Composable XML Integration Grammars

Wenfei Fan* Minos Garofalakis Ming Xiong Xibei Jia †
Bell Laboratories, Lucent Technologies University of Edinburgh
{wenfei,minos,xiong}@research.bell-labs.com x.jia@sms.ed.ac.uk

ABSTRACT

The proliferation of XML as a standard for data representation and exchange in diverse, next-generation Web applications has created an emphatic need for effective XML data-integration tools. For several real-life scenarios, such XML data integration needs to be *DTD-directed* – in other words, the target, integrated XML database must conform to a prespecified, user- or application-defined DTD. In this paper, we propose a novel formalism, *XML Integration Grammars (XIGs)*, for specifying DTD-directed integration of XML data. Abstractly, an XIG maps data from multiple XML sources to a target XML document that conforms to a predefined DTD. An XIG extracts source XML data via queries expressed in a fragment of XQuery, and controls target document generation with tree-valued attributes and the target DTD. The novelty of XIGs consists in not only their automatic support for DTD-conformance but also in their *composability*: an XIG may embed local and remote XIGs in its definition, and invoke these XIGs during its evaluation. This yields an important modularity property for our XIGs that allows one to divide a complex integration task into manageable sub-tasks and conquer each of them separately. To efficiently evaluate XIGs we provide algorithms for merging XML queries in an XIG and for scheduling queries and embedded XIGs. These lead to an effective framework, as well as a design tool for XQuery, for effectively specifying and computing complex, DTD-directed XML integration.

Categories and Subject Descriptors: H.2.4 [Database Management]: Distributed Databases

General Terms: Algorithms, Design, Experimentation.

Keywords: XML, Data Integration, Grammar

1. INTRODUCTION

XML [9] is rapidly emerging as the dominant standard for data representation and exchange on the Web. The ubiquity of XML, in conjunction with the diversity of next-generation Web applications that rely on it as a data-exchange format, clearly highlights the need for effective XML integration tools, *i.e.*, tools that can efficiently collect data from multiple distributed XML sources and incorporate it in a target XML document. In practice, such XML integration is typically *DTD-directed* – that is, the integration task is constrained by a predefined DTD that the target XML document is required to conform to. The need for DTD-conformance is evident in real-life data exchange: enterprises agree on a common DTD and then exchange and interpret their XML data based on this predefined DTD. Another important application of DTD-conformance concerns security: the integrated XML document, as a view of the original data, is required to conform to a prespecified view DTD in order to both hide confidential information and facilitate effective formulation of user queries over the secure integrated view.

Example 1.1: Consider the XML-to-XML transformation of promotional data for a car sale. The source data is specified by the DTD D^s_{sale} depicted in Fig. 1(a), in which '$*$' indicates one or more occurrences. It consists of `cars` promoted and their `features`. Each `feature` is identified by a `fid`, a key of the feature, and may be composed of other features. To exchange the data, one wants to convert the source data to a target document conforming to the DTD D_{sale} given in Fig. 1(c) (we omit the definition of elements of PCDATA type). The target DTD groups `features` under each car for `sale`, along with the composition hierarchy of each `feature`. Observe that the target DTD is recursive: the element type `features` is indirectly defined in terms of itself.

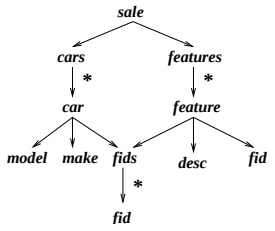
As another example, consider a view for car dealers. Each dealer maintains a local XML document specified by a source DTD D^s_{dealer}, which describes the dealer, cars carried by the dealer, and invoice, as depicted in Fig. 1(b). Some information is confidential, such as `invoice` and `quantity`, as indicated by the shadowed nodes in Fig. 1(b), which should not be made public. To hide the confidential data, one wants to define a view for each dealer such that the dealer data can only be accessed through the view. As a user interface the dealers want to provide the view DTD D_{dealer} given in Fig. 1(c) and requires the views to conform to D_{dealer}. Here the `inStock` status of a `car` is `yes` if its `quantity` in the original document is no less than 1; this disjunction in the target DTD leads to a non-deterministic structure. □

Ensuring the conformance of an integrated XML document (created through multiple XML data sources) to a predefined target DTD is a non-trivial problem. First, note that the target DTD itself may specify a fairly complex schema structure, e.g., recursive and/or non-deterministic with disjunctions. Second, the integration task may be large-scale and naturally "hierarchical" – in other words, the integration may involve a large number of distributed data sources, where some of the sources are virtual, in the sense that they are views that need to be created via XML integration. This latter requirement suggests that effective XML-integration specifications should be *composable*, such that large, complex integration tasks can be built via composition of simpler sub-tasks. This is along the same lines as modularity in programming-language principles – the key idea is to divide a complex task into manageable sub-tasks and conquer each sub-task separately.

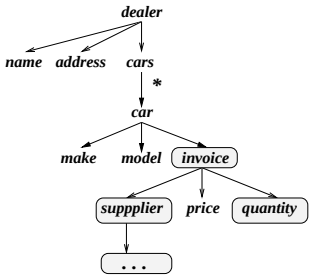
Example 1.2: Let us consider integration of XML data for car dealers in a region together with sale promotion data. The regional integration is to extract data from XML sources and construct a single target document that consists of `sale` data, information of all the `dealers` in the region, and `cars` carried by these

*Also affiliated with the University of Edinburgh. Supported in part by NSFC 60228006, EPSRC GR/S63205/01 and GR/T27433/01.
†Supported by EPSRC GR/S63205/01.

CIKM'04, November 8–13, 2004, Washington, DC, USA.



(a) DTD graph for D^s_{sale}



(b) DTD graph for D^s_{dealer}

D_{sale}:

```
<!ELEMENT promotion (sale*)
<!ELEMENT sale (make, model, features)
<!ELEMENT features (feature*)
<!ELEMENT feature (desc, features)
```

D_{dealer}:

```
<!ELEMENT dealer (name, address, cars)>
<!ELEMENT cars (car*)
<!ELEMENT car (make, model, price, inStock)
<!ELEMENT inStock (yes | no)
```

(c) Target DTDs

Figure 1: Example: Car sale and car dealers

dealers and promoted by `sale`. As shown in Fig. 2(a), the XML sources include (1) a `sale` document conforming to DTD D^s_{sale}, and (2) `dealer` views conforming to D_{dealer}, as described in Example 1.1. The target document is required to conform to the DTD D given in Fig. 2(b). Specifically, the integration is to transform the `sale` source data to D_{sale}, and collect `dealer` information from the views; for each dealer, it only gathers data for `cars` that are promoted by `sale`.

This integration task is rather complex. First, the target DTD is recursive and non-deterministic; its DTD graph (Fig. 2(c)) is cyclic and contains dashed edges (we use dashed edges to denote disjunction to distinguish from solid edges for concatenation). Second, the integration is "hierarchical": it involves a number of XML views distributed across the dealers' sites, which are in turn the result of transformation from local documents conforming to D^s_{dealer}. These views serve not only as data sources for the regional integration, but also as independent user interfaces for the dealers. Third, there is dependency on different parts of the target document: the generation of `cars` under `dealers` depends on `promotion`. Putting these in a single integration specification makes it hard to design, read and verify the correctness of the specification. □

Why not Use XQuery or XSLT? Obviously, a straightforward solution to DTD-directed XML data integration would be to employ some well-known XML query language (*e.g.*, XQuery [11], XSLT [12]) to define an integrated XML view, and then check whether the resulting view conforms to the prescribed DTD. Unfortunately, such an obvious approach quickly runs into a number of technical difficulties. First and foremost, using full XML query languages to define an integrated view *cannot guarantee DTD-conformance*. Specifically, type inference for such derived XML views is too expensive to be used in practice: it is intractable for extremely restricted view definitions, and undecidable for realistic views [2]. Similarly, accurate XML type checking is a hard problem – thus, languages such as XQuery typically implement only approximate type checking. Worse still, such an approach provides no guidance whatsoever on how to specify a DTD-conforming XML view. This means that DTD-directed integration becomes a trial-and-error process where, if a resulting view fails to type-check, the view definition needs to be modified and the type-checking process must be repeated. For complex integration mappings, reaching a DTD-conforming integrated view through repeated trial-and-error can be a very long and arduous process. Second, while Turing-Complete XML query languages (such as XQuery) can express very complex integration mappings, optimization for such languages still remains to be explored, and their complexity makes it desirable to work within a more limited formalism. That is, when it comes to large-scale XML data integration, it is often desirable to trade excessive expressive power for efficiency and ease-of-use.

