PRATA: A System for XML Publishing, Integration and View Maintenance

Gao Cong Wenfei Fan* Xibei Jia **Shuai Ma**
University of Edinburgh
{gao.cong@,wenfei@inf.,x.jia@sms.,sma1@inf.}ed.ac.uk

Abstract

We present PRATA*, a system that supports the following in a uniform framework: (a)* XML *publishing,* i.e., *converting data from databases to an* XML *document, (b)* XML *integration,* i.e., *extracting data from multiple, distributed databases, and integrating the data into a single* XML *document, and (c) incremental maintenance of published or integrated* XML *data (view),* i.e., *in response to changes to the source databases, efficiently propagating the source changes to the* XML *view by computing the corresponding* XML *changes. A salient feature of the system is that publishing, integration and view maintenance are* schema-directed*: they are conducted strictly following a user-specified (possibly recursive and complex)* XML *schema, and guarantee that the generated or modified* XML *document conforms to the predefined schema. We discuss techniques underlying* PRATA *and report the current status of the system.*

1 Introduction

It is increasingly common that scientists want to integrate and publish their data in XML. As an example, consider how biologists exchange their data (the story is the same in other areas such as astronomy, earth sciences and neuroinformatics.) A biologist may manage a collection of experimental data by using some database management system (DBMS). In addition, information will be brought in from other databases and integrated with the basic data. To share information, the community of biologists interested in the topic get together and decide that there should be some standard format for data exchange, typically XML. In addition, they produce some XML DTD or schema to describe that format such that all members of the community exchange their data in XML with respect to the schema. Now one needs to handle the migration of data through various formats and to ensure that the resulting XML data conforms to the predefined schema. This is known as *schema-directed publishing/integration*. More specifically, XML publishing is to extract data from a traditional database, and construct an XML document that conforms to a *predefined* XML *schema*. XML integration is to extract data from *multiple, distributed* data sources, and construct an XML document (referred to as XML view) that conforms to a given schema.

With XML publishing/integration also comes the need for *maintaining* the published XML data (view). Biologists constantly update their databases. To propagate the source changes to the XML view, a naive approach might be to redo the publishing and integration from scratch. However, when the source data is large, the publishing and integration process may take hours or even days to complete. A better idea is by means of *incremental* XML *view maintenance*: instead of re-computing the entire XML view in response to source changes, only *changes to the* XML *view* are computed, which is often much smaller than the XML view and takes far less time to compute.

Schema-directed XML publishing and integration are, however, highly challenging. XML schemas are often complex, arbitrarily nested and even recursive, as commonly found in *e.g.,* biological ontologies [6]. This makes it hard to ensure that an XML view conforms to such a predefined schema. Add to this the difficulties introduced by XML constraints (*e.g.,* keys and foreign keys) which are often specified in a schema and have also to be satisfied by published and integrated XML data. These are further complicated by incremental XML view maintenance, which requires that changes to XML views should not violate the schema.

It is clear that new automated tools are needed. In particular, much in need is a uniform system to support XML publishing, integration and incremental view maintenance such that all these interact in the right way. However, no commercial systems are capable of supporting these. Indeed, while Microsoft SQL 2005 [9], Oracle XML DB [11] and IBM DB2 XML Extender [7] support XML publishing, they either ignore the schema-directed requirement (by using a fixed document template instead of an XML schema), or do not allow recursive XML schema. Worse still, none of these supports XML integration. When it comes to incremental XML view maintenance, to the best of our knowledge, no commercial DBMS provides the functionality. As for research prototypes, Clio [10] focuses on relational data integration based on schema mapping and does not address XML integration. While SilkRoute [5] and XPERANTO [12] are developed for XML publishing, they allow neither recursive XML schemas nor constraints. None of these systems supports XML integration or incremental view maintenance.

To this end we present PRATA, a system under development, that supports schema-directed XML publishing, integration and incremental view maintenance in a uniform framework. As depicted in Fig. 1, the system consists of three main components (modules): XML publishing, XML integration and incremental XML view maintenance.

To our knowledge, PRATA is the first and the only system that is capable of supporting all of these. It is worth

*Supported in part by EPSRC GR/S63205/01, GR/T27433/01 and BBSRC BB/D006473/1. Wenfei Fan is also affiliated to Bell Laboratories, Murray Hill, USA.

mentioning that while XML publishing is a special case of XML integration where there is a single source database, we treat it separately since it allows us to develop and leverage specific techniques and conduct the computation more efficiently than in the generic integration setting.

