execution time) for the merged query graph. The interaction of scheduling and merging is one of the most delicate points of the optimization algorithm. The problem of scheduling queries in the presence of DAG-like query dependencies, such as those considered here, is NP-hard, and thus an approximate scheduling algorithm is an essential component of our solution.

The various steps of the optimization algorithm are illustrated in Fig. 7. The optimization algorithm proceeds as follows. Given the AIG query dependency graph G, invoke the algorithm `Merge` (see Section 5.4) to merge nodes (queries) in G that lead to reduction in query evaluation cost. This involves a function for estimating the cost of a query plan (see Section 5.2), and an algorithm, `Schedule` (see Section 5.3) for computing a *good* query execution ordering for G – an approximation to the optimal schedule for G. At the end of this merging phase, the final schedule is generated and submitted to the runtime engine of the middleware for use in the execution phase. Note that, while the AIG remains fixed during optimization, the query dependency graph G is updated in each stage of the iterative optimization.

The query dependency graph and query plan that might be generated for the example are shown in Figs. 7(a) and 7(b). At runtime, the middleware issues queries to each data source according to the query plan, and ships output tables produced by queries to sources that are dependent on these results as soon as they become available. At the end of this process, the middleware applies the tagging plan to produce the final XML tree.

5.2 Cost Evaluation

In order to estimate the cost of an AIG, information is needed about the cost of executing individual queries, shipping data, and the degree of inter-query parallelism. In our current implementation, we assume that data sources provide a query costing API, *i.e.,* for a given a query Q to be executed on a data source S, S provides estimates for both the processing time of evaluating Q (in seconds), denoted by $\mathtt{eval\_cost(Q)}$, as well as the output size (number of tuples and tuple width in bytes) of Q, denoted by $\mathtt{size(Q)}$. In particular, if Q references the results of another query Q', the API is able to accept cost estimates of Q' (e.g., cardinality information) as inputs in the computation of $\mathtt{eval\_cost(Q)}$.

The mediator also maintains information about the costs for communicating with the various data sources. This information is used to estimate the communication time (in seconds) to ship results to/from data sources. We denote by $\mathtt{trans\_cost(S_1, S_2, B)}$ the cost of transferring data of size B bytes from source S_1 to S_2. Note that if neither S_1 nor S_2 refers to the mediator, then the data is shipped from S_1 to S_2 via the mediator; and if S_1 and S_2 are the same source, $\mathtt{trans\_cost(S_1, S_2, B)} = 0$.

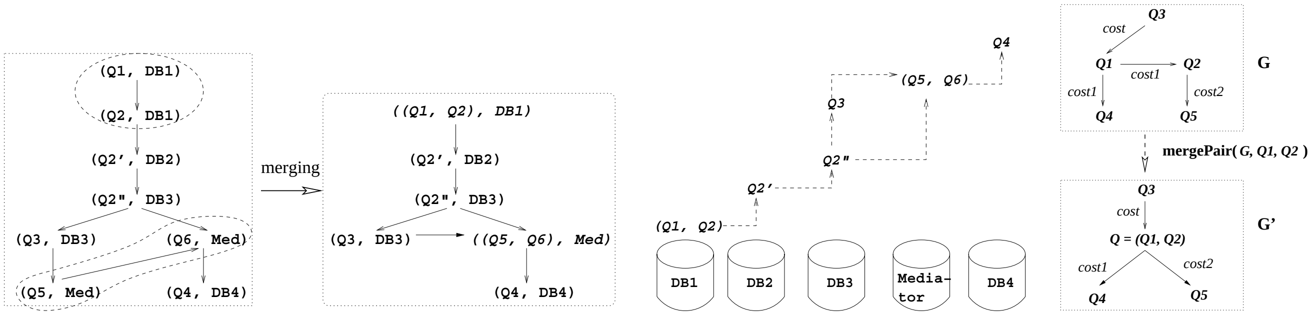
Given the evaluation costs of each individual query, and the cost of shipping data across data sources, the total execution time can be derived from the execution plan produced by the scheduling algorithm (see Section 5.3). An execution plan P contains a schedule for each data source S_i which consists of a sequence of queries $\pi_i = < Q_{i,1}, Q_{i,2}, \cdots, Q_{i,m} >$ to be executed (in the given order) at S_i. Note that the schedules in P have to be consistent with the partial ordering in the query dependency graph G; i.e., if $Q_{i,k}$ is reachable from $Q_{i,j}$ in G, then $j < k$. Let $\mathit{comp\_time}(Q_{i,j})$ denote the completion time of the evaluation of query $Q_{i,j}$ on data source S_i. Then the response time of evaluating the execution plan P, denoted by $\mathit{cost}(P)$, is simply the maximum of $\mathit{comp\_time}(Q_{i,j})$ over all the queries $Q_{i,j}$ in P. The completion time of a query Q at source S_i can be computed recursively as follows:

