University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

January 2002

Querying XML with Mixed and Redundant Storage

Alin Deutsch
University of Pennsylvania

Val Tannen
University of Pennsylvania, val@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Deutsch, Alin and Tannen, Val, "Querying XML with Mixed and Redundant Storage" (2002). *Technical Reports (CIS)*. Paper 32.
http://repository.upenn.edu/cis_reports/32

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-02-01.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/32
For more information, please contact repository@pobox.upenn.edu.

Querying XML with Mixed and Redundant Storage

Abstract
This paper examines some of the issues that arise in the process of XML publishing of mixed-storage proprietary data. We argue that such data will reside typically in RDBMS's and/or LDAP, etc, augmented with a set of native XML documents. An additional challenge is to take advantage of redundancy in the storage schema, such as mixed materialized views that are stored for the purpose of enhancing performance.

We argue that such systems need to take into consideration mappings in both directions between the proprietary schema and the published schema. Thus, reformulating queries on the (published) XML schema into executable queries on the stored data will require the effect of both composition-with-views (as in SilkRoute and XPERANTO) and rewriting-with-views (as in the Information Manifold and Agora).

Using any of the simple encodings of relational data as XML, the mappings between schemas and the materialized views can be expressed in XQery, just like the queries on the published schema. For query reformulation we give an algorithm that uses logical assertions to capture formally the semantics of a large part of XQuery. We also give a completeness theorem for our reformulation algorithm. The algorithm was implemented in an XML query rewriting system and we present a suite of experiments that validate this technique.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-02-01.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/32

Querying XML with Mixed and Redundant Storage

Alin Deutsch and Val Tannen

Abstract

This paper examines some of the issues that arise in the process of XML publishing of mixed-storage proprietary data. We argue that such data will reside typically in RDBMS's and/or LDAP, etc, augmented with a set of native XML documents. An additional challenge is to take advantage of redundancy in the storage schema, such as mixed materialized views that are stored for the purpose of enhancing performance.

We argue that such systems need to take into consideration mappings in both directions between the proprietary schema and the published schema. Thus, reformulating queries on the (published) XML schema into executable queries on the stored data will require the effect of both composition-with-views (as in SilkRoute and XPERANTO) and rewriting-with-views (as in the Information Manifold and Agora).

Using any of the simple encodings of relational data as XML, the mappings between schemas and the materialized views can be expressed in XQuery, just like the queries on the published schema. For query reformulation we give an algorithm that uses logical assertions to capture formally the semantics of a large part of XQuery. We also give a completeness theorem for our reformulation algorithm. The algorithm was implemented in an XML query rewriting system and we present a suite of experiments that validate this technique.

1 Introduction

XML is widely accepted as the standard for data exchange between businesses on the Internet. Consequently, most corporations need to *publish* [1] as XML selected portions of their proprietary business data. This paper examines some of the tools that are needed to facilitate such a process.

Most business data is stored in relational DBMS's. More recently, business application have also begun producing native XML data. Moreover, as a consequence of the XML data exchange phenomenon itself, corporations are acquiring business data as native XML. If this acquired XML has a "nice" structure it might be easy and efficient to store it also in relational form. For general XML documents however, there is considerable debate on whether to store them into RDBMS or to use for them some native XML storage technique. We will summarize this debate in section 9 below. However, no clear winner has emerged so far. Therefore, we will assume that the proprietary data of a corporation consists of one or more RDBMS's augmented with a set of native XML documents. [2] We will show how to handle such **mixed** storage schemas in order to answer queries asked against a published XML schema.

A second issue is performance tuning. In an RDBMS an administrator will add indexes to speed up certain classes of queries. Most such systems can also speed up queries by rewriting them to use the views that a user chooses to materialize. Indexes and materialized views are forms of **redundancy** in storage. Redundancy is even more useful in mixed storage. For example, adding to the RDBMS some views of the natively stored XML can make a big difference.

We are led to consider a system for publishing (as XML) proprietary data residing in **M**ixed **A**nd **R**edundant **S**torage, or **MARS**. The technical focus of this paper is on **query reformulation**. These are queries on the

[1] We use the term "published" for all such exchanges, be they public domain, subscription lists, or deals with selected partners.

[2] Storing some of the data in other systems such LDAP is also plausible. We describe in section 8 how to extend our techniques.

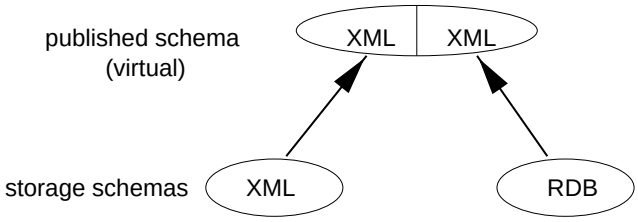

Figure 1: Simple MARS Configuration

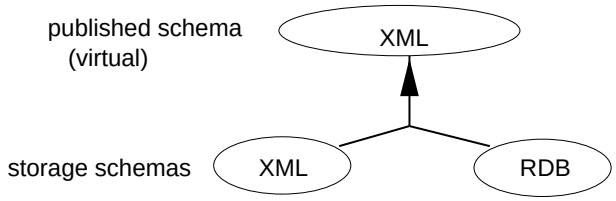

Figure 2: General MARS Configuration

virtual/published XML schema and they need to be reformulated into queries on the mixed storage schema, while taking advantage of storage redundancies.

We have designed a query reformulation algorithm that deals with complex storage and tuning requirements and we have implemented it in a system called MARS. To understand the problem we solve, we give a context (configuring and tuning MARS applications) and a motivating example involving musical data.

Configuring the MARS system First we need to specify the published XML schema, but open standards already exist for this task [28, 29]. Assuming that the schema of the stored/proprietary data is given, next we need to express the relationship between the proprietary data and the published data. Since in general *less* information is being published, such a relationship can be fully captured only by mappings from the proprietary schema to the published schema. In fact in the motivating example below we have a configuration that cannot be captured at all with mappings in the opposite direction, i.e., from the published schema to the proprietary schema [3] . Given the mixed storage, for configuring a MARS application we may need to specify:

1. Mappings from stored/proprietary RDB to virtual/published XML (as in Figure 1)
2. Mappings from stored/proprietary XML to virtual/published XML (also as in Figure 1)
3. More generally, mappings that integrate the stored XML and the stored RDB (as in Figure 2). This is more complicated, but sometimes inevitable: consider a mapping that performs a join between XML and RDB data but then we hide (project away) the join attributes.

The motivating example below corresponds to the simpler case in Figure 1 but our system deals with the general case in Figure 2.

Tuning the performance of the MARS system Indexes and materialized views add redundant data in hopes of speeding up queries. Indexing in RDBMS is well-understood. There is active research on XML indexing (eg., [19, 12]), but already it is clear that systems can make use of the XML analog of what was called *access support relations* in OODBMS [21]. These can be expressed as materialized RDB views of XML data. For performance tuning with mixed storage we may need to specify (and materialize):

1. Stored XML views of the stored XML or of the virtual XML. Such views might simply be queries previously asked. Caching the results of selected queries is a standard technique [4].
2. Stored RDB views of the same. This allows us to rewrite some queries that access both XML and RDB data into just RDB queries. Experience has shown that the "set-oriented" processing in RDB engines is generally better than the "graph-navigation" processing associated with XML [27]. [4]

[3] The same problem arises in information integration (global integration schema vs. local sources schema) and in physical data independence (logical schema vs. physical schema). We discuss it together with related work in section 9

[4] This begs the question: why not store it all in an RDBMS? But not all XML data is easily stored this way. See our discussion in section 9.