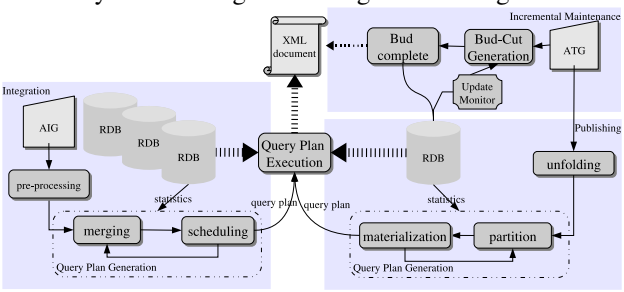

Figure 1. the System Architecture

2 XML Publishing

This module allows users to specify mappings from a relational database schema R to a predefined XML schema D, via a GUI and in a novel language *Attribute Translation Grammar* (ATG) that we proposed in [2]. Given an ATG σ and a database instance I of R, the system automatically generates an XML document (view) $\sigma(I)$ of I such that $\sigma(I)$ is guaranteed to conform to the given DTD D. This process is called *schema-directed publishing*.

Source relational schema R_0:

chapters(*chapter_id*, *name*)
receptors(*receptor_id*, *chapter_id*, *name*, *code*)
refs(*ref_id*, *chapter_id*, *year*, *title*)
cite(*ref_id*, *receptor_id*)

Figure 2. Relational schema R_0 for Receptors

We next demonstrate the ATG approach for publishing relational data in XML, by using a simplified example taken from the IUPHAR (International Union of Pharmacology) Receptor Database [8], which is a major on-line repository of characterisation data for receptors and drugs. We consider tables chapters, receptors, refs and cite shown in Fig. 2 (the primary keys for all tables are underlined). Table chapters stores the receptor families where each family has a *chapter_id* (primary key) and a *name*. Table receptors stores the *receptor_id*, *chapter_id*, *name* and *code* of receptors. Table refs stores publications on receptor families. Each reference has a *ref_id*, *title* and a *year* of the publication. It is associated with a unique receptor family through the attribute (foreign key) *chapter_id*. Finally, table cite records many-to-many relationships between receptors and references with *ref_id* and *receptor_id* (as foreign keys) pointing to references and receptors, respectively.

Target DTD D_0:

```
<!ELEMENT db          (family*)>
<!ELEMENT family*     (name, receptors, references )>
<!ELEMENT references (reference*)>
<!ELEMENT reference   (title, year)
<!ELEMENT receptors   (receptor*)>
<!ELEMENT receptor    (name, receptors)>
/* #PCDATA is omitted here.  */
```

Figure 3. DTD D_0 for publishing data of R_0

One wants to represent the relational Receptor data as an XML document that conforms to the DTD D_0 shown in Fig. 3. The document consists of a list of receptor families, identified by *chapter* ids. Each *family* collects the list of *receptor*s and the list of *ref*erences in the family (chapter). Note that *receptor* (thus D_0) is recursively defined: the *receptors* child of *receptor* element E_0 can also take an arbitrary number of *receptor* $E_1, E_2, \cdots, E_n$ as its children, which are the receptors related to E_0 through references. To avoid redundant information, if a *receptor* appears more than once in the reference hierarchy, the document stores it only once, *i.e.,* its first appearance.

The ATG σ_0 for publishing the Receptor data is given in Fig. 4, which extends the predefined DTD D_0 by associating semantic rules (*e.g.,* Q_1) to assign values to the variables (*e.g.,* $family). Given the Receptor Database, the ATG σ_0 is evaluated top-down: starting at the root element type of D_0, it evaluates the semantic rules associated with each element type encountered, and creates XML nodes following the DTD to construct the XML tree. The values of the variables $\$A$'s are used to control the construction. Now we give part of the ATG evaluation process.

(1) At each `receptors` element s, the target tree T is further expanded as follows. As $\$receptors.tag$ is assigned "0" from the family element's evaluation, the first branch of SQL query Q_2 is triggered. This branch finds the tuples of $receptor\_id$ and $name$ associated with each family f from the `receptors` relation, by using variable $\$receptors.id$ (passed from $\$family.chapter\_id$ in the last step) as a constant parameter. For each r of these tuples, a `receptor` child of s is created carrying r as the value of its variable.

(2) At each `receptor` element r, `name` element and `receptors` element s' are created (note that "1" is assigned to $\$receptors.tag$, and $\$receptor.receptor\_id$ is accumulated into $\$receptors.ids$ in this process).

(3) At each `receptors` element s', the second branch of SQL query Q_2 is triggered because $\$receptors.tag$ is now "1". This branch finds the tuples of $receptor\_id$ and $name$ from the `receptors` and `cite` relations related to receptor r that have *not* been processed in previous steps, by using variable $\$receptors.id$ and $\$receptors.ids$ as constant parameters. Note here that the variable $\$receptors.ids$ is used to decide whether or not a `receptor` has been processed earlier, and thus avoid redundant inclusion of the same `receptor` in the document T. For each r' of these tuples, a `receptor` child of s' is created carrying r' as the value of its variable, and the `receptor` element is in turn created as described in (2).