$$\begin{aligned} &\mathit{comp\_time}(Q_{i,m}) = \mathtt{eval\_cost(Q_{i,m})} + \mathit{max}\{T\}; \\ &\quad \text{where } T = \{\mathit{comp\_time}(Q_{i,m-1})\} \cup \{\mathit{comp\_time}(Q_{j,n}) + \\ &\qquad\qquad \mathtt{trans\_cost(S_j, S_i, size(Q_{j,n}))} \mid Q_{j,n} \rightarrow_G Q_{i,m}\} \end{aligned}$$

Thus $\mathit{cost}(P) = \max\{\mathit{comp\_time}(Q_{i,j}) \mid Q_{i,j} \in \pi_i, \pi_i \in P\}$, and can be computed in at most quadratic time using dynamic programming.

5.3 Scheduling Algorithm

Given a query dependency graph G, the goal of scheduling is to produce an execution plan P that is consistent with G and that minimizes the response time. Unfortunately, for any reasonable cost function on query plans, including the cost function given in Section 5.2, the problem of finding the optimal execution plan is NP-hard, even when there is only one data source (by reduction from the problem for sequenc-



(a) Query dependency graph (b) Execution plan (c) Query merging

Figure 7: Dependency graph, query merging and scheduling

```
Algorithm Schedule($G$)

1. Let $L$ denote the sequence of queries in $G$ sorted
   in reverse topological order;
2. for each $Q \in L$ do
3.     $\ell evel(Q) = 0$;
4.     for each $Q \rightarrow_G Q'$ do
5.         $\ell evel(Q)$ =max{trans_cost(S, S', size(Q)) + $\ell evel(Q')$,
                      $\ell evel(Q)$},   where $Q$ and $Q'$ are
                          evaluated at $S$ and $S'$, resp.;
6.     $\ell evel(Q) = \ell evel(Q) +$ eval_cost(Q);
7. for each data source $S_i$ do
8.     Let $\pi_i$ be the set of queries in $G$ evaluated at $S_i$;
9.     Sort $\pi_i$ such that $Q$ precedes $Q'$ in $\pi_i$ if
       $\ell evel(Q) > \ell evel(Q')$;
10. return execution plan $P = \{\pi_i \mid S_i$ is a source$\}$;
```

Figure 8: Algorithm Schedule

```
Algorithm Merge($G$)

$P$ := Schedule($G$);    cost := $cost(P)$;
repeat
  benefit := false;    $G_{new} = G$;
  for each $Q_1, Q_2 \in G$ scheduled for the same source do
      $G'$ := mergePair($G, Q_1, Q_2$);
      if ($G'$ is acyclic)
      then $P'$ := Schedule($G'$);    c := $cost(P')$;
          if c < cost
          then benefit := true;  cost := c;  $G_{new} := G'$;
  $G := G_{new}$;
until (benefit = false);
return $G$;
```

Figure 9: Algorithm Merge

ing to minimize completion time [16]). We thus present a heuristic that gives an approximate solution.