3. Stored XML views of the stored RDB. This comes up eg. when an XML *warehouse* is needed for various reasons [16]. If such a view was materialized, a MARS system might be able to take advantage of it, depending on the class of queries and on the quality of the native XML storage [12].

We give an example that features the first two kinds of materialized views.

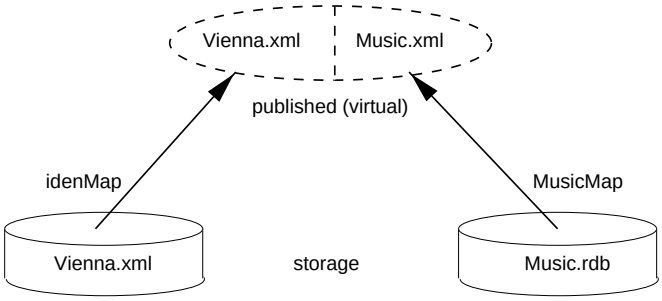

Figure 3: Motivating example configuration

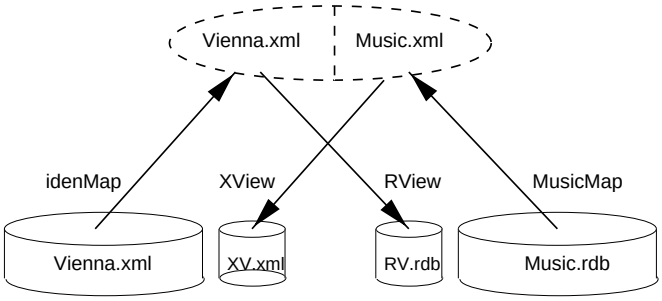

Figure 4: Motivating example after tuning

Motivating example: a MARS application. [5] Consider the proprietary Music.rdb relational database (Figure 5) that is published as Music.xml using a mapping MusicMap. Rather than giving a DTD or an XML Schema [29] for Music.xml we show in Figure 6 some pieces of the *virtual* XML published data that correspond to the example tuples shown in Music.rdb. The DTD or XML Schema can be readily figured out. We will show in due course how to express MusicMap but we want to point out here that the information in the `id` and `aid` attributes is *not* published. It is therefore impossible to capture this relationship through a mapping from published to proprietary.

The mixed storage of this example also includes a natively stored XML document Vienna.xml (see Figure 7). This part of the proprietary data is published in its entirety through the *identity* mapping idenMap and we use the same name Vienna.xml for the virtual published data. The entire configuration is shown in Figure 3.

Consider now querying the published XML. We formulate such a query in Figure 8 using the XQuery [30] language. Answering this query requires accessing both the RDB and the XML stored data. This may change if we tune the system by adding redundant materialized views as follows.

Can we do it all in the RDBMS? Although not shown, assume also that the `<works>` subelement of Vienna.xml is very unstructured, containing works described in various formats, annotations, reviews, anecdote, pictures, etc. It may be counterproductive the store this part of the document in an RDBMS (except perhaps as a LOB), but it makes a lot of sense to store the relationship between the nicely structured parts as a relation RV.rdb namely, the person name and the spouse information. Note that RV.rdb is *lossy* (looses information) not only because it omits the `<works>` elements, but also because it doesn't distinguish among composers, architects and writers. The relation RV.rdb does not *have* to be located in the same RDBMS as Music.rdb (see Figure 4) but many queries could benefit if both Music.rdb and RV.rdb are made available to the same relational optimizer. We will show below how to express the view RView that produces RV.rdb.

Can we do it all with XML? Imagine that we store an XML view XV.xml that is defined from the Music.xml part of the published schema (corresponding view expression XView is below). Then we might be able to answer some queries by accessing only XV.xml and Vienna.xml. The diagram of this MARS application after configuration and tuning is shown in Figure 4.

The query reformulation problem here is the following: given a query Q on Music.xml + Vienna.xml, reformulate Q into a query Q' on Music.rdb + RV.rdb + Vienna.xml + XV.xml such that Q and Q' are equivalent given the

[5] We ask the forgiveness of the reader for basing this example on a funny song by Tom Lehrer, about Alma Schindler, who was married to Gustav Mahler, Walter Gropius, and Franz Werfel.

definitions of MusicMap, idenMap, RView, and XView. If we had only the mappings MusicMap and idenMap to contend with, we could just *compose* Q with the mappings (as is done in [17, 9] for the case of just relational storage). If we had only the views RView and XView to deal with, we could use *rewriting-with-views* as in [22]. Having all four creates a completely new set of challenges. To continue the example we need to see the mappings and the views.

Expressing Views and Queries. What user-level language(s) should be used for configuring and tuning MARS? As query language expressions, a mapping, a view, and a query are the same thing. Since we use XQuery for the queries on the published XML, is XQuery enough? Clearly XML $\rightarrow$ XML mappings/views can be given in XQuery but for RDB $\rightarrow$ XML and XML $\rightarrow$ RDB we have an interesting choice. None of the two data models, relational or XML, is "included" in the other in the same manner in which, say, relations are a particular case of nested relations. However, each can be generically encoded in the other. Such encodings do not lose information and they come together with query translations that preserve them. They can also be easily decoded, which is essential, for example when a view gives RDB data encoded as XML, data that then must be stored in an RDBMS. Given such encodings, the mixed mappings/views can be actually expressed in the standard query languages of XML and RDB, namely XQuery and SQL, see Figure 11.

It turns out that encoding XML as RDB and the resulting SQL views are user-unfriendly. Specifically, what can be written as a short XPath expression corresponds in SQL to FROM and WHERE clauses that are just too large to be forced upon a human user [22]. Therefore, in MARS *at user level* we choose to encode RDB in XML, as [26] does [6]. There are several simple and friendly encodings of relations as semistructured data or XML [1, 7] and any of them can be selected by the MARS administrator. We chose one of these encodings and fixed it for this paper. We describe it here by example, by encoding some of the Music.rdb tuples, see Figure 12. With this encoding, we give MusicMap, RView, and XView in Figures 13, 9, and 10.

Motivating example (continued). Let Q be the query on published XML shown in Figure 8. Given the schema mappings and materialized views in our example, Q can be reformulated in many ways. Here are three interesting queries, obviously equivalent to Q in this application:

```
R1 = for    p in documents(Vienna.xml)//person, pn in p/name, s in p/spouse, sf in s/fN, sm in s/mN,
            u in documents(encode(Music.rdb)), a in u//author, m in u//maiden
     where  a/id/text() = m/aid/text() and a/first/text() = sf/text() and m/name/text() = sm/text()
     return <res>pn/text()</res>

R2 = for    v in documents(encode(RV.rdb))/RV
            u in documents(encode(Music.rdb)), a in u//author, m in u//maiden
     where  a/id/text() = m/aid/text() and a/first/text = v/sfn/text() and m/name/text() = v/smn/text()
     return <res>v/pn/text()</res>

R3 = for     p in documents(Vienna.xml)//person, pn in p/name, s in p/spouse, sf in s/fN, sm in s/mN,
             c in documents(XV.xml)/R/BY, cf in c/FN, cm in c/LN/maiden,
     where   cf/text()=sf/text() and cm/text()=sm/text()
     return  <res>pn/text()</res>
```

The query R_1 uses only the original mixed storage proprietary data, Music.rdb and Vienna.xml. R_2 uses the materialized view RV.rdb together with Music.rdb and therefore it can be executed as a purely relational query, while R_3 uses Vienna.xml and the materialized view XV.xml, both natively stored XML..