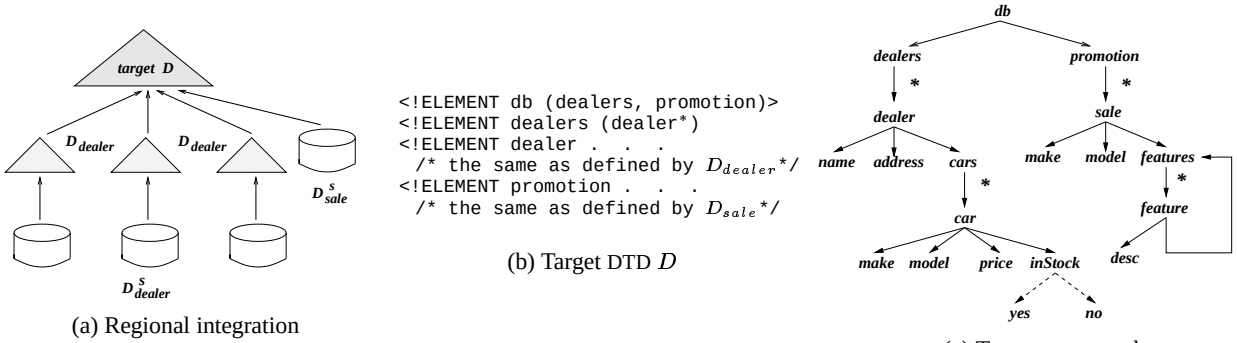
Prior Work. Although a number of integration systems have been developed for semistructured data and XML [3, 6, 13, 16, 10, 15, 24, 25], they typically provide very little support for modularity or ensuring DTD-conformance, especially when the prescribed DTD is recursive and/or non-deterministic. Similarly, in the realm of commercially-available systems, support for modular, DTD-directed XML data integration is either non-existent or still at a fairly primitive stage. Nimble's Integration suite (`www.nimble.com`) allows users to pose queries over distributed XML data sources to synthesize a result XML document but does not address the issues of schema conformance or modular integration specifications. BEA's WebLogic Integration and Liquid Data suites (`www.bea.com`) allow for XML-to-XML transformations through a visual mapping tool that allows users to specify simple matchings between schema elements; it is unclear how these tools can be used to specify complex, hierarchical integration with a complicated target DTD.

Active XML (AXML) [1, 23] proposes a novel notion of intentional XML documents with embedded function calls to remote Web services. AXML is designed to support data exchange and Web-service calls via an unlimited class of embedded functions; furthermore, it also supports XML data integration through the use of XML tree templates with embedded function calls. However, this template-based approach to integration can typically only produce mild variations of a fixed document structure. The functionality of AXML for supporting embedded Web services is, in a sense, complementary to the problem of DTD-directed XML integration, where the goal is to construct an integrated view guaranteed to conform to a predefined, possibly complex DTD.

Closest to our work are Attribute Integration Grammars (AIGs), a grammar-based formalism for schema-directed integration of relational data in XML [5, 4]. AIGs extend a target DTD with tuple-valued attributes and SQL queries over the relations. These earlier proposals are, however, inadequate for XML integration. First, they are restricted to *flat, relational sources* [5, 4]. Second, and perhaps most importantly, while AIGs guarantee schema-conformance, they are *not composable*: a large integration task must be specified with a *single* AIG on top of a large DTD. Developing an effective, modular solution for large-scale, DTD-directed XML data integration poses a whole new set of difficult research challenges, including the need for a significantly more powerful, composable formalism and novel optimization/evaluation techniques.

Our Contributions. In this paper, we propose a novel formalism, *XML Integration Grammars* (*XIGs*), for the modular specification of complex, DTD-directed XML integration tasks. More concretely the key contributions of our work are summarized as follows.

• **Introduction of XIGs: The First Composable Specification Language to Support Complex, DTD-Directed XML Integration.** Our XIG formalism represents the first effort for modular, DTD-directed XML integration, by incorporating tree attribution, XQuery, and embedded local/remote XIG calls. In a nutshell, XIGs



(a) Regional integration

(b) Target DTD D

(c) Target DTD graph

Figure 2: Example: XML integration

are built using *localized semantic rules* around productions in the target DTD which can comprise (1) *queries* over the XML sources expressed in a fragment of the XQuery language, and (2) *embedded XIG calls* which can be either local (*i.e.,* executed at the same site) or remote (*i.e.,* executed remotely). Our XIG semantic rules guarantee DTD-conformance by constructing *tree-valued attributes* following the target DTD productions. XIGs are also *composable*: local/remote XIGs can be treated as "black-box" functions returning DTD-conforming XML trees, and can be embedded in an XIG definition in order to compute certain tree-valued attributes. Thus, XIGs support modular specifications of XML integration, with benefits including ease of specification/verification and reusable code.

Note that our XIG formalism is *not* yet another XML transformation language; instead, XIGs are to serve as a *user/application-level interface* for specifying DTD-directed integration in XQuery. XIGs provide *guidance* on how to specify XML integration in a manner that automatically guarantees DTD conformance. Furthermore, XIGs rely on semantic rules that are *local* to each DTD production, thereby allowing integration sub-tasks to be declaratively specified for each production in isolation – this allows our XIGs to simplify a complex integration task by breaking it into *small, production/element-specific pieces* that can be specified independently. XIG definitions rely solely on DTDs and XQuery, and there is no need to study any new, specialized integration language. Moreover, XIGs can be defined using some specific XQuery fragment that allows for more optimizations than full-fledged XQuery, and, thus, can promise better performance. We are currently developing APIs and tools to facilitate integration specifications with XIGs.

• **XIG-Based Middleware-System Architecture for DTD-Directed Integration, Incorporating Novel, Efficient XIG-Evaluation Algorithms.** Based on our XIG formalism, we propose a middleware system for DTD-directed XML integration, along with algorithms for efficiently evaluating XIGs. Note that, in principle, it may be possible to translate any XIG into an XQuery expression and evaluate it using an XQuery-execution engine; however, taking a middleware-based approach to XIG evaluation allows us to devise several effective, XIG-specific optimization techniques that can be applied outside the generic XQuery engine. More specifically, we demonstrate how to capture recursive DTDs and recursive XIGs in a uniform framework, and propose a cost-based algorithm for scheduling local XML queries/XIGs and remote XIGs to maximize parallelism. We also provide an algorithm for merging multiple XQuery expressions into a single query without using "outer-union/outer-join". Combined with possible optimization techniques for the XQuery fragment used in XIG definitions, such optimizations can yield efficient evaluation strategies for DTD-directed XML integration.

• **Preliminary Results from a Prototype System Implementation Validating our Approach.** We have implemented a prototype middleware system for DTD-directed XML integration based on our XIG formalism and algorithms. Our prototype is built on top of the Galax XQuery engine (`db.bell-labs.com/galax`) and has been tested with several synthetic XML data sets. Our experimental results validate our approach, clearly demonstrating the effectiveness of our XIG query-merging optimizations. Another set of experiments based on randomly-generated XIG query-dependency graphs verifies the effectiveness of our XIG-scheduling strategies.

Our XIGs are a first, yet concrete, step toward XML integration directed by XML Schema [28]. The ultimate goal is to provide a design tool for XQuery to facilitate schema-directed integration of XML data, validating constraints in parallel with DTD-directed XML document generation in a uniform framework (note that run-time constraint/DTD checking is quite different from static analysis of consistency of XML Schema). The notion of XIGs is inspired by composable [14] and higher-order [27] attribute grammars, which have proved useful in compiler construction. XIGs are not mild extensions of those formalisms: their definitions and evaluations are very different. Among other things, the attribute grammar formalisms are to parse an input string with a source context-free grammar and then compute attributes associated with the parse tree; in contrast, XIGs are to generate an XML tree directed by a target DTD; the target XML tree is computed via queries in a fragment of XQuery rather than syntactic parsing.

2. PRELIMINARIES

DTDs. Without loss of generality, we define a DTD to be (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 of the following form:

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

where ϵ is the empty word, B is a type in Ele (referred to as a *child type* of A), and '$+$', '$,$' and '$*$' denote disjunction, concatenation and the Kleene star, respectively (we use '$+$' instead of '$|$' to avoid confusion). We refer to $A \rightarrow P(A)$ as the *production* of A. It has been shown in [5] that all DTDs can be converted to this form in linear time by introducing new element types. To simplify the discussion we do not consider XML attributes, which can be easily incorporated. Examples of DTDs can be found in Figs. 1 and 2.