Details on the above conceptual level evaluation process and more effective techniques to generate efficient evaluation plans are show in [2]. These include query-partitioning and materializing intermediate results, in order to reduce communication cost between the publishing module and the underlying DBMS, based on estimates of the query cost and data size. We omit the details here due to the space con-

straint, but encourage the reader to consult [2].

Semantic Attributes: /*omitted*/
Semantic Rules:
db → **family***
Q_1: $family ← **select** chapter_id, name **from** chapters

family → **name**, **receptors**, **references**
$fname = ($family.name), $references = ($family.chapter_id),
$receptors = (0, $family.chapter_id, ∅)

receptors → **receptor***
Q_2: $receptor ← case $receptors.tag of
0: **select** receptor_id, name, $receptors.ids
from receptors
where chapter_id = $receptors.id
1: **select** a.receptor_id, a.name, $receptors.ids
from receptors a, cite b, cite c
where b.receptor_id = $receptors.id **and**
b.ref_id = c.ref_id **and**
b.receptor_id <> c.receptor_id **and**
a.receptor_id = c.receptor_id **and**
a.receptor_id **not in** $receptors.ids

receptor → **name**, **receptors**
$rname = ($receptor.name),
$receptors = (1, $receptor.receptor_id, $receptor.ids ∪ $receptor.receptor_id)

references → **reference***
Q_3: $reference ← **select** title, year
from refs
where chapter_id = $references.chapter_id

reference → **title**, **year**
$year = ($reference.year), $title = ($reference.title)

A → **S** /* A is one of *name, title, year* */
$S = ($A.val)

Figure 4. An example ATG σ_0

3 XML Integration

Extending the support for ATGs, this module provides a GUI for users to specify mappings from *multiple, distributed* databases to XML documents of a schema D, in our language *Attribute Integration Grammar* [1]. In addition, given an AIG and source databases, this module extracts data from the distributed sources and produces an XML document that is guaranteed to conform to the given DTD D and satisfy predefined XML contraints Σ.

As an example, suppose that the tables chapters and receptors in Fig. 2 are stored in a database DB1, while the tables refs and cite are stored in DB2. Now the IPUHAR department wants to generate a report about the receptors and corresponding publications. The report is required to conform to a fixed DTD D_1, which extends the D_0 of Fig. 3 as follows: elements `fid` and `rec_id` are added to the `family` production and `receptor` production as their first subelements respectively; and `ref_id` and `fid` are added to the `reference` production as its first two subelements (from table DB2 : refs we can see one `reference` can only relate itself to one `family`). In addition, for each `family`, the document is to collect all the `references` that are publications cited directly or indirectly by `receptors` in the `family`.

These introduce the following challenges. First, the integration requires multiple distributed data sources. As a result a single SQL query may access multiple data sources, referred to as *multi-source queries*. Second, receptors are defined recursively. Third, for each family, `references` can only be computed after the `receptors'` computation is completed since it is to collect all references in the `receptors` subtree, which is recursively defined and has an unbounded depth. In other words, this imposes a dependency between the `references` and the `receptors` subtrees. As a result the XML tree can not be simply computed as ATGs [1] by using a top-down method.

In addition, the XML report is also required to satisfy the flowing XML constraints:

ϕ_1: references(reference.ref_id → reference)
ϕ_2: db(reference.fid ⊆ family.fid)

Here ϕ_1 is a key constraint asserting that each subtree rooted at a `references` node, `ref_id` is a key of `reference` elements; and ϕ_2 is an inclusion constraint asserting that the families cited by references in the `db` must be presented in the `family` subtree of the `db`.

As remarked in Section 1, no commercial systems or research prototype can support schema-directed XML integration with respect to both DTD and XML constraints. The only effective technique for doing this is our *Attributes Integration Grammars* (AIGs) reported in [1], which is the underlying technique for the integration module of PRATA.

Based on multi-source query decomposition and constraints compilation PRATA is capable of integrating data from multiple sources and generating an XML document that is guaranteed to both conform to a given DTD and satisfy predefined XML constraints. PRATA also leverages a number of optimization techniques to generate efficient query evaluation plans [1]. In particular, it uses a cost-based scheduling algorithm, by taking into account dependency relations on subtrees, to maximize parallelism among underlying relational engines and to reduce response time. Due to the lack of space we omit the details here (see [1]).