It is quite likely that R_1 executes slower than any of R_2, R_3. But each of these two can be faster than the other. This depends on whether RV.rdb is stored in the same RDBMS as Music.rdb, on how much flexibility we have in storing what might a be a "cached query", namely XV.xml, and also on current and future progress in efficient XML

[6] In our system we *do* make use of encoding XML as RDB, but only *internally*, see sections 4 and 5

storage techniques [12]. Using cost information and heuristics, an optimizer will pick one of these queries toward an execution plan.

In a nutshell, our reformulation algorithm must be powerful enough to find R_1, R_2, R_2 and perhaps other queries, starting just from Q and the mappings and views defined in this MARS application. In general we have a *space of reformulated queries* that needs to be explored. The kind of mappings and views we consider here cannot be handled *together* by existing approaches [13, 17, 9, 22]. Different ideas are needed.

Our approach to query reformulation: compile mappings/views to constraints, use them for query rewriting. Because the mappings/views go in both directions between the stored and published schemas a good way of putting it all in the same framework is to follow the idea in [14] of capturing views with logical assertions (aka constraints), which are largely "direction-neutral".

However, [14] only shows how certain relational and OO views can be captured by a certain class of constraints. Those specific constraints can then be used to rewrite queries, following a classical technique called the *chase* [6]. Doing the same for mappings/views expressed in XQuery is challenging, for reasons including the following:

1. It is not clear, a priori, *who the constraints are.*

2. *XQueries contain descendant navigation steps* (transitive closure of child navigation steps), which cannot be expressed in FO logic, in particular in relational conjunctive queries and with constraints.

3. *XQueries are interpreted over trees*, while conjunctive queries are interpreted over arbitrary relations. Consequently, even if we encode XML using some form of an edge relation, classical rewriting algorithms will miss even the most obvious rewritings, because they hold only on instances in which the edge relation corresponds to a tree, and not on arbitrary interpretations of this relation.

4. *XQueries are nested.* Their `return` clause may contain subqueries, correlated to them by using variables bound in the outer `for` clause.

5. *XQueries create new XML elements*, as opposed to conjunctive queries who only return values appearing in the input.

6. *XQueries return deep, recursive copies of XML subtrees* from the input, as opposed to conjunctive queries who only copy values of base type. These copies are trees with distinct nodes, but which are indistiguishable from the source nodes by the data they carry as attributes or by their children.

7. XQueries have list semantics, as opposed to conjunctive queries, who have set semantics.

Internally: a relational encoding of XML. We overcome the obstacles we have just listed by using several technical ideas (section 4) that share the same framework, based on capturing everything with *relational* constraints. However, these relational constraints are written over a special schema which is a generic encoding of XML (see section 3). The starting point in fact is a set of generic constraints that captures some (not all, but just enough for many purposes) of the semantics of XML data and navigation.

Rest of the paper: An overview of the MARS system is given in section 2. Section 3 describes the internal representation of constraints and queries. The compilation of mappings/views into their internal representation as sets of relational constraints is described in section 4. Our reformulation algorithm which uses chase-rewriting is given in section 5. In section 6 we present a theoretical completeness result for our technique. In section 7 we report on experiments with the MARS implementation of our reformulation algorithm. Some extensions (section 8), a survey of related work (section 9), and conclusions and plans for furter work complete the paper.

Songs, their titles, their authors, with names; not all authors have a maiden name.

Conceptual schema: `song(title, aid)` `author(id, last, first)` `maiden(aid, name)`

Example tuples:

```
song(title="Kennst Du meine Naechte?", aid=1)    author(id=1, last="Mahler-Gropius-Werfel", first="Alma")
song(title="Alma", aid=2)                         author(id=2, last="Lehrer", first="Tom")
                            maiden(aid=1, name="Schindler")
```

Figure 5: Music.rdb (proprietary, stored in RDBMS)

```
<xdb>
 <lied>
   <title>Kennst Du meine Naechte?</title>
   <composer>
     <first>Alma</first>
     <last>Mahler-Gropius-Werfel<\last>
           <maiden>Schindler</maiden></last></composer></lied>
 <lied>
   <title>Alma</title>
   <composer>
     <first>Tom</first>
     <last>Lehrer</last></composer></lied>
</xdb>
```

The relationships song–author and author–maiden name are captured here just by the nesting and presence/absence of elements.

Figure 6: Example pieces of Music.xml (published, virtual)

Creative individuals from turn-of-the-century Vienna.

```
<vienna>
 <composers>
  <person>
   <name>Gustav Mahler</name>
   <spouse>
      <fN>Alma</fN>
      <mN>Schindler</mN></spouse>
   <works>...</works></person>
  <person><name>Alma Schindler...
 </composers>
 <architects>
  <person>
   <name>Walter Gropius</name>
   <spouse>
      <fN>Alma</fN>
      <mN>Schindler</mN></spouse>
   <works>...</works></person>
 </architects>
 <writers>
  <person>
   <name>Franz Werfel</name>
   <spouse>
      <fN>Alma</fN>
      <mN>Schindler</mN></spouse>
   <works>...</works></person>
 </writers>
</vienna>
```

Figure 7: A fragment of Vienna.xml (proprietary, stored as native XML)

"Return the names of Viennese individuals who were married to a lied composer":

```
for    p in documents(Vienna.xml)//person, pn in p/name, s in p/spouse, sf in s/fN, sm in s/mN,
       c in documents(Music.xml)//lied/composer, cf in c/first, cm in c//maiden
where  cf/text()=sf/text() and cm/text()=sm/text()
return <res>pn/text()</res>
```

Figure 8: A query against the published schema

The conceptual schema of RV.rdb*:* `RV(pn, sfn, smn)` *...*

RView *below produces* RV.rdb *encoded in XML (using the encoding we chose for this paper, see also Figure 12).*

```
<rvrdb>
 for    p in documents(Vienna.xml)//person,
        pn in p/name, s in p/spouse,
        sfn in s/fN, smn in s/mN
 return
     <RV><pn>pn/text()</pn>
         <sfn>sfn/text()</sfn>
         <smn>smn/text()</smn></RV>
</rvrdb>
```

Figure 9: RView produces RV.rdb (redundant, stored in RDBMS)

...`pn` *is person name,* `sfn,smn` *are spouse's first and maiden name.*

XView *below produces* XV.xml *as an XML document.*

```
<xvxdb>
 for    ld in documents(Music.xml)//lied,
        c in ld/composer,
 return <R><T>  ld/title </T>
           <BY> <FN>c/first</FN>
                <LN>c/last</LN></BY></R>
</xvxdb>
```

Figure 10: XView produces XV.xml (redundant, stored native XML)

```
      Encode                  XQuery
RDB  >>>>>>>>>  RDB as XML < ------ > XML

      Encode                    SQL
XML  >>>>>>>>>  XML as RDB < ------ > RDB
```

Figure 11: Encodings across models

```
<rdb>
 <song><title>Naechte</title><aid>1</aid></song>
 <song><title>Alma</title><aid>2</aid></row></song>
 <author><id>1</id>
        <last>Mahler</last><first>Alma</first></author>
 <author><id>2</id>
         <last>Lehrer</last><first>Tom</first></author>
</rdb>
```