Similarly to the list-scheduling heuristic [25], we assign a priority value to each query in G to reflect its "criticality", and then sort the queries based on their priority values. The basic idea is to optimize the critical paths of G, i.e., the sequences of queries that affect the overall completion time. For a query Q on data source S, we define a *path cost* of Q to be the cost for evaluating all the queries along a path from Q to a leaf query in the dependency graph G, and we adopt the maximum path cost as the priority value of Q, denoted by $\ell evel(Q)$. Then, $\ell evel(Q)$ can be computed recursively:

$$\begin{aligned}\ell evel(Q) = \texttt{eval\_cost(Q)} + \max\{&\ell evel(Q') + \\ &\texttt{trans\_cost(S,S',size(Q))} \mid Q \rightarrow_G Q'\}\end{aligned}$$

where S and S' denote, respectively, the sources where Q and Q' are evaluated. Intuitively, $Q_{i,j}$ is given a higher priority than $Q_{i,k}$ if $\ell evel(Q_{i,j}) > \ell evel(Q_{i,k})$, i.e., if the evaluation of the queries dependent on $Q_{i,j}$ takes longer time than those depending on $Q_{i,k}$. This motivates us to order $Q_{i,j}$ before $Q_{i,k}$ on source S_i.

Putting these together, we have Algorithm Schedule in Fig. 8. First, steps 1 to 6 compute $\ell evel(Q)$ for each query Q, and then steps 7 to 9 create a schedule for each data source by sorting the queries to be evaluated at the source based on $\ell evel(Q)$. It is easy to verify that the algorithm takes at most quadratic time.

5.4 Merging Algorithm

We next present a merging algorithm that selects certain queries at the same source based on cost estimates, and combines them into a single query. Query merging can reduce the data communication overhead, and may also lead to a more efficient query plan, since more optimization opportunities are offered for the data source's query optimizer. However, injudicious merging could lead to less execution parallelism since merging queries also result in their data dependencies being "merged". Thus, query merging has to be optimized jointly with query scheduling.

Algorithm Merge takes a query dependency graph G as input and iteratively derives a new query dependency graph G' from G by determining the "best" pair of queries in G to merge at each iteration. Two queries can be merged only if they are executed at the same data source and the resultant new query dependency graph from their merging remains acyclic. The cost of the execution plan for a query dependency graph G is computed using Schedule(G) as discussed in the previous section. The iterative query merging process continues until no pair of queries that can be merged would lead to reduction of the execution cost.

The function mergePair derives a new dependency graph G' from an input dependency graph G by merging two of its queries Q_1 and Q_2 into a single query Q'. More specifically, G' is constructed from G as follows: for each query Q, if $Q \rightarrow_G Q_1$ or $Q \rightarrow_G Q_2$, we add an edge $Q \rightarrow_G Q'$; similarly, if there is an edge $Q_1 \rightarrow_G Q$ or $Q_2 \rightarrow_G Q$, we add an edge $Q' \rightarrow_G Q$. Finally, we remove Q_1, Q_2 and all their edges from G; the resulting graph is G', as depicted in Fig. 7(c).

We next describe how a new query Q is generated from the merging of two queries Q_1 and Q_2. For the simple case where there are no data dependencies between Q_1 and Q_2, Q is simply the outer-union of Q_1 and Q_2. We include an extra "tagging" column in the output relation of Q to identify whether the tuples are from Q_1 or Q_2 so as to facilitate the extraction of the relevant tuples from Q at later stages. For the case where the queries Q_1 and Q_2 are related by data dependencies, e.g., $Q_1 \rightarrow_G Q_2$, they are merged by inlining Q_1 in Q_2 , combining the common key paths, as in the outer join approach of [15, 3].

The extraction and shipping of relevant tuples is important for reducing the communication costs. Consider the example dependency graph in Fig. 7(c), which shows the merging of two queries Q_1 and Q_2 into a new query Q. Each directed edge in the figure is labeled with the communication cost of shipping the output of one query to be used as input for another query. Note that since Q_4 needs only the output of Q_1 (similarly, Q_5 needs only the output of Q_2), the relevant tuples from Q are extracted before shipping them to the target query; consequently, the communication costs in G and G' remain the same.