An XML document (tree) T *conforms to* a DTD D if (1) there is a unique node, the *root*, in T labeled with r; (2) each node in T is labeled either with an Ele type A, called an A *element*, or with PCDATA, called a *text node*; (3) each A element has a list of children of elements and text nodes such that 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. We call T a *document* (instance) of D if T conforms to D.

XQuery. XIGs can be defined with any fragment of XQuery that supports FLWR constructs [11] and permits effective optimization.

Specifications of XML integration typically do not need a Turing-Complete language. The trade-off for the expressive power of the full-fledged XQuery is to leverage techniques for optimization and termination analyses that are not applicable to Turing-Complete languages, and to efficiently conduct XML integration tasks commonly found in practice.

Given a fragment of XQuery, we extend its syntax by incorporating XIG calls in the top level `let` clauses. Specifically, we consider the class of queries defined as follows:

$$Q ::= q \mid let\ \$x := XIG\_call\ Q,\ \ XIG\_call ::= U_V{:}V(U) \mid V(U)$$

where q is a query in the fragment, V is an XIG, U_V is the URI of V (for remote XIG), and U is the URI of a source XML document. Here $U_V{:}V(U)$ denotes a remote XIG call, and $V(U)$ is a local XIG call. The semantics of a query "$let\ \$x := XIG\_call\ Q$" is to first evaluate the XIG, assign the result of the evaluation to $x as a constant, and then evaluate the XQuery expression q. We refer to this extension as XQ^e. As will be seen shortly, an XIG is defined with a target DTD D and is evaluated to an XML document of D; thus the XIG can be viewed as an XML expression of "type" D.

As will be seen in Sec. 5, although theoretically any XIG can be translated to an XQuery function and be evaluated using an XQuery-execution engine, there are performance reasons for not doing this.

3. XML INTEGRATION GRAMMARS (XIGs)

XIG Syntax. An XIG V is a partial function from a collection X of XML sources to documents of a target DTD D, referred to as *an XIG from X to D* and denoted by $V : X \to D$. Specifically, let $D = (Ele, P, r)$; then, V is defined on top of D as follows.

- *Attributes:* For each element type A in Ele, V defines an *inherited* attribute $Inh(\mathtt{A})$ and a *synthesized* attribute $Syn(\mathtt{A})$, whose values are a single XML element. Inherited attributes are computed top-down and are used to pass data parameter, whereas synthesized attributes are computed bottom-up and are used to hold partial results (XML subtrees).
- *Rules:* For each production $p = A \to \alpha$ in P, V defines a set $rule(p)$ of semantic rules consisting of: (1) for each child type B in α, a rule for computing $Inh(\mathtt{B})$ by extracting data from sources via an XQ^e query, which may take the parent $Inh(\mathtt{A})$ as a parameter; and, (2) for the parent type A, a rule for $Syn(\mathtt{A})$ by grouping together $Syn(\mathtt{B})$ for all B in α.
- *Input/Output:* The sources X is called the *input* of V, the value of the synthesized attribute $Syn(\mathtt{r})$ of the root is the *output* of V, and D is the *type* of V.

Given an input X, $V(X)$ returns $Syn(\mathtt{r})$, which is an XML document conforming to the target DTD D.

Example 3.1: Fig. 3 gives an XIG that defines a view for local dealers: given the URI U of a local document specified by the DTD D^s_{dealer} of Fig. 1(b), $V_{dealer}(U)$ returns an XML document conforming to D_{dealer} of Fig. 1(c). Thus V_{dealer} can be treated as a function: $D^s_{dealer} \to D_{dealer}$. The XIG is defined on top of the (target) view DTD D_{dealer} with XQ^e queries and tree attribution. For each element type A in D_{dealer}, it defines two attributes $Inh(\mathtt{A})$ and $Syn(\mathtt{A})$, which contain a single XML element as their value. For each production of D_{dealer}, it defines a set of rules via XQ^e to compute the inherited attributes of the children, using the inherited attribute of the parent as a parameter. In addition, there is a single rule for computing the synthesized attribute of the parent, by collecting the synthesized attributes of its children. □

For a production $p = A \to \alpha$, the semantic rules $rule(p)$ enforce that $Syn(\mathtt{A})$ is indeed an A element. The generic form of the (per-production) XIG semantic rules is as follows.

XIG: $V_{dealer}(U)$

dealer → name, address, cars
Inh(name)= {U/dealer/name}; *Inh*(address) = {U/dealer/addr};
Inh(cars) = {U/dealer/cars};
Syn(dealer) = <dealer> {*Syn*(name)} {*Syn*(address)}
{*Syn*(cars)} </dealer>

cars → car*
Inh(car) ← for $c in *Inh*(cars)/car return $c;
Syn(cars) = <cars> {⊔*Syn*(car)} </cars>

car → make, model, price, inStock
Inh(make) = {*Inh*(car)/make}; *Inh*(model) = {*Inh*(car)/model};
Inh(price) = {*Inh*(car)/invoice/price}; *Inh*(inStock) = {*Inh*(car)};
Syn(car) = <car> {*Syn*(make)} {*Syn*(model)}
{*Syn*(price)} {*Syn*(inStock)} </car>

inStock → (yes + no)
Inh(yes) = {if *Inh*(inStock)[invoice/quantity < 1]
then <empty/> else <yes/>}
Inh(no) = {if *Inh*(inStock)[invoice/quantity < 1]
then <no/> else <empty/>}
Syn(inStock) = {if *Inh*(inStock)[invoice/quantity < 1]
then *Syn*(no) else *Syn*(yes)}

yes → ϵ
Syn(yes) = *Inh*(yes) /* similarly for **no** */

name → PCDATA
Syn(name) = <name> {*Inh*(name)/text()} </name>
/* similarly for **address, make, model, price** */

Figure 3: XIG $V_{dealer}(U)$ defining dealer views

• $p = A \to \mathtt{PCDATA}$. Then, $rule(p)$ is defined as $Syn(\mathtt{A}) = \{Q(Inh(\mathtt{A}))/text()\}$, where Q is an XQ^e query that returns PCDATA and treats $Inh(\mathtt{A})$ as a constant parameter. See, *e.g.*, the rule for production *name → PCDATA* in the XIG V_{dealer} of Fig. 3.

• $p = A \to B_1, \ldots, B_n$. Then, $rule(p)$ consists of $Inh(\mathtt{B_i}) = Q_i(Inh(\mathtt{A}))$, for each $i \in [1, n]$, and $Syn(\mathtt{A}) =$ <A> $\{Syn(\mathtt{B_1}) \ldots Syn(\mathtt{B_n})\}$ </A>, where, for each $i \in [1, n]$, Q_i is an XQ^e query that returns a single element (subtree). As an example, see the rules for *car → make,model,price,inStock* in V_{dealer}.

• $p = A \to B_1 + \ldots + B_n$. Then $rule(p)$ is defined as:

$Inh(\mathtt{B_i})$ = **let** $c := $Q_c(Inh(\mathtt{A}))$ **return** {**if** $C_i($c)$ **then** $Q_i(Inh(\mathtt{A}))$ **else** <empty/>} /* for $i \in [1, n]$*/,

$Syn(\mathtt{A})$ = **let** $c := $Q_c(Inh(\mathtt{A}))$ **return**
{if $C_1($c)$ **then** <A> $Syn(\mathtt{B_1})$ </A> **else** . . .
else if $C_n($c)$ **then** <A> $Syn(\mathtt{B_n})$ </A> **else** <empty/>}

where Q_c is an XQ^e query, referred to as the *condition query* of $rule(p)$, which is evaluated only once for all the rules in $rule(p)$; Q_i is an XQ^e query that returns a single element; and, the C_i's are *mutually-exclusive* Boolean XQ^e expressions: one and only one C_i is true for all $i \in [1, n]$. See, *e.g.*, the rules for the production *inStock → yes+no* in V_{dealer}.

• $p = A \to B^*$. Then, $rule(p)$ is defined as:

$Inh(\mathtt{B})$ ← **for** $b **in** $Q(Inh(\mathtt{A}))$ **where** C($b) **return** $b,

and $Syn(\mathtt{A}) =$ <A> $\sqcup Syn(\mathtt{B})$ </A>, where Q is an XQ^e query that may return a (possibly empty) set of elements, C is an XQ^e Boolean expression, and '⊔' is a list constructor. For each $b generated by Q, the rules for processing B are evaluated, treating $b as a value of $Inh(\mathtt{B})$. Then, the rule for $Syn(\mathtt{A})$ groups together the corresponding $Syn(\mathtt{B})$'s into a list using ⊔ in the default document order. See, *e.g.*, the rules for *cars → car** in V_{dealer}.