Figure 12: Music.rdb encoded as XML

```
let  $music := documents(encode(Music.rdb))
<xdb>
 for    $song  in  $music//song
 return
  <lied>
    {$song/title}
    <composer>
     for    $author  in  $music//author
     where  $author/id/data() = $song/aid/data()
     return
      {$author/first}
      <last>
        {$author/last}
        for    $maiden  in $music//maiden
        where  $maiden/aid/data() = $author/id/data()
        return
          <maiden>$maiden/name</maiden>
      </last>
    </composer>
  </lied>
</xdb>
```

Figure 13: MusicMap

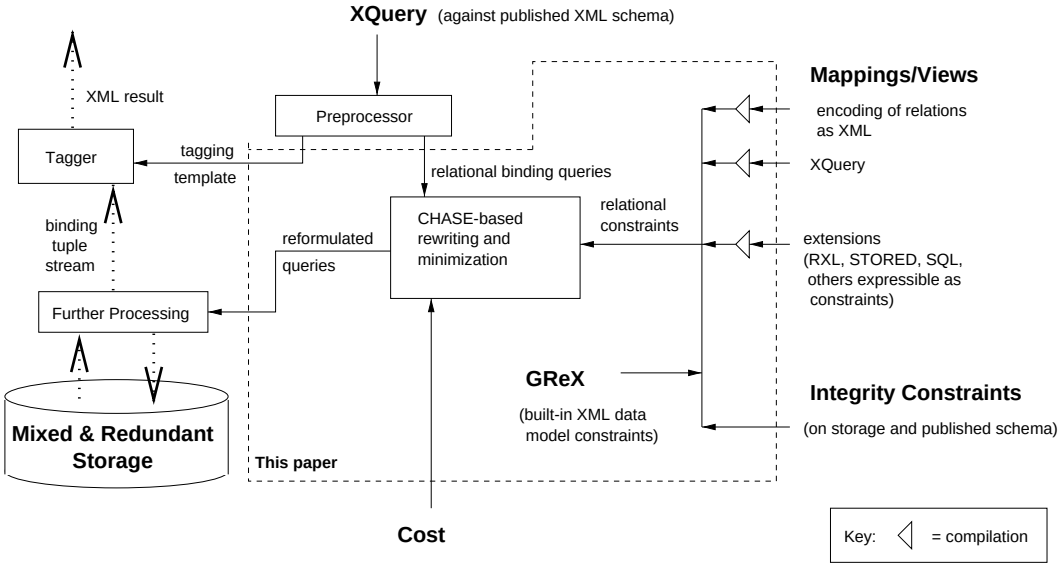

Figure 14: MARS Architecture

2 MARS Architecture

The architecture of the MARS systems is shown in Figure 14. Its input are XQueries against the published XML schema, the set of mappings and views used for configuration and tuning and integrity constraints known to hold on the published and storage schemas. It also features a plug into which a cost module can be inserted. Any input XQuery is fed to a preprocessor, which, after simple "normalization" transformations breaks it into one or more **binding queries** and a **tagging template**. We explain each of them.

Binding Queries. There is a binding query for each `for` clause appearing in the (normalized) query. The output of a binding query is a relation, whose attributes are exactly the variables bound in the `for` clause (the `for` variables). Each tuple in the output of a binding query corresponds to an assignment of values to the `for` variables (i.e., a *binding*).

Internally, the binding queries are translated to relational form. This is done following a generic encoding of XML and XQuery into the relational model. We call this encoding GReX (see section 3). GReX consists of a relational schema and a set of constraints. The schema contains such relations as `child`,`desc`,`tag`, intended to capture navigation in the XML data model (e.g. the fact that an element is a child or descendant of another, that it has a certain tag, etc.). The constraints are used to capture this intended meaning (for instance every element has at most one parent and tag, `desc` contains the transitive closure of `child` etc.).

Example. Consider the query Q from Figure 8. After normalization and GReX encoding, MARS produces a binding query, shown here as a relational conjunctive query:

$$
\begin{aligned}
B_Q(p, pn, s, sf, sm, c, cf, cm) \leftarrow\ & \texttt{viennaRoot}(r), \texttt{desc}(r, d), \texttt{child}(d, p), \texttt{tag}(p, ''person''), \\
& \texttt{child}(p, pn), \texttt{tag}(pn, ''name''), \texttt{text}(pn, pnt), \texttt{child}(p, s), \texttt{tag}(s, ''spouse''), \\
& \texttt{child}(s, sf), \texttt{tag}(sf, ''first''), \texttt{text}(sf, sft), \texttt{child}(s, sm), \texttt{tag}(sm, ''maiden''), \texttt{text}(sm, smt), \\
& \texttt{musicRoot}(r'), \texttt{desc}(r, d'), \texttt{child}(d', l), \texttt{tag}(l, ''lied''), \texttt{child}(l, c), \texttt{tag}(c, ''composer''), \\
& \texttt{child}(c, cfn), \texttt{tag}(cfn, ''first''), \texttt{text}(cfn, sft), \texttt{child}(c, cmn), \texttt{tag}(cmn, ''maiden''), \texttt{text}(cmn, smt)
\end{aligned}
$$

Compilation, Rewriting under constraints. Binding queries are still expressed against the published, virtual,

XML schema so are not directly executable. They are then reformulated by the chase rewriting and minimization module. This is the central component of MARS and it proceeds by chase-rewriting relational queries with relational constraints all of them expressed over the relational schema GReX. For the reformulation to take into account the schema mappings, materialized views and possibly integrity constraints which are specified as part of the configuration and tuning process, all of these must be *compiled* to relational constraints over GReX (as shown in section 4).

Due to the redundancy, any binding query has in general several equivalent reformulations, some potentially cheaper to execute than others. When exploring the search space for reformulations (see section 5), the cost module is used to find the optimal one. In the absence of reasonable cost information, the system provides a default heuristic cost which seems to perform well.

Processing reformulations. The tagger. The obtained reformulations of the binding queries undergo further processing (such as translating the queries sent to the individual stored sources to the language these sources speak, be it XQuery, SQL, LDAP- or DOM-based programs). Each stored source will return a stream of tuples of variable bindings,. These streams are combined and turned into XML by a component called **tagger** that uses the tagging template. If the amount of data returned by the binding queries is large, tagging can become a bottleneck. This issue is orthogonal to the reformulation issue that we investigate in this paper. Ample research has been done on it [17, 9], and in MARS we have adopted the *late tagging, sorted outer union* approach of [9].

Focus of this paper. As shown in the architecture diagram (dotted rectangle in Figure 14), in this paper, we focus on the compilation of mappings and views to constraints (section 4), and on the reformulation against the storage schema of the binding queries using these constraints (section 5).