4 Incremental XML Views Maintenance

This module maintains published XML viewsc $\sigma(I)$ based on our incremental computation techniques developed in [3]. In response to changes ΔI to the source database I, this module computes the XML changes ΔT to $\sigma(I)$ such that $\sigma(I \oplus \Delta I) = \Delta T \oplus \sigma(I)$,while minimizing unnecessary recomputation. The operator $\oplus$ denotes the application of these updates,

As remarked in Section 1, scientific databases keep being updated. Consider the XML view published by the ATG σ_0 of Fig. 4 from the the Receptor Database I specified by Fig. 2. Suppose that the relational Receptor database is updated by insertions $\Delta refs$ and $\Delta cite$ to the base relations $refs$ and $cite$ respectively. This entails that the new reference information must be augmented to the corresponding reference subtrees in XML views. Moreover, the insertions also increase the related receptors for some receptor nodes in the XML view and the augment may result in further expansion of the XML view. Given the recursive nature of σ_0,

this entails recursive computation and is obviously nontrivial.

The incremental algorithm in [3] is based on a notion of ΔATG. A ΔATG $\Delta\sigma$ is statically derived from an ATG σ by deducing and incrementalizing SQL queries for generating edges of XML views. The XML changes ΔT are computed by $\Delta\sigma$, and represented by of a pair of edge relations (E^+, E^-) , denoting the insertions(buds) and deletions(cuts). The whole process is divided into three phases: (1) a *bud-cut generation phase* that determines the impact of ΔI on *existing* parent-child(edge) relations in the old XML view T by evaluating a fixed number of incrementalized SQL queries; (2) a *bud completion phase* that iteratively computes newly inserted subtrees top-down by pushing SQL queries to the relational DBMS; and finally, (3) a *garbage collection phase* that removes the deleted subtrees.

The rational behind this is that the XML update ΔT is typically small and more efficient to compute than the entire new view $\sigma(I \oplus \Delta I)$. The key criterion for any incremental view maintenance algorithm is to precisely identify ΔT; in other words, it is to minimize recomputation that has been conducted when computing the old view T. Our incremental maintenance techniques make maximal use of XML sub-trees computed for the old view and thus minimize unnecessary recomputation. It should be pointed out that the techniques presented in this section are also applicable to XML views generated by AIG. We focused on ATG views just to simplify the discussion. Due to the limited space we omit the evaluation details here (see [3]).

5 PRATA: Features and Current Status

Taken together, PRATA has the following salient features, which are beyond what are offered by commercial tools or prototype systems developed thus far.

Schema conformance. PRATA is the first system that automatically guarantees that the published or integrated XML data conforms to predefined XML DTDs and schemas, even if the DTDs or schemas are complex and recursive.

Automatic validation of XML constraints. In a uniform framework for handling types specified by DTDs or schemas, PRATA also supports automatic checking of integrity constraints (keys, foreign keys) for XML.

Integration of multiple, distributed data sources. PRATA is capable of extracting data from multiple, distributed databases, and integrating the extracted data into a single XML document. A sophisticated scheduling algorithm allows PRATA to access the data sources efficiently in parallel.

Incremental updates. PRATA is the only system that supports incremental maintenance of *recursively defined* views, and is able to efficiently propagate changes from data sources to published/integrated XML data.

Novel evaluation and optimization techniques. Underlying PRATA are a variety of innovative techniques including algorithms and indexing structures for query merging [2], constraint compilation, multi-source query rewriting, query scheduling [1], and bud-cut incremental computation [3], which are not only important for XML publishing and integration, but are also useful in other applications such as multi-query evaluation and database view maintenance.

Friendly GUIs. PRATA offers several tools to facilitate users to specify publishing/integration mappings, browse XML views, analyze published or integrated XML data, and monitor changes to XML views, among other things.

The current implementation of PRATA fully supports (a) schema-directed XML publishing, and (b) incremental maintenance of XML views of a single source, based on the evaluation and optimization techniques discussed in previous sections. For XML schemas, it allows generic (possibly recursive and non-deterministic) DTDs, but has not yet implemented the support for XML constraints. A preliminary prototype of the system was demonstrated at a major database conference [4], and is deployed and evaluated at Lucent Technologies and European Bioinformatics Institute [4]. We are currently implementing (a) XML integration and (b) incremental maintenance of XML views of multiple sources. Pending the availability of resources, we expect to develop a full-fledged system in the near future.

References

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

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

[3] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, 2004.

[4] B. Choi, W. Fan, X. Jia, and A. Kasprzyk. A uniform system for publishing and maintaining XML. In *VLDB*, 2004. Demo.

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

[6] GO Consortium. Gene Ontology. `http://www.geneontology.org/`.

[7] IBM. DB2 XML Extender. *http://www-306.ibm.com/software/data/db2/extenders/xmlext/*.

[8] IUPHAR. Receptor Database. `http://www.iuphar-db.org`.

[9] Microsoft. XML support in Microsoft SQL erver 2005, December 2005. *http://msdn.microsoft.com/library/en-us/dnsql90/html/sql2k5xml.asp/*.

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

[11] Oracle. Oracle Database 10g Release 2 XML DB Technical Whitepaper. *http://www.oracle.com/technology/tech/xml/xmldb/index.html*.

[12] 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.