• $p = A \to \epsilon$. Then, $rule(p)$ is defined as $Syn(\mathtt{A}) = Q(Inh(\mathtt{A}))$, where Q is an XQ^e query such that $Q(Inh(\mathtt{A}))$ returns either <A/>, or <empty/> if the value of $Syn(\mathtt{A})$ is not to be included in the target document. See, *e.g.*, the rule for *yes → ϵ* in V_{dealer}.

Several subtleties are worth mentioning. First, recall that $Syn(\mathtt{A})$ is defined in terms of $Syn(\mathtt{B_i})$. In the rule for computing $Syn(\mathtt{A})$ one may replace $Syn(\mathtt{B_i})$ with the XQ^e query for computing $Syn(\mathtt{B_i})$

(defined in the rules for B_i). For example, in the XIG V_{dealer}, the rules for `dealer` and `car` can be rewritten as:

dealer → name, address, cars
```
Inh(cars) = {U/dealer/cars};
Syn(dealer) = <dealer> {U/dealer/name}
        {U/dealer/addr} {Syn(cars)} </dealer>
```
car → make, model, price, inStock
```
Inh(inStock) = {Inh(car)};
Syn(car) = <car>{Inh(car)/model} {Inh(car)/make}
    {Inh(car)/invoice/price} {Syn(inStock)}</car>
```

These substitutions can avoid unnecessary computation of inherited attributes that are not needed elsewhere. Second, as XIGs support tree attribution and return XML trees, semantic attributes can be computed via other XIGs; such an example will be given in the rule for *Syn*(`promotion`) in the XIG V of Fig. 5. Furthermore, as embedded XIGs ensure conformance to their target DTDs, one can use them as expressions without complicating the type analyses. This makes XIGs composable. Finally, observe that when $Inh(\mathtt{A})$ is the empty tree, $Q(Inh(\mathtt{A}))$ is not necessarily empty.

XIG Semantics. We next give a simple operational semantics for an XIG $V : X \rightarrow D$. Given an instance of X, V evaluates its attributes via its rules, and returns $Syn(\mathtt{r})$ of the root r of D as its output. The evaluation is carried out top-down, using a stack. The root r is first pushed onto the stack. For each node A at the top of the stack, we compute its subtree $Syn(\mathtt{A})$. To do this, we first identify the production $p = A \rightarrow \alpha$ in D, and for each B in α, we evaluate $Inh(\mathtt{B})$ using the semantic rules in $rule(p)$. The exact procedure depends on the specific form of the p production. For example, if $p = A \rightarrow B_1, \ldots, B_n$, then for each B_i, we compute $Inh(\mathtt{B_i})$ by evaluating $Q_i(Inh(\mathtt{A}))$; we then push B_i onto the stack and proceed to process them in the same way using the value of $Inh(\mathtt{B_i})$; then, after all the B_i's are evaluated and popped off the stack (*i.e.*, when all the $Syn(\mathtt{B_i})$'s are available), we compute $Syn(\mathtt{A})$ by collecting all the $Syn(\mathtt{B_i})$'s, such that A has a unique B_i child for each $i \in [1, n]$. (The process for other production rules is similar; due to space constraints, we defer the details to the full paper.) Finally, after $Syn(\mathtt{A})$ is computed, we pop A off the stack, and use $Syn(\mathtt{A})$ to evaluate other nodes until no more nodes are in the stack. At this stage, $Syn(\mathtt{r})$ is computed and returned as the output of the XIG evaluation. Note that for each A, its inherited attribute is evaluated first, then its synthesized attribute, which is an A-subtree. The evaluation takes *one-sweep*: each A element is visited twice, first pushed onto the stack and then popped off after its subtree is constructed. It should be mentioned the conceptual evaluation strategy given above is just to illustrate the semantics of XIGs; we shall provide optimization techniques in Section 6.

4. CASE STUDY

To illustrate the idea of DTD-directed integration with XIGs, consider the integration described in Example 1.2. The regional integration is to extract data from dealer views and a sale document, and construct a target document conforming to the target DTD D of Fig. 2(b), where the dealer views are themselves mapped from local sources at dealer's sites. We divide this task into three parts, and specify each with an XIG as follows.

• $V_{dealer} : D^s_{dealer} \rightarrow D_{dealer}$ is the XIG of Fig. 3 that defines a view for dealers: given the URI U of a local document specified by the DTD D^s_{dealer} of Fig. 1(b), $V_{dealer}(U)$ returns an XML document conforming to D_{dealer} of Fig. 1(c). Each local dealer has a V_{dealer} residing at its site and serving as a view. While the view DTD D_{dealer} is visible to the users, the source DTD D^s_{dealer} and the definition of V_{dealer} are not. The view does not reveal confidential information about `invoice` and `quantity`.

XIG: $V_{sale}(X)$

promotion → sale*
```
Inh(sale) ← for $c in X/sale/cars/car return $c;
Syn(promotion) = <promotion>{⊔Syn(sale)}</promotion>
```
sale → make, model, features
```
Inh(make)={Inh(sale)/make};  Inh(model)={Inh(sale)/model};
Inh(features) = {Inh(sale)/fids};
Syn(sale) = <sale> {Syn(make)} {Syn(model)}
          {Syn(features)} </sale>
```
features → feature*
```
Inh(feature) ← for $f in Inh(features)/fid  return $f;
Syn(features) = <features>{⊔Syn(feature)}</features>
```
feature → desc, features
```
Inh(desc)=X/sale/features/feature[fid=Inh(feature)]/desc;
Inh(features)=X/sale/features/feature[fid=Inh(feature)]/fids;
Syn(feature)=<feature>{Syn(desc)}{Syn(features)}</feature>
```
make → PCDATA
```
/* similarly for model, desc */
Syn(make) = {Inh(make)}
```

Figure 4: XIG $V_{sale}(X)$ for converting sale data

XIG: $V(R, X)$

db → dealers, promotion
```
Syn(db) = <db> {Syn(dealers)} {Syn(promotion)} </db>
```
promotion → sale*
```
Syn(promotion) = V_sale(X)
```
dealers → dealer*
```
Inh(dealer) = for $Y in R/dlink return $Y;
Syn(dealers) = <dealers>{⊔Syn(dealer)}</dealers>
```
dealer → name, address, cars
```
Inh(name) = let $p := Inh(dealer)/U_V  /* similarly for */
            let $u := Inh(dealer)/U   /* address, cars */
            let $v = $p:V_dealer($u)
            return $v/dealer/name;
Syn(dealer) = <dealer> {Syn(name)} {Syn(address)}
                {Syn(cars)} </dealer>
```
cars → car*
```
Inh(car) ← let $s := V_sale(X)
           for $c in Inh(cars)/car
               $c' in $s/promotion/sale
           where $c/make=$c'/make and $c/model=$c'/model
           return $c;
Syn(cars) = <cars> {⊔Syn(car)} </cars>
```
car → make, model, price, inStock
```
Syn(car) = Inh(car)
```
name → PCDATA
```
/* similarly for address */
Syn(name) = Inh(name);
```

Figure 5: XIG $V(R, X)$ for regional integration

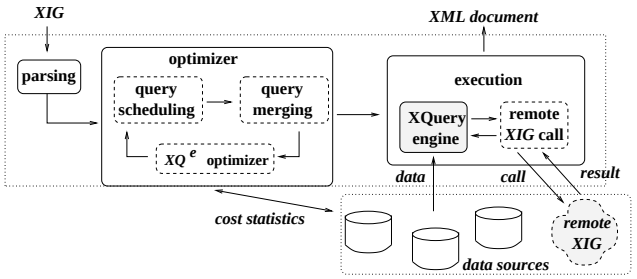
• $V_{sale} : D^s_{sale} \rightarrow D_{sale}$ is an XIG that converts sale data: given the URI X of a sale document specified by D^s_{sale} of Fig. 1(a), $V_{sale}(X)$ returns an XML document conforming to the DTD D_{sale} of Fig. 1(c). This XIG V_{sale} is local: it is at the integration site.

• V is an XIG for regional integration: it is defined with V_{dealer} as a remote XIG and V_{sale} as a local XIG. It takes as input the URI X of the sale source and an XML file R containing information for dealers in the region. Specifically, R consists of a sequence of `dlink`'s, and each `dlink` is of the form (U_V, U), where U_V is the URI of V_{dealer} and U is the URI of the local source data at the same dealer site[1]. The XIG V invokes $V_{sale}(X)$ and $U_V : V_{dealer}(U)$ for each (U_V, U) to collect data from dealer sources and then constructs an XML document conforming to the target DTD of Fig. 2(b).