3 The MARS Internal Representation

The internal representation of XML data and XML navigation is a **G**eneric **Re**lational representation of **X**ML and so we call it **GReX**. GReX consists of the relational schema (root, el, child, desc, tag, attr, id, text) and of a set of relational constraints outlined below. [7] The "intended" meaning of the relational symbols in GReX is the following. The unary predicate root denotes the root of the XML document, and the unary relation el is the set of its elements. child and desc are subsets of el $\times$ el and they say that their second component is a child, respectively a descendant of the first component. tag $\subseteq$ el $\times$ *string* associates the tag in the second component to the element in the first. attr $\subseteq$ el $\times$ *string* $\times$ *string* gives the element, attribute name and attribute value in its first, second, respectively third component. id $\subseteq$ *string* $\times$ el associates the element in the second component to a string attribute in the first that uniquely identifies it (if DTD-specified ID-type attributes exist, their values can be used for this). text $\subseteq$ el $\times$ *string* associates to the element in its first component the string in its second component. Some (but

[7] When we have several documents, we have several roots. In the examples in this paper, we use (less generic) root names that identify the document.

not all!) of this intended meaning is captured by the following set GReX of first-order relational constraints

(base) $\forall x, y\ [\ \texttt{child}(x,y) \rightarrow \texttt{desc}(x,y)\ ]$

(trans) $\forall x, y, z\ [\ \texttt{desc}(x,y) \wedge \texttt{desc}(y,z) \rightarrow \texttt{desc}(x,z)\ ]$

(refl) $\forall x\ [\ \texttt{el}(x) \rightarrow \texttt{desc}(x,x)\ ]$

(el_c) $\forall x, y\ [\ \texttt{child}(x,y) \rightarrow \texttt{el}(x) \wedge \texttt{el}(y)\ ]$

(el_d) $\forall x, y\ [\ \texttt{desc}(x,y) \rightarrow \texttt{el}(x) \wedge \texttt{el}(y)\ ]$

(el_{id}) $\forall s, x\ [\ \texttt{id}(s,x) \rightarrow \texttt{el}(x)\ ]$

(el_r) $\forall x\ [\ \texttt{root}(x) \rightarrow \texttt{el}(x)]$

(line) $\forall x, y, u\ [\ \texttt{desc}(x,u) \wedge \texttt{desc}(y,u) \rightarrow x = y \vee \texttt{desc}(x,y) \vee \texttt{desc}(y,x)]$

(oneTag) $\forall x, t_1, t_2\ [\ \texttt{tag}(x,t_1) \wedge \texttt{tag}(x,t_2) \rightarrow t_1 = t_2\ ]$

(id) $\forall s, e_1, e_2\ [\ \texttt{id}(s,e_1) \wedge \texttt{id}(s,e_2) \rightarrow e_1 = e_2\ ]$

(noLoop) $\forall x, y\ [\ \texttt{desc}(x,y) \wedge \texttt{desc}(y,x) \rightarrow x = y\ ]$

(oneParent) $\forall x, y, z\ [\ \texttt{child}(x,z) \wedge \texttt{child}(y,z) \rightarrow x = y\ ]$

(noShare) $\forall x, y, u, v\ [\ \texttt{child}(x,u) \wedge \texttt{child}(x,v) \wedge \texttt{desc}(u,y) \wedge \texttt{desc}(v,y) \rightarrow u = v\ ]$

(oneRoot) $\forall x, y\ [\ \texttt{desc}(x,y) \wedge \texttt{root}(y) \rightarrow \texttt{root}(x)]$

Observe that `(base)`, `(trans)`, `(refl)` above only guarantee that `desc` *contains* its intended interpretation, namely the reflexive, transitive closure of the `child` relation. There are many models satisfying these constraints, in which `desc` is interpreted as a proper superset of its intended interpretation, and it is well-known that we have no way of ruling them out using first-order constraints, because transitive closure is not first-order definable. The fact that we can nevertheless use the constraints in GReX and classical relational (therefore first-order) techniques for deciding containment under constraints comes therefore as a pleasant surprise.

Note that except for (*line*), all constraints in GReX are *embedded dependencies* (as [2] calls them, but also known as tuple- and equality-generating dependencies [6]) for which a deep and rich theory has been developed. (*line*) contains disjunction but so do the XPath expressions, implicitly, via the | operator. Extending the theory to *disjunctive embedded dependencies* like (*line*) is fairly straightforward [6, 15].

4 Compiling Mapping/View Expressions to Constraints

Schema mappings and materialized views play different conceptual roles in MARS but the problem of their compilation is the same, since both are XQuery expressions. Instead of a general formulation, we illustrate our compilation procedure on an example. We take for this the mapping MusicMap from Music.rdb to Music.xml and we give some of the GReX constraints that capture it. This mapping is really a composition of the generic encoding of RDB into XML (Figure 12) and the actual XQuery in Figure 13, call this XQuery MusicMap. We first show how to capture the encoding, then we express MusicMap with constraints, showing how we address the challenges listed in the introduction.

4.1 Compiling Generic XML Encodings of Relations

These encodings relate every relational tuple with a distinct XML element. (see Figure 12. This element is uniquely determined by the values of the tuple's attributes. Therefore, the encoding is really a function from tuples to elements, which is also injective (distinct tuples are encoded as distinct elements).

In our example, there is a unique `rdb`-element e_r which is a child of the root. Every tuple $song(t, aid)$ is encoded as a `song`-element e_s, which is a child of e_r (author and maiden tuples are similar). e_s in turn has a `title`-subelement e_t and an `aid`-subelement e_a, which contain as text a copy of the corresponding `title`, respectively `aid` attributes. We can model this encoding in two steps: first we state the existence of the unique element e_r as well as of the elements e_s, e_t, e_a for every `song` tuple, as functions of its attribute values. Then we specify the tag names of these elements, and the parent-child relationship between them. [8]

[8] In formal XQuery data model terminology, in the first step we actually specify the existence of element *nodes*, whose tag information

```
                                        root                                root
                                                                              |
                                       Fr()=er                           tag(er)=''rdb''
                        step1                                  step2          |
song(''Alma'', ''2'') ----->         Fs(Alma,2)=es            ----->     tag(es)=''song''
                                                                        /                \
                                 Ft(Alma,2)=et   Fa(Alma,2)=ea         tag(et)=''title''  tag(ea)=''aid''
                                                                          |                  |
                                                                        text(''Alma'')     text(''2'')
```

Step 1: Elements as functions of tuples. Let's call F_s, F_t, F_a the functions taking as argument a `song` tuple and returning an XML element: $F_s(t, aid) = e_s, F_t(t, aid) = e_t, F_a(t, aid) = e_a$. The unique element e_r is independent of any tuple and can be modeled as the image of a function of no arguments: $F_r() = e_r$. We call F_s, F_t, F_a, F_r *Skolem functions*, using the terminology from SilkRoute's mapping language, RXL [17] for similar functions that take as argument relational data and create new XML nodes.

In order to capture these functions with constraints, we extend the schema with the relations G_s, G_t, G_a, G_r storing the associated function graphs: $F_K(t, aid) = e_K$ if and only if $G_K(t, aid, e_K)$ where K is either of s, t, a, and $F_r() = e_r$ if and only if $G_r(e_r)$. For these relations to represent the graphs of the intended Skolem functions, they must satisfy the following constraints: (i) the first two components of each G_K must functionally determine the third (in other words, the relations are functional), (ii) their projection on the first two components –corresponding to the domain of the Skolem functions– must coincide with the set of `song` tuples, (iii) the third component must functionally determine the first two (the Skolem functions are injective, as they encode distinct tuples as distinct elements). This is how we express these constraints (where G stands for G_K and F for F_K, for any K): [9]

$$\forall t \forall a \forall e_1 \forall e_2\ [G(t, a, e_1) \wedge G(t, a, e_2) \rightarrow e_1 = e_2] \quad G\text{'s third component is a function } F \text{ of its first two} \tag{1}$$

$$\forall t \forall a\ [\texttt{song}(t, a) \rightarrow \exists n\ G(t, a, n)] \quad \texttt{song} \text{ is included in the domain of } F \tag{2}$$

$$\forall t \forall a \forall n\ [G(t, a, n) \rightarrow \texttt{song}(t, a)] \quad \text{the domain of } F \text{ is included in } \texttt{song} \tag{3}$$

$$\forall t_1 \forall a_1 \forall t_2 \forall a_2 \forall n\ [G(t_1, a_1, n) \wedge G(t_2, a_2, n) \rightarrow t_1 = t_2 \wedge a_1 = a_2] \quad F \text{ is injective} \tag{4}$$