One can verify that Algorithm Merge takes at most $O(n^5)$ time where n is the size of the dependency graph G. This is also the overhead of entire optimization phase, including the costs of the function *cost* and Algorithms Schedule, Merge.

5.5 Discussion

We discuss here several extensions of the evaluation algorithm described previously.

We first describe the modifications necessary for dealing with recursion. In the presence of recursion in the DTD embedded in an AIG σ, the graph representation of σ becomes cyclic. We begin with a user-supplied estimate d of the maximum depth of the output tree, and calculate from it a (partial) AIG by iteratively unfolding the recursive rules until all nodes that are still expandable are of depth above d. Optimization is performed at compile time on the resulting AIG σ' as before, with the caveat that certain nodes in the dependency graph may have queries with inputs depending upon further expansion. If at runtime all queries can be evaluated, the computation of σ' terminates and the final output relations of σ can be produced. If there are nodes Q in the dependency graph that are blocked waiting for tables that require further processing for their materialization, then the recursion is unrolled again starting from the element type in σ' corresponding to Q: this process continues until all inputs are available, with the value of d being updated to reflect the new depth information. It is worth mentioning that the depth of recursive evaluation of XML documents found in practice is generally fairly small [8]; hence a conservative estimate of the recursion depth will yield a non-recursive DTD equivalent to the original in most cases. This allows us to exploit the cost-based estimation used in the non-recursive case, while avoiding as much as possible the need to iterate the process at runtime.

We now describe several additional features of the middleware that can lead to significant performance improvements. The algorithm described here made use of static query schedules for simplicity – significant efficiency gains can accrue from using *dynamic scheduling*, in which a runtime scheduler updates the query plans for each site in parallel with evaluation, taking advantage of knowledge about workload on data sources. Additional gains come from enhancing the query-processing power of the middleware engine. For our prototype, the middleware does not possess a relational engine: any middleware processing is performed within application code. A simple extension would be to provide a relational query-processor on the middleware, whose optimizer can then be used to estimate the cost of middleware processing.

	small	medium	large
patient	2500	3300	5000
visitInfo	11371	14887	22496
cover	2224	3762	8996
billing	175	250	350
treatment	175	250	350
procedure	441	718	923

Table 1: Cardinalities of tables for different datasets

Related work. Evaluation algorithms for middleware systems has been explored in relational integration systems such as Garlic [17]. Garlic allows cost-based optimization and processing of relational queries over multiple sources, dealing with issues of scheduling and parallelization as we do, while allowing for sources with limited query capabilities. In the XML context the relational queries to be optimized can be modified at the cost of post-processing to reconstruct the final output tree. The specification of complex interdependences in AIGs leads to the problem of optimizing a set of queries with DAG-like dependencies, as opposed to a single query tree in systems such as Garlic. Recent XML publishing middleware such as [5, 15, 26] give evaluation algorithms specifically geared to relational-to-XML mappings. The formalisms there feature single data sources, and templates that correspond to fixed-depth top-down evaluation.

6. Experimental Results

In order to verify the effectiveness of our query scheduling and query merging algorithms, we conducted a preliminary experimental evaluation using the AIG described in Example 1.1. The relational datasets were constructed in two steps: first, we used the ToXgene data generator[1] to produce XML data that conforms to a canonical relational DTD; we then used a simple parser that reads the XML data and generates a comma-separated file (which can be bulk-loaded into the RDBMS). We generated datasets by varying the sizes of the different relations (see Table 1). Further variants were obtained by unfolding the recursive grammar rule "*procedure* → *treatment**" multiple times to obtain non-recursive DTDs with between 2 to 7 levels of **trId** elements. Note that the main effect of unfolding is the increase in the size of intermediate results. For example, for the *Large* data set, the cardinality of a 3-way self join of the procedure table is 4055, whereas the cardinality of a 4-way self join is 6837. The query plans were run on DB2 v8.1 for Linux. The total evaluation time was computed by simulating the transfer of temporary tables among the distributed data sources, *i.e.,* the mediator and different databases, using different bandwidths.