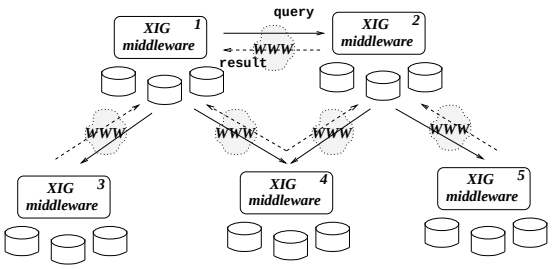
V_{dealer} has been presented in Fig. 3. V_{sale} and V are presented next.

Sale Data. An XIG $V_{sale} : D^s_{sale} \rightarrow D_{sale}$ for converting sale data is given in Fig. 4. Given a source X, $V_{sale}(X)$ is evaluated top-down. Starting from `promotion`, it uses an XQ^e query

[1]Assume that the definition of V_{dealer} is not accessible to anyone except the dealer, and that the local source document is only accessible to the dealer or via V_{dealer}, although their URIs are public.



(a) XIG middleware architecture



(b) XIG middleware communication

Figure 6: XIG-based framework for DTD-directed integration

to extract `car` elements from X, and treats each `car` $\$c$ as a value of *Inh*(`sale`). For each $\$c$ the rules for `sale` are evaluated, which compute *Inh*(`make`), *Inh*(`model`) and *Inh*(`features`) by extracting the corresponding fields from $\$c$, and invoke the rules for `features` in turn. Note that `features` is recursively defined and thus its subtree has an unbounded depth. The depth is determined at run time: if the XQ^e query for computing *Inh*(`feature`) does not find any `fid`, the rules for computing `feature` subtree are not triggered, the list $\sqcup$*Syn*(`feature`) is empty, and the construction of `features` subtree is complete. After all the subtrees *Syn*(`features`) are constructed, *Syn*(`sale`) is computed, followed by *Syn*(`promotion`). This shows that XIGs are capable of expressing XML integration with a recursive DTD. Note that XIGs adopt a *data-driven semantics*: the XML tree height in the recursive case and the choice of a production in the non-deterministic case are determined by queries on the source data at run-time.

Regional Integration. Finally, we provide an XIG V in Fig. 5 for integrating XML data of car dealers and promotion information. It is defined with embedded XIGs: local XIG V_{sale} and remote XIGs V_{dealer}. The local XIG V_{sale} is invoked to produce $Syn($`promotion`$)$, which is an XML tree conforming to the `promotion` type of the target DTD D. To produce *Syn*(`dealers`), it first finds from the input document R the URI $\$p of the view V_{dealer} and the URI $\$u of the local dealer source for each dealer. For each pair ($\$p, $\$u), it then invokes the remote XIG V_{dealer} via $\$p:V_{dealer}(\$u)$ to compute its view. The result of the computation, $\$v, is shipped back to the integration site and is used as a constant in the queries for computing *Inh*(`name`), *Inh*(`address`) and *Inh*(`cars`). To find the cars that are promoted, *i.e.*, those appearing in $V_{sale}(X)$, V invokes $V_{sale}(X)$ and selects cars that are in both $\$v and $V_{sale}(X)$. For each car $\$c selected, it simply returns $\$c as *Syn*(`car`), since V_{dealer} ensures that $\$c indeed conforms to the `car` type in the target DTD D. This example shows how a complex integration task can be carried out in terms of component XIGs.

5. XML INTEGRATION WITH XIGs

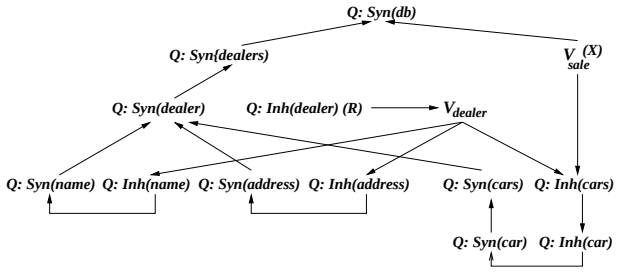
XIG Middleware Architecture. We propose a middleware system for XIG evaluation. As shown in Fig. 6(a), our middleware takes an XIG V as an input, evaluates V and generates an XML document conforming to the target DTD of V. More specifically, our XIG middleware servers use a local XQuery engine to evaluate XQ^e queries over local data sources. An XIG server also communicates with other servers. It invokes a remote XIG V' along the same lines as a *remote procedure call*: it sends a request along with appropriate data parameters to the server where V' is located; the remote server then evaluates V' and sends the result back. Note that a remote XIG may in turn invoke XIGs at other servers. For example, as depicted in Fig. 6(b), server 1 invokes remote XIGs at servers 2, 3 and 4, and to evaluate the remote XIG call of server 1, server 2 in turn invokes XIGs at servers 4 and 5.

Note that, although theoretically one can translate an XIG specification of a complex integration task into a large XQuery function (by simply *merging* the localized semantic rules for all DTD productions), such brute-force query-merging typically leads to poor performance in practice. First, injudicious query merging relies on the optimizer of the underlying XQuery-engine to optimize a large query, schedule execution of queries and XIGs, and produce efficient execution plans. However, even sophisticated relational optimizers do not work well on large SQL queries, not to mention XQuery optimizers that remain to be explored. Indeed, injudicious query-merging has proved ineffective in relational publishing/integration practice, and this was one of the main motivations for developing middleware systems and appropriate optimization techniques [4, 5, 6, 15, 26]. Second, it is possible to develop optimization techniques that are effective for the specific XQ^e fragment used in our XIGs but are not applicable to XQuery in general and are unlikely to be supported inside a generic XQuery engine. This suggests that potential optimizations developed for our XQ^e fragment should probably be accommodated in our middleware server (outside the XQuery engine). Third, XQuery specification [11] has not yet defined remote procedure calls, and thus the feasibility of such a brute-force XIG-to-XQuery translation is pending the availability of XQuery support for remote procedure calls.

Thus, central to our XIG middleware is an *XIG optimizer* module (Fig. 6(a)) whose goal is to generate an efficient execution plan that minimizes the response time of an XIG evaluation. After an initial parsing phase, which derives the dependency relation on the queries of the input XIG V, our XIG optimizer generates an execution plan for V using a cost-based approach that: (1) merges certain queries in V that are processed at the same source into a larger query to reduce communication costs, (2) schedules execution of XQ^e queries and XIGs to increase parallelism, and (3) leverages an external optimizer for (partially-merged) XQ^e queries to produce efficient XQ^e-execution plans. Finally, the execution plan is carried out, by evaluating optimized (merged) XQ^e queries embedded in V via a local XQuery engine, and by invoking remote XIG calls. Compared to its counterparts for XML publishing [5, 10, 15] and relational data integration in XML [4], our XIG optimizer raises a number of new issues that we briefly address below; we provide detailed optimization algorithms in Section 6.

XIG Recursion vs. DTD Recursion. Our XIG-based integration framework involves two forms of recursion: recursive target DTDs and recursive XIGs (*i.e.*, XIGs defined in terms of themselves). In contrast, previous work on XML integration/publishing either ignores recursion or considers recursive DTDs only [5, 4].

The key observation here is that recursive DTDs can be captured with recursive XIGs. Indeed, the computation of any recursively-defined A-elements can be rewritten to an equivalent local, recursive XIG V_A. For example, we can easily define a recursive XIG for computing the recursively-defined `features` elements in Fig. 4. The rewriting is conducted in the parsing phase of the XIG mid-



(a) The query dependency graph of V

(b) Query dependency (c) Query merging

Figure 7: Query dependency, scheduling and merging

dleware of Fig. 6(a). This allows us to handle the two forms of recursion in a uniform framework.

Recursive XIG evaluation also raises *termination* issues. To avoid potential infi nite invocation loops, our XIG middleware servers employ a dynamic control mechanism based on keeping track of local XIG invocations and using that information to detect cycles in the call chain. Furthermore, XIG servers also cache the results of XIG evaluations to avoid possible redundant computation. (Due to space constraints, the details of dealing with recursion and recursive XIG evaluation are deferred to the full paper.)

Query Dependencies. XIGs support sideways information passing in an implicit way: common computation is specifi ed with an XIG, which is invoked wherever it is needed. For example, the XIG V of Fig. 5 uses V_{sale} to specify the computation of the `promotion` subtree, and invokes V_{sale} at two different places where the subtree is needed. This yields a more flexible information passing mechanism than other proposals, *e.g.*, data passing between siblings [4]. However, naive evaluation of V may lead to repeated evaluation of V_{sale}. To eliminate unnecessary XIG recomputation, our system explicitly captures the dependencies among XIG queries through a *query dependency graph*.