The coresponding constraints for G_r are particular cases of the above, in which there are no arguments t, a since F_r takes the empty tuple as argument. It is easy to see that the constraints corresponding to (1) and (3) above are trivially satisfied in this case, and we are left with the analogons to (2) and (4), of which we show the latter:

$$\forall e_1 \forall e_2\ [G_r(e_1) \wedge G_r(e_2) \rightarrow e_1 = e_2] \quad F_r \text{ is injective } (e_r \text{ is unique}) \tag{5}$$

Step 2: Adding tagged tree structure. We still have to specify how the elements are organized in the XML tagged tree structure. For example, e_r is tagged `rdb`, e_s is tagged `song` (shown below), etc.:

$$\forall t \forall a \forall n\ [G_s(t, a, n) \rightarrow \texttt{tag}(n, ''song'')] \tag{6}$$

Moreover, the unique e_r is a child of the root and every e_s is a subelement of e_r. Every e_t is a subelement of the e_s related *to the same tuple*, and has a text child whose value corresponds to the `title` attribute of this tuple. We illustrate the constraint capturing the latter statement, as the more complex one. The treatment of e_a is similar.

and parent-child relationship are given in the second step.

[9] Observe that if we eliminate the existential quantifier in (2) above by bringing the statement in Skolem normal form, n is expressed as a function $F(t, a)$, which explains the naming choice.

$$\forall t \forall a \forall e_s \forall e_t [G_s(t, a, e_s) \wedge G_t(t, a, e_t) \rightarrow \texttt{child}(e_s, e_t) \wedge \texttt{text}(e_t, t)] \quad (7)$$

Finally, we must state that the encoding contains only information from our music database, and nothing more. That is, any `rdb, song, title` etc. element, appearing anywhere in the document is related to some tuple via a Skolem function. Moreover, its parent is determined by this tuple: the root is the parent of e_r, e_r that of e_s, and e_s that of the e_t related to *the same tuple* as e_s. Similarly for e_a. We illustrate for e_t:

$$\forall r \forall d \forall c\ [\texttt{root}(r) \wedge \texttt{desc}(r, d) \wedge \texttt{child}(d, c) \wedge \texttt{tag}(c, '' title'') \rightarrow \exists t \exists a\ G_s(t, a, d) \wedge G_t(t, a, c)] \quad (8)$$

Let us denote by GReXMusic the set of constraints obtained from the compilation of the encoding as shown above. Notice that all these constraints are disjunctive embedded dependencies. The size of GReXMusic is easily seen to be linear in the size of the relational schema Music.rdb.

4.2 Compiling mappings/views specified in XQuery

It is known from [14] how a view `v` defined by a conjunctive query *cq* can be captured with constraints. The idea is to write two inclusion constraints between the extent of `v` and the result of *cq* (one for each inclusion). However, XQuery features essential ingredients that go beyond the expressiveness of relational conjunctive queries, and capturing them with constraints is challenging. We illustrate on MusicMap given in Figure 13.

According to the XQuery standard, MusicMap executes in two stages. In the first, which we will call the *binding* stage, the `for` clause is evaluated, returning the **list of bindings** for the variable *song* (introduced by the $ sign), ordered in the document order. *song* is bound to every `song`-element reachable by navigating to a descendant element of the root (note the use of `//` for **descendant navigation**), and from there to a child `song`-element. In the second stage, which we call the *tagging* stage, a fresh `lied` **element is created** and output for every binding obtained in the first stage. This element is a child of the new document's root. Its contents is given by the result of several **nested, correlated queries**, whose execution follows in turn the binding and tagging steps. For example, the `$song/title` subexpression is shorthand for a query returning **deep, recursive copies** of all child `title`-elements of the element to which *song* was bound. Also, the contents of the `composer`-element is explicitly defined by a nested query. We have highlighted in bold font the ingredients going beyond conjunctive queries, and we show below how we capture them with constraints.

Obstacle: list semantics for variable bindings.

Partial Solution: reduce lists to ordered sets, disregard order.

Our approach, based on compilation into relational constraints does not fully capture document order. But what are the limitations implied by this? Notice that reformulating queries that order their result according to the values in the input (thus overriding the original document order) is an unrelated, easy problem. The hard issue is that of deciding whether a certain reformulation *preserves* the ordering in the result of the original query when the latter performs no ordering of its own, using instead the document order. In scenarios involving no native XML documents, but instead publishing relational/OO/LDAP data, there is no document order to begin with, so this becomes a non-issue. For the case of native XML sources, this problem is still an open research issue, which we do not address in this paper.

Even if we disregard order in the list of variable bindings, we are not yet in a conjunctive query context, because now the list becomes a bag. However, this bag does not contain duplicates if all variables are bound to elements (two distinct entries in the list of bindings for an element-typed variable may be isomorphic, but may not have the

same identity) [30]. In other words, the list is really an ordered set in this case. This is the case for the outer `for` clause of MusicMap in our example. The same observation holds if variables of non-element type, i.e. bound to an attribute, text or tag, are explicitly bound using the `distinct` keyword. Otherwise , the following simple trick will help us reduce the bag of bindings to a set: add to the `for` clause a variable that binds to the element owning the attribute/tag/text (if not already there). For instance, the clause `for //lied//maiden/text() $mt` in Q from our example is equivalently rewritten to `for //lied//maiden $m, $m/text() $mt`.

This trick is not needed in the case of MusicMap, and we can therefore define the set B_O of the variable bindings in its outermost `for` clause by the simple conjunctive query corresponding to the translation against the schema of GReX:

$$B_O(s) \leftarrow \texttt{root}(r), \texttt{desc}(r,d), \texttt{child}(d,s), \texttt{tag}(s,''\,song'')$$

We extend the schema with an element B_O, and specify the new relation's extent with two inclusion constraints; one says that it includes the result of the defining query, the other one gives the inverse inclusion:

$$\forall r \forall d \forall s\ [\texttt{root}(r) \wedge \texttt{desc}(r,d) \wedge \texttt{child}(d,s) \wedge \texttt{tag}(s,''\,lied'') \rightarrow B_O(s)] \quad \text{every tuple of variable bindings is in } B_O \tag{9}$$

$$\forall s\ [B_O(s) \rightarrow \exists r \exists d\ \texttt{root}(r) \wedge \texttt{desc}(r,d) \wedge \texttt{child}(d,s) \wedge \texttt{tag}(s,''\,lied'')] \quad \text{every tuple in } B_O \text{ is a variable binding} \tag{10}$$

Obstacles: interpretation over tagged trees, and descendant navigation
Solution: add constraints of GReX

Notice the `child`, `tag` and `desc` atoms in B_O's definition, as well as in the corresponding constraints. Of course, we do not interpret these symbols over arbitrary relations, but rather we want to capture the intended meaning of navigation in the XML tree, to a desendant of the root, and from there to a child with tag "lied". Recall from section 3 that part of the intended meaning is captured using the constraints in GReX (saying for instance that there are no cycles in the graph, that each element has precisely one parent, that the descendant relationship is transitive, reflexive, etc.) Therefore, when performing the reformulation of a query with respect to MusicMap we will do so under the constraints describing MusicMap *and* the constraints in GReX.