Fig. 10 demonstrates the impact of query merging on the performance of AIG evaluation for datasets of different sizes and unfolding levels. The evaluation time includes both the query evaluation and communication costs (for a bandwidth of 1Mbps among data sources). Each entry in Fig. 10 shows the ratio of the evaluation time without query merging to that with query merging. The results clearly demonstrate a significant gain in performance (up to a factor of 2.2) when query merging is enabled. As expected, the performance improvement increases with larger datasets and more complex

[1] http://www.cs.toronto.edu/tox/toxgene

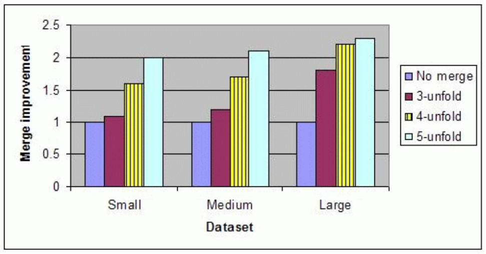


Figure 10: Improvement due to query merging

DTDs.

7. Conclusion

We have proposed a new approach for integrating relational data from different sources into an XML document under both syntactic and semantic constraints. This is based on a novel notion of AIGs. AIGs not only yield a uniform framework for ensuring both DTD-conformance and XML-constraint satisfaction in data integration, but also provide the ability to support data-driven and context-dependent generation of XML documents. We have also implemented a prototype of a middleware system for evaluating AIGs efficiently. The system explores new optimization techniques for merging and scheduling queries over multiple sources in the presences of DAG-like inter-query dependencies; our preliminary experimental results show that these techniques yield substantial reductions in processing time.

Several extensions to the optimization framework are targeted for future work. We intend to incorporate dynamic scheduling algorithms, and to make use of selectivity estimates within our cost function. In the optimization process, we recognize the need to take into account both restrictions on the capabilities of sources (as in [17]) and limitations on processing power and cache memory size on the mediator. We are also investigating methods for statically generating query plans for AIGs based on recursive DTDs, utilizing statistics on the depth of chains within source relations. Finally, we are studying extensions of AIGs which yield mappings that are guaranteed to be information-preserving.

Acknowledgments: Wenfei Fan is supported in part by NSF Career Award IIS-0093168 and NSFC 60228006. We thank Denilson Barbosa for helping us generate the datasets used in the experiments.

8. References

[1] M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *PODS*, 2002.

[2] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. In *SIGMOD*, 2000.

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

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, Feb. 1998. `http://www.w3.org/TR/REC-xml/`.

[5] 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.

[6] R. G. Cattell. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

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

[8] B. Choi. What are real DTDs like. In *WebDB*, 2002.

[9] J. Clark. XSL Transformations (XSLT). W3C Recommendation, Nov. 1999. `http://www.w3.org/TR/xslt`.

[10] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *SIGMOD*, 1998.

[11] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Ligiez, S. Tison, and M. Tommasa. *Tree Automata Techniques and Applications*. Online, 1999.

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

[13] P. Deransart, M. Jourdan, and B. Lorho (eds). Attribute Grammars. *LNCS 323*, 1988.

[14] R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Semantics and query answering. In *ICDT*, 2003.

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

[16] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[17] L. M. Haas, D. Kossmann, E. L.Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, 1997.

[18] D. Kossman. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

[19] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[20] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, 1998.

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

[22] Y. Papakonstantinou and V. Vianu. Type inference for views of semistructured data. In *PODS*, 2000.

[23] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating web data. In *VLDB*, 2002.

[24] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[25] V. Rayward-Smith, F. Burton, and G.J.Janacek. Scheduling Parallel Programs Assuming Preallocation. In P. Chretienne, E. Coffman, J. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*, pages 145–165. John Wiley & Sons, 1995.

[26] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.

[27] 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.

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