The *query dependency graph* of an XIG V contains a node for each query/XIG in V, and a directed edge from Q_1 to Q_2 if and only if the result of Q_1 is consumed by Q_2. For example, the dependency graph of the XIG V of Fig. 5 is depicted in Fig. 7(a), in which $Q : Inh(\texttt{A})$ and $Q : Syn(\texttt{A})$ denote the queries for computing $Inh(\texttt{A})$ and $Syn(\texttt{A})$, respectively. The graph describes top-down dependencies on inherited attributes, bottom-up dependencies on synthesized attributes, and producer-consumer relationships introduced by embedded (local and remote) XIGs. Note that there is a single node representing the XIG V_{sale}, which is evaluated once and its result is used to compute both *Syn*(`db`) and *Inh*(`car`). Also, note that, once recursively-defi ned elements are rewritten as recursive XIGs (XQ^e functions), the parsing phase of the middleware inspects whether the query dependency graph is cyclic and allows only *directed, acyclic graph (DAG)*. (Cyclic dependency graphs are infeasible and are rejected for evaluation.)

Query Scheduling. Based on the query dependencies in an XIG, our XIG-middleware optimizer orders the execution of queries/XIGs such that local queries and remote XIGs can be evaluated in parallel. For example, consider the dependency of Fig. 7(b), where $Q_1, \ldots, Q_6$ are local queries and V is a remote XIG call. One may want to execute Q_4 before Q_2 and Q_3 such that V can be evaluated by another server in parallel with Q_2, Q_3, and thus improve the overall response time. It is, however, nontrivial to develop an optimal scheduling strategy. Among other things, XIGs are complex tasks and a remote XIG may trigger other remote XIGs. For example, referring to Fig. 6(b), server 1 triggers remote XIGs at servers 2 and 4, while the remote XIG at server 2 may invoke another XIG at server 4, competing for the resources of server 4.

Query Merging. Another optimization technique is to *merge* multiple XQ^e queries into a single query. For example, an XIG without remote XIG calls can be easily rewritten as a single XQ^e query. The merged XQ^e queries can then be optimized via an XQ^e optimizer (Fig. 6(a)). Query merging could reduce the communication overhead between the middleware and the underlying XQuery engine, and thus potentially speed up the query execution. On the other hand, injudicious query merging may change the query dependency graph, lead to unnecessary delay of other query executions, and decrease parallelism. For example, merging Q_2, Q_3, Q_4 of Fig. 7(b), results in the query dependency DAG shown in Fig. 7(c). As a result, the remote XIG call V is delayed as it becomes dependent on Q_2 and Q_3 as well, thus decreasing the potential parallelism among remote XIGs and local queries. Clearly, the decision of whether or not to merge certain queries should be *cost-based*; furthermore, given the dependence of execution cost on scheduling, query merging and scheduling are obviously dependent on each other.

6. XIG EVALUATION AND OPTIMIZATION

We next present two cost-based algorithms, scheduling and merging XQ^e/XIG expressions, to improve the response time of XIG evaluation. These can be combined with optimizations for XQ^e queries, *i.e.*, the middleware is open to and can accommodate optimization techniques for specifi c XQuery fragments.

Scheduling an XIG Evaluation. Assume a given *query dependency DAG* that captures the data and execution dependencies between the various components (namely, XQ^e and XIG expressions) comprising an XIG. Note here that, although XQ^e queries are typically executed locally, XIG nodes can be either local or remote (*i.e.*, with the XIG executed at a remote server). Effectively scheduling such an XIG-dependency DAG over an architecture of distributed servers is a very challenging problem. In addition to all the complications typically associated with scheduling a DAG of interdependent (*i.e.,precedence-constrained*) tasks over a distributed architecture (*e.g.*, communication overhead, parallel execution), a crucial, distinguishing characteristic of our problem is that XIGs are *complex tasks* that can invoke other (local or remote) XIG tasks for their evaluation. In essence, this means that, instead of simply utilizing a single server, the evaluation of an XIG node in the query dependency DAG can utilize several different servers (through embedded remote XIG calls). This strict co-scheduling requirement makes our XIG-scheduling problem quite different from those studied in the context of conventional scheduling for parallel/distributed systems, where the assumption is that either each task uses a single site [20] or that tasks can be migrated across different subsets of sites [8, 17].[2] Similarly, work on dynamic/adaptive scheduling strategies for distributed database and data-integration systems (*e.g.*, [21, 7]) is applicable only at run-time, that is, when the query plan is actually executed. In contrast, our focus here is on *compile-time* scheduling in order to determine an effective XIG-evaluation plan; thus, our scheduling model needs to be able to capture all the complexities of XIG evaluation.

Given an XIG query dependency graph G, determining a schedule for G over the underlying architecture of distributed servers that

[2]Note that the corresponding scheduling problem for AIG evaluation [4] also assumes only single-site queries.

minimizes the overall XIG execution time (*i.e.*, the *makespan* of the schedule) is an essential step in optimizing XIG evaluation. Our scheduler needs to make its decisions at XIG-optimization time, which means that it needs to rely on estimates for query/XIG execution costs, result sizes, and communication overheads. In our development and ongoing implementation, we assume that each server s in the underlying system offers a query/XIG-costing API that, given a query/XIG node t to be executed at s returns (1) an estimate $l(t)$ for the processing time of t's execution on s; and, (2) a subset of sites $S(t)$ (including s) that are utilized in the evaluation of t (where $|S(t)| > 1$, if t is an XIG node with embedded remote XIG calls).[3] Thus, for each node t in the dependency graph, the underlying server APIs provide us with the execution time of t as well as the (sub)set of servers used during this execution. Our XIG-scheduling problem can then be abstracted as follows.

XIG SCHEDULING($G, \mathcal{S}, l(), S()$)
- **Given:** A dependency DAG $G = (V, E)$ defining a partial order (precedence) relation "$\prec$" over a set of tasks $V = \{t_1, \ldots, t_n\}$; set of distributed servers $\mathcal{S}$. For each task $t \in V$, $l(t)$ is the execution time of t and $S(t) \subseteq \mathcal{S}$ is the set of servers used during t's execution.
- **Find:** An assignment of start times to tasks $\mathsf{start} : V \rightarrow R^+$, such that:
  1. Concurrently-executing tasks *do not collide* on servers – that is, for all $i \neq j$, if $[\mathsf{start}(t_i), \mathsf{start}(t_i) + l(t_i))] \cap [\mathsf{start}(t_j), \mathsf{start}(t_j) + l(t_j))] \neq \phi$ then $S(t_i) \cap S(t_j) = \phi$.
  2. Precedence constraints are satisfied – that is, for all $t_i \prec t_j$ we have $\mathsf{start}(t_j) \geq \mathsf{start}(t_i) + l(t_i)$; and,
  3. The schedule makespan $\max_i\{\mathsf{start}(t_i) + l(t_i)\}$ is *minimized*.

It is easy to verify that our XIG-scheduling problem is actually the precedence-constrained generalization of the *Set Scheduling* problem recently introduced by Goel et al. [18]. Even for their simpler case of fully-independent tasks (*i.e.*, $\prec = \phi$), Goel et al. demonstrate that the problem is $\mathcal{NP}$-hard and hard to approximate, by giving a simple, approximation-preserving reduction from the *Minimum Graph Coloring* problem [18]. Given the intractability of our XIG SCHEDULING problem, we now propose a heuristic scheduling algorithm for query dependency graphs that produces an approximate solution to our scheduling problem.