Obstacle: nested queries in `return` clause.
Solution: specify sets of variable bindings separately, using decorrelation.

In the second, *tagging* stage of MusicMap's execution, the list of bindings is scanned and for every `song`-element bound to *song*, a distinct `lied`-element is output. The `$song/title` expression nested within the `lied` tag is shorthand for a nested query iterating over all `title`-subelements of the `song` element, and returning copies thereof:

Q_{lied}($song) = `for $t in $song/title return $t`.

Let's call this query Q_{lied}, and observe that it is correlated with the outer query part via the *song* variable (a fact denoted by its appearance in parentheses). We show below how we express Q_{lied} with constraints. All other nested queries are captured similarly.

Although here we are interested in capturing the *semantics* of XQueries containing nested subqueries, we will do so by borrowing an idea developed for their *evaluation*. A naive evaluation strategy suggested by the above semantics specification would execute as many calls of Q_{lied} as there are distinct bindings for *song*. However, classical work on optimization of nested SQL queries suggests an alternative strategy, based on decorrelation (see reference in [9]). According to this strategy, the bindings of each `for` clause would be first computed separately, and then put together using outer joins. Here for instance is a query defining the set B_{lied} of bindings for Q_{lied}, where the bindings of *song* are inherited from outer `for` clause:

$$B_{lied}(s,t) \leftarrow B_O(s), \texttt{child}(s,t), \texttt{tag}(t,''\,title'')$$

```
Q = for E(song)
```

```
return <lied>
         Qlied(song)= for F(song,t)   Q's binding                      Q's tagging    for s in BO
                      return t        ---------->   BO(song):- E(song) ----------->      output <lied>
         ...                           stage                              stage          Qlied(s)
       </lied>                                                                           ...
                                                                                         output </lied>
```

correlated execution

```
Q's and Qlied's    BO(song) :- E(song)                  tagging    for s in BO
---------------->                                     -------->     output <lied>
binding stage      Blied(song,t) :- BO(song),F(song,t)                for (s,t) in Blied
                   ...                                                 output copy of t
                                                                      ...
                                                                    output </lied>
```

decorrelated execution

B_{lied} is captured again with two inclusion constraints, just as done for B_O. We show only one inclusion:

$$\forall s \forall t\ [B_O(s) \wedge \texttt{child}(s,t) \wedge \texttt{tag}(t,''title'') \rightarrow B_{lied}(s,t)] \quad \text{every variable binding in } Q_{lied} \text{ is in } B_{lied} \tag{11}$$

We proceed similarly for the variables bound by the subqueries nested within the `composer`, `last` and `maiden` tags.

So far, we have only described the binding stage of an XQuery, specifying the sets of tuples of variable bindings they compute (illustrated on B_O, B_{lied}). However, we still have to describe how these bindings are used for the creation of new elements in the tagging stage. Note that elements are created in two situations: either as new elements one for each occurrence of a tag in the `return` clause, or as copies of existing elements (such as the `title` subelement of the `song`-element to which the variable `$song` was bound. We first treat the creation of new elements.

Obstacle: new element creation in `return` clause.

Solution: model it using injective Skolem functions.

For every binding of the variable *song* in Q's outermost `for` clause, new, distinct `lied`, `composer`, etc. elements are created and output. Since distinct bindings generate distinct elements, there is a one-to-one correspondence between the tuples of bindings in B_O and these `lied` and `composer` elements in Q's output. Recall from section 4.1 that we already know how to describe the fact that XML elements are uniquely determined by relational tuples, namely when the latter are encoded as the former. Observe that the only difference from the encoding scenario is how the tuples related to XML elements are obtained: in the encoding scenario, they were tuples in the original data sources, while here they are tuples of variable bindings that are (conceptually) computed in an XQuery's binding stage. Regardless of where these tuples come from, we can describe their correspondence to XML elements applying the same idea: introduce Skolem functions and describe their graphs with constraints.

Let F_{lied}, F_{comp} be the Skolem functions relating each binding tuple s in B_O to the `lied`, respectively `composer` element generated for s. Also, let *musicRoot* be a function of no arguments whose result is the root of the virtual `Music.xml` document. Similarly, F_{xdb} is a Skolem function whose result is the top `xdb` element. Then Q's execution is described below in terms of these Skolem functions. For simplicity of presentation, we omit the nested queries, handled similarly by more nested loops and Skolem functions which take as arguments the tuples of variable bindings of their respective queries.

```
  compute set of tuples of variable bindings  B_O;
rRoot = musicRoot(), e_xdb = F_xdb();
make  e_xdb a child of rRoot tagged ``xdb'';
foreach binding s in B_O {
   e_l = F_lied(s); e_c = F_comp(s);
   make e_l a child of e_xdb, tagged ``lied'';
   make e_c a child of e_l tagged ``composer'';
   ...  continue recursivley for nested queries ...
}
```

We omit the constraints capturing the Skolem functions, as they have the same shape as constraints (1) through (4) shown in 4.1. The constraints organizing the created elements in the tagged tree structure specified by the return clause are similar to constraints (7) through (8).

Obstacle: deep copy of XML subtrees in the output.

Solution: element copies described by Skolem functions, copies of subtrees by recursive constraints

Recall that the nested query Q_{lied} does not return the `title` elements bound to by the variable t, but rather deep, recursive copies thereof. In our scenario, MusicMap is expressed over the XML encoding of relations and therefore it is possible to conclude that the subtree rooted at `title` elements only contains a text which is equal to the value of the `title` attribute in the corresponding `song` tuple. However, we adopt a more general approach here: we treat the encoded RDB as an arbitrary XML document, call it Xencode(Music.rdb). and we show how to capture the copying of arbitrary subtrees of the `title` elements. It is the job of the reformulation algorithm to arrive at the conclusion that `title` subtrees are really just text copies of a certain relational attribute, by reasoning with the constraints in GReXMusic. This has the advantage that we handle XQueries with arbitrary input, not just those ranging over XML encoding of relational data. We can therefore capture in the same way other mappings expressed by XQueries, such as materialized XML views of original XML documents.

Since a distinct deep copy is generated for each binding (s,t) in B_{lied}, we formalize as the predicate $C_{lied}(s,t,e,e')$ the fact that element e' is the copy determined by (s,t) of a descendant e of the `title`-element t. Notice that for any fixed copy (that is, for fixed s,t), the subtree rooted at t and its copy are in one-to-one correspondence: $C_{lied}(s,t,e,e') \Leftrightarrow F_{s,t}(e) = e'$ for some injective function $F_{s,t}$. This correspondence is modeled by constraints saying that for fixed s,t, the third component of C_{lied}-tuples functionally determines the fourth, and viceversa:

$$\forall s \forall t \forall e \forall e_1' \forall e_2' \; [C_{lied}(s,t,e,e_1') \wedge C_{lied}(s,t,e,e_2') \rightarrow e_1' = e_2'] \quad C_{lied}(s,t,e,e') \Leftrightarrow F_{s,t}(e) = e' \tag{12}$$

$$\forall s \forall t \forall e_1 \forall e_1 \forall e' \; [C_{lied}(s,t,e_1,e') \wedge C_{lied}(s,t,e_2,e') \rightarrow e_1 = e_2] \quad F_{s,t} \text{ is injective} \tag{13}$$

Finally, we use the following constraints to say that for any binding s,t, the copy of the `title`-element t is made a child of the `lied`-element created for s,t (14) and that copying an element means copying its tag (15), attributes and text (omitted, similar to (15)) and recursively its child (16) and descendant elements (omitted, similar to (16)).

$$\forall s \forall t \forall e_l [F_{lied}(s,e_l) \wedge B_{lied}(s,t) \rightarrow \exists t' \; C_{lied}(s,t,t,t') \wedge \texttt{child}(e_l,t') \wedge \texttt{tag}(t',''title'')] \quad \text{copy } \texttt{title}\text{, make child of } \texttt{lied} \tag{14}$$

$$\forall s \forall t \forall e \forall te \forall e' \; [C_{lied}(s,t,e,e') \wedge \texttt{tag}(e,te) \rightarrow \texttt{tag}(e',te)] \quad \text{copy tag} \tag{15}$$

$$\forall s \forall t \forall e \forall c \forall e' \; [C_{lied}(s,t,e,e') \wedge \texttt{child}(e,c) \rightarrow \exists c' \; C_{lied}(s,t,c,c')] \quad \text{recursively copy child elements} \tag{16}$$

Putting all pieces together, we denote the constraints capturing the meaning of MusicMap with MusicMapConstr. It is easy to see that the size of MusicMapConstr is linear in that of MusicMap.

Remark. The constraints in MusicMapConstr show how we capture the semantics of an XQuery (MusicMap)

in a purely declarative way, using first-order logic statements (constraints). The benefit of using constraints is that they are more amenable to reasoning with than the algorithmic specification in the W3C recommendation. However, we do not advocate constraints as user-level language. Instead, we translate to them mappings given as queries in the XQuery, SQL, RXL, STORED, etc. languages. The DBA can moreover write his own mappings, possibly more complex than queries in the above languages, as long as they can be expressed with/translated to constraints. Our personal favourite is an RXL-style syntax allowing relational atoms in `return` clause, or equivalently, XQuery extended with Skolem functions as explicit primitives and with the ability to bind relational tuple variables .

4.3 Mappings Specified Directly with Constraints

Recall from our example the relational view RV.rdb storing the more structured part of the Vienna.xml document. We have expressed this view in XQuery, but there is also a different, more straightforward way to achieve the same effect, writing directly constraints. We capture RV.rdb with a set RViewConstr of two constraints, one for each inclusion between the extent of RV.rdb and the fragment of Vienna.xml it stores. Here is one of them:

$$\begin{aligned}\forall r \forall d \forall p \forall pn \forall pnt \forall s \forall sf \forall sft \forall sm \forall smt \ [\ & \texttt{viennaRoot}(r) \wedge \texttt{desc}(r,d) \wedge \texttt{child}(d,p) \wedge \texttt{tag}(p,''person'') \\ & \wedge \texttt{child}(p,pn) \wedge \texttt{tag}(pn,''name'') \wedge \texttt{text}(pn,pnt) \\ & \wedge \texttt{child}(p,s) \wedge \texttt{tag}(s,''spouse'') \\ & \wedge \texttt{child}(s,sf) \wedge \texttt{tag}(sf,''first'') \wedge \texttt{text}(sf,sft) \\ & \wedge \texttt{child}(s,sm) \wedge \texttt{tag}(sm,''maiden'') \wedge \texttt{text}(sm,smt) \\ & \rightarrow \texttt{RV}(pnt,sft,smt)\ ] \end{aligned} \tag{17}$$

Note that the binding part of the XQuery in Figure 9 corresponds to the premise of (17)'s implication. The "shortcut" in expressing RV.rdb is taken in the conclusion of the implication, which mentions the target relation, rather than its XML encoding.

5 The Reformulation Algorithm

As we have explained before, we handle XQueries posed against the published XML schema by breaking them into several binding queries and a tagging part in charge of combining the results of the binding queries and creating the XML output. The binding queries correspond to the `for` clauses in the XQuery: one for the outermost clause, and one each for the nested queries (recall our discussion of nested queries in section 4, involving B_O, B_{lied} for MusicMap). The tagging is performed on the result of the binding queries, according to the *late tagging, sorted outer join* approach of [9] and is not our concern here. We focus on the reformulation of binding queries.

Reformulation. More precisely, given a binding query B against the published schema P, the $\mathcal{SS}$ storage schema elements (this includes any materialized views/cached queries), and the set M of mappings and views between P and SS, a reformulation R of B is a query posed against SS, equivalent to B over all (P, SS)-instances I in which I's restrictions to P and SS are related according to M. "Related" depends on the way the mapping is specified: for instance, if some schema element p of P is defined as a view over SS, then p's extent must be equal to the result of the query defining the view. If the mapping is given in form of constraints, these must be satisfied by the (P, SS)-instance.

Strategy: reduction to rewriting conjunctive queries under constraints. We reduce the problem of finding reformulations to that of equivalently rewriting queries under constraints, by translating the mappings/views M to a set of constraints Σ_M and searching for rewritings of B under the constraints in Σ_M and GReX. Of course, it is a priori not at all clear that these rewritings under constraints coincide with the reformulations we seek. We

shall address this issue shortly. As seen in section 4.2, binding queries are conjunctive queries, as they return sets of tuples of variable bindings (recall B_O for example). We are therefore interested in rewriting conjunctive queries under constraints.

The C&B algorithm [14]. This problem was addressed in [14], which introduced the C&B algorithm for rewriting a generalization of conjunctive queries with views, under constraints like ours but without disjunction. We adapt and extend their ideas to our scenario. The original C&B algorithm runs in two phases: the first, called *chase*, constructs the search space for rewritings, the second (*backchase*) explores it. We adapt this algorithm to our needs, inserting a stage in between the two, which is called *reduce* and which uses domain-specific knowledge to further reduce the search space constructed by the chase.

Phase 1: chase. In the first phase, the original query is *chased* with the available constraints. The chase is a rewriting technique developed by classical relational dependency theory [2], consisting of repeateadly applying chase *steps*. In every step, a mapping m is sought from the universally quantified variables of a constraint c to the variables of the query Q such that the image under m of the premise of c's implication is a subset of Q's atoms. If such a mapping exists, but cannot be extendend to the conclusion of c's implication, we say that the chase step is *applicable* and its effect consists in adding to Q the atoms from c's conclusion. The chase stops when no more steps are applicable.

Example. Suppose we want to reformulate query Q from the motivating example. The set of tuples of variable bindings produced in its binding stage is described by the following conjunctive query (given also as an example in section 2.

$$
\begin{aligned}
B_Q(p, pn, s, sf, sm, c, cf, cm) \leftarrow\ & \texttt{viennaRoot}(r), \texttt{desc}(r, d), \texttt{child}(d, p), \texttt{tag}(p, ''person''), \\
& \texttt{child}(p, pn), \texttt{tag}(pn, ''name''), \texttt{text}(pn, pnt), \texttt{child}(p, s), \texttt{tag}(s, ''spouse''), \\
& \texttt{child}(s, sf), \texttt{tag}(sf, ''first''), \texttt{text}(sf, sft), \texttt{child}(s, sm), \texttt{tag}(sm, ''maiden''), \texttt{text}(sm, smt), \\
& \texttt{musicRoot}(r'), \texttt{desc}(r, d'), \texttt{child}(d', l), \texttt{tag}(