In a nutshell, our scheduling algorithm (termed SCHEDXIG) belongs to the class of *list-scheduling* algorithms, originally introduced by Graham for multiprocessor scheduling [19]. SCHEDXIG maintains a list L of *ready* tasks (*i.e.*, tasks whose predecessors in the dependency graph G have already completed), and schedules the next ready task $t \in L$ at the earliest possible start time (*i.e.*, the earliest time at which all servers in $S(t)$ become available). Since our goal is to minimize the overall execution time in the schedule for G, we maintain the tasks in the ready list L sorted in decreasing order of *"criticality"*, where the criticality of a ready task t (denoted by $\mathsf{crit}(t)$) captures t's potential in becoming the bottleneck (*i.e.*, lie in the critical path) for the parallel execution of G. Note that estimating the criticality of a task node in the complex-task model used in our XIG-scheduling problem is non-trivial – our criticality measure needs to account not only for the serialization effects in the parallel execution (introduced by the dependency edges in G), but also for the possibility of collisions of independently-executed tasks utilizing the same server(s). Our SCHEDXIG algorithm employs such a criticality measure that is a simple-to-compute lower bound $\mathsf{crit}(t)$ on the parallel-execution time of all DAG paths rooted at task node t and captures both the serialization and the server-collision effects mentioned above. More formally, let $\mathsf{paths}(t)$ denote the set of all paths rooted at task t (including t itself) and leading to some "sink" node in G, and let $G(t)$ denote the corresponding subgraph of G. Also, given a task t, define the *server-usage vector* $\mathbf{v}(t)$ of t to be a numeric vector of dimensionality $|\mathcal{S}|$ (*i.e.*, the number of servers in the system), and components defined as: $\mathbf{v}(t)[i] = l(t)$ if $i \in S(t)$, and 0 otherwise (where we assume, w.l.o.g, that $\mathcal{S} = \{1, 2, \ldots, |\mathcal{S}|\}$). Thus, $\mathbf{v}(t)$ basically captures the processing-time requirements of t on each server used during t's execution. We estimate the criticality of t, $\mathsf{crit}(t)$, as the *maximum* of the following two quantities:

1. The *Critical-Path Length under* t, $\mathsf{CP}(t) = \max_{p \in \mathsf{paths}(t)} \{\sum_{u \in p} l(u)\}$, which captures the effects of dependencies (*i.e.*, serialization constraints) in the parallel execution of $G(t)$; and,
2. The *Maximum Server Load under* t, $\mathsf{SL}(t) = \max_i \{\sum_{u \in G(t)} \mathbf{v}(u)[i]\}$, which captures the effects of possible server collisions and server bottlenecks during the parallel execution.

Example 6.1: Consider a simple instance of our XIG SCHEDULING problem, with 4 tasks $V = \{t_1, \ldots, t_4\}$ and the task dependencies $t_1 \rightarrow t_2 \rightarrow t_3$, $t_1 \rightarrow t_4$. Assume a 3-server configuration, and let $\mathbf{v}(t_1) = [2, 0, 0]$, $\mathbf{v}(t_2) = [0, 8, 0]$, $\mathbf{v}(t_3) = [5, 0, 0]$, and $\mathbf{v}(t_4) = [0, 0, 10]$. It is easy to see that, in this scenario, $\mathsf{CP}(t_1) = \max\{2+8+5, 2+10\} = 15$ and $\mathsf{SL}(t_1) = \max\{[2+5, 8, 10]\} = 10$, which implies that $\mathsf{crit}(t) = \max\{\mathsf{CP}(t_1), \mathsf{SL}(t_1)\} = 15$; that is, the dominant factor in this parallel execution comes from the serialization in the $t_1 \rightarrow t_2 \rightarrow t_3$ dependency chain. In contrast, assume that t_4 is a complex (XIG) task that utilizes both servers 2 and 3, *i.e.*, $\mathbf{v}(t_4) = [0, 10, 10]$. It is again easy to see that, in this case, even though the critical-path length $\mathsf{CP}(t_1)$ remains the same, the maximum server load becomes $\mathsf{SL}(t_1) = \max\{[2+5, 10+8, 10]\} = 18$, which implies that $\mathsf{crit}(t) = \mathsf{SL}(t_1) = 18$ – thus, the dominant execution-time factor has shifted to the processing bottleneck created by the collision of t_2 and t_4 on server 2. □

The pseudo-code for our SCHEDXIG algorithm is given in Fig. 8; its worst-case time complexity is $O(n|\mathcal{S}| \log n)$ (*e.g.*, using a max-heap for L). Note that, even though we presented the algorithm SCHEDXIG as an optimization-time technique, it is actually an online algorithm that can readily be used to schedule XIG executions at run-time (based on task-criticality estimates) as servers become available. Finally, we should note that our complex-task model can be generalized along the lines of the *preemptable/time-shared* resource model of [17] to allow for servers to be effectively time-shared across different tasks, since, *e.g.*, an XIG node can typically impose different processing requirements on the remote servers it utilizes. This gives rise to several challenging scheduling issues that we are exploring in our ongoing work.

Merging Queries. Query merging may also speed up XIG evaluation; however, it can also change the dependency DAG and, thus, the execution schedule (and corresponding evaluation cost). Thus, query merging and scheduling are clearly inter-dependent. Our query merging problem is to determine, given a dependency graph G, what query nodes to merge such that the *estimated response time* of the resulting dependency graph G' (*i.e.*, the makespan of the schedule returned by SCHEDXIG (G')) is minimized.

Given a dependency graph G, there are exponentially many choices for merging queries in G; moreover, recall that the scheduling problem is already intractable. Given the inherent difficulty of the problem, we outline a greedy heuristic algorithm, termed MERGEXIG, that iteratively calls our SCHEDXIG scheduler for optimizing XIG query merging and evaluation. In a nutshell, MERGEXIG takes

[3] To simplify the exposition, we assume that the query/XIG-processing time $l(t)$ also includes the cost of communicating input/output data to/from the executing server s, which also allows us to leave result-size estimates out of our scheduling-problem formulation. Both aspects can be incorporated into our scheduling model and algorithms in a straightforward fashion.

Procedure SCHEDXIG $(G, \mathcal{S})$
Input: XIG dependency graph G, set of servers $\mathcal{S}$.
Output: Schedule start() for executing G over $\mathcal{S}$.
begin
1. **for each** node t in G **do**
2. compute crit(t) := max{CP(t), SL(t)}
3. L := list of ready XQ^e/XIG tasks in G in decreasing order of crit(t)
4. **while** $L \neq \phi$ **do**
5. $t := L[1]$ /* first ready task in L*/
6. start(t) := earliest time in our schedule that all servers in $S(t)$ become available
7. M := list of tasks in G that become ready after the completion of t (in decreasing crit())
8. L := **merge**$(L - L[1], M)$
9. **endwhile**
end

Figure 8: Our XIG-Scheduling Algorithm.

an XIG dependency graph G as input and returns an efficient evaluation schedule as output. At each step, MERGEXIG considers each pair of query nodes (Q_1, Q_2) in G that are processed at the same source for potential merging into a single query node Q (resulting in a new dependency graph G'); the query pair resulting in the (acyclic) dependency graph G' with the lowest SCHEDXIG-estimated evaluation cost (*i.e.,* the smallest makespan for SCHEDXIG (G')) is merged. The iteration in MERGEXIG continues until no further cost reduction is possible; at that time, SCHEDXIG is invoked on the final (merged) dependency graph to determine the final XIG execution schedule. It is easy to see that the worst-case time complexity of MERGEXIG (or, our entire XIG optimization procedure) is $O(n^3|\mathcal{S}| \log n)$.

Next, we consider how to merge a pair of queries, namely, given a query pair (Q_1, Q_2), how to generate a single query Q to compute both Q_1 and Q_2. For a pair of queries that are not dependent on each other, the merged query can be simply expressed as:

<result> <q1>{Q_1}</q1> <q2>{Q_2}</q2> </result>

It is straightforward to separate and extract the results of Q_1 and Q_2 from the result of the merged query.

Now, consider a pair (Q_1, Q_2) where Q_2 uses the result of Q_1. In particular, consider a production $A \rightarrow \alpha$, and we want to merge the queries for computing Inh(A) (Q_1) and Inh(B) (Q_2), where B is in α. If Q_1 yields a sequence of values of Inh(A), we want the merged query Q to compute a corresponding sequence of Inh(B) values. We associate a "key" with each value of Inh(B) in order to determine the position of the B element in the target XML document. The key of an Inh(B) value is generated by concatenating the key of the corresponding Inh(A) value \$x and an id f_{sk}(\$x). Here f_{sk} is a Skolem function that, given a value, generates a unique id (see, *e.g.,* [22] for discussions on Skolem functions). Using the keys, the synthesized attribute Syn(A) can be computed by sort-merging the values of Syn(B) for all B in α *w.r.t.* key values.

For example, recall the rules associated with a production $A \rightarrow B_1, \ldots, B_n$ (similarly for other productions). Let query Q_1 compute Inh(A) and return either a single s-element

<s> <val> v </val> <key> k </key> </s>,

or a sequence of s-elements enclosed by a tag <seq>, where v is a value of Inh(A) and k is the key of v. Then, the merged query for computing $Inh(B_i)$ is (abusing XQuery syntax):

```
let $a := Q1 return
 {if $a/s
  then <s> <val>{Qi($a/s/val)}</val>
           <key>{($a/s/key, fsk($a/s/val))}</key> </s>
  else <seq> for $a' in $a/seq/s
             let $v := $a'/val
             let $k := $a'/key
             return <s> <val>{Qi($v)}</val>
                        <key>{($k, fsk($v))}</key> </s>
       </seq>}
```

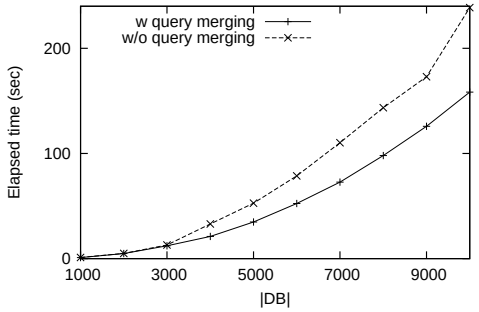


Figure 9: Effectiveness of query merging on XIG V_{dealer}

The query returns either a single pair (Inh(B), key) or a sequence of such pairs depending on the input Inh(A). (Note that the sibling and parent/child relations are captured by the keys.) This example shows how to merge a pair of queries with dependency on them.

It is worth mentioning that our query-merging strategy given above does not introduce any null values, in contrast to the out-union/outer-join approaches of [5, 4, 15, 10]. Also observe that merging is not carried out if it does not reduce the total cost of XIG evaluation.

7. EXPERIMENTAL EVALUATION

We next present preliminary results from an experimental evaluation of our XIG-based techniques. A prototype of our XIG-based middleware has been built on top of the Galax XQuery engine (db.bell-labs.com/galax) and Java RMI. The source databases are built based on the source DTD D^S_{dealer} and D^S_{dealer} by using the Toxgene data generator (www.cs.toronto.edu/tox/toxgene). The database size, $|DB|$, is given as the number of cars. A fraction f of the cars are on sale. For the recursive definition of feature in the sale data (recall D^S_{sale} from Fig. 1(c)), we generate 1 to 3 random features for each car and the depth of the recursion is limited to 2. The experiments were run on a distributed system connected by a local area Ethernet. Each site has a 2.4GHz Pentium 4 processor and 512M RAM. The cost of an XQuery query is estimated by pre-running the query in Galax. In future implementation of XQuery engines, APIs may be provided for query cost estimation. Unless otherwise stated, each experiment was run 5 times and the average is reported.

Query Merging. Figure 9 shows the impact of query merging on the performance of the XIG V_{dealer} for different database sizes. Since V_{dealer} only involves a single server, scheduling is not needed here. The results clearly indicate that the evaluation strategy with query merging outperforms the one without merging. The performance gain is about 30% for large databases. Note that the gain comes from reducing the number of Galax calls, as Galax does not support query optimization and thus merged queries are not optimized by Galax. The performance gain from query merging is expected to be further improved pending the availability of optimization in XQuery engines (to our knowledge, no stable XQuery engine supports all of our queries and optimizations).

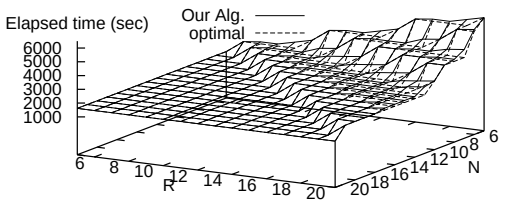


Figure 10: Scalability and benefits of query composability

Query Composability. The next set of experiments verifies the scalability and benefits of our XIG evaluation algorithm (namely,

Parameter	Meaning	Value
PathLen	Length of root-to-leaf path	[4, 8]
NoRoots	No. of root nodes	[1, 5]
ProbXIG	Probability of XIG-call nodes	0.5
FanIn	Node fan-in	1
FanOut	Node fan-out	[2, 4]
QueryCost	Local query cost (msec)	[10, 300]
XIGCost	Remote XIG cost (msec)	[100, 3000]
N	No. of servers	[2, 20]

Table 1: Settings for dependency graph generation

SCHEDXIG and MERGEXIG put together) with two workloads W_1 and W_2. To better demonstrate the impact of remote XIG calls, W_1 slightly extends the XIG $V(R, X)$ of Fig. 5 by adding a remote XIG which encodes the join in the rule for computing *Inh*(`car`), while W_2 further extends V_{dealer} in W_1 by adding an extra join on the car model. Both workloads were run on the distributed system. Figure 10 compares the evaluation time of W_2 obtained by using our evaluation algorithm with that of an optimal scheduling and merging strategy, which is computed manually as the XIGs involved are simple. In Fig. 10, $|DB|$ and f are fixed as 5000 and 10%, respectively. The number of servers N and the number of URIs R are varied from 5 to 20. The remote calls are uniformly distributed over the servers. The results show that our algorithm performs well; indeed, its performance nearly matches the optimal one. Furthermore, Fig. 10 indicates that our algorithm also scales well – its evaluation time decreases when the number N of servers increases, *i.e.*, it is roughly linear in $1/N$; moreover, the performance is better when the N/R ratio gets larger. The results of evaluating W_1 are similar.

Scheduling. To study workload sensitivity of our criticality-based *XIG-Scheduling* algorithm (denoted as *CRIT*), we compare its performance with two traditional scheduling algorithms *Shortest Task First (STF)* and *Longest Task First (LTF)* using randomly-generated XIG query-dependency graphs. Table 1 gives parameter settings for our random dependency-graph generator. Each node in a graph has a single parent (*i.e.*, fan-in of 1), whereas node fan-out is chosen uniformly between 2 and 4. The length of root-to-leaf path is chosen uniformly between 4 and 8. The probability of a node being an XIG call is 0.5, and its execution site is distributed uniformly among all servers. The ranges of costs for queries and XIG-calls are determined based on the response times obtained by using our prototype. Figure 11 depicts the performance of the algorithms. Each simulation was run 100 times to obtain sufficient confidence intervals of average elapsed times. The number of servers, N, varies from 2 to 20. Clearly, when N is small (*e.g.*, $N = 2$), scheduling is a non-issue and all algorithms perform similarly. However, as N increases, *CRIT* does better at exploiting parallelism, and it outperforms *STF* and *LTF* by more than 40% and 60%, respectively.

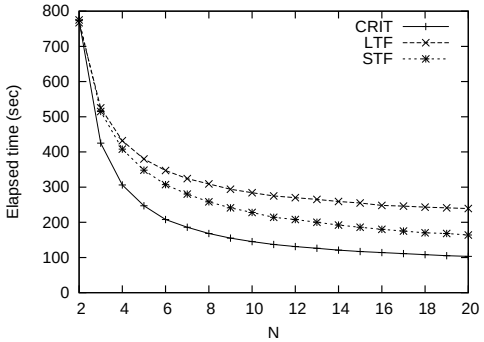


Figure 11: Effectiveness of XIG-Scheduling

These results show that our query-merging algorithm is effective for optimizing XIG evaluation, and that our XIG-scheduling technique significantly outperforms traditional scheduling algorithms.

8. CONCLUSION

We have proposed a novel language, XIGs, for specifying XML integration. XIGs automatically support conformance to a target DTD, and allow one to build a large, complex integration via composition of component XIGs. We have also developed novel optimization algorithms for evaluating XIGs. We are currently developing APIs to simplify XIG specifications, and exploring index structures for efficient evaluation. We also plan to identify practical XQuery fragments for XIG optimization and termination analyses.

9. REFERENCES

[1] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, 2003.

[2] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *PODS*, 2001.

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

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

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

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

[7] L. Bouganim, F. Fabret, C. Mohan, and P. Valduriez. Dynamic query scheduling in data integration systems. In *ICDE*, 2000.

[8] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic load balancing in hierarchical parallel database systems. In *VLDB*, 1996.

[9] 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/`.

[10] 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.

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

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

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

[14] R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *POPL*, 1992.

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

[16] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, and V. Vassalos. The TSIMMIS approach to mediation: Data models and languages. *J. Intelligent Information Systems (JIIS)*, 8(2):117–132, 1997.

[17] M. Garofalakis and Y. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, 1997.

[18] A. Goel, M. R. Henzinger, S. Plotkin, and E. Tardos. Scheduling data transfers in a network and the set scheduling problem. In *ACM STOC*, 1999.

[19] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, 1969.

[20] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *VLDB*, 1995.

[21] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[22] A. S. Kosky. *Transforming Databases with Recursive Data Structures*. PhD thesis, University of Pennsylvania, 1996.

[23] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging intensional XML data. In *SIGMOD*, 2003.

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

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

[26] J. Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. *VLDB Journal*, pages 302–314, 1999.

[27] S. D. Swierstra and H. Vogt. Higher order attribute grammars. *Attribute Grammars, Applications and Systems*, 1991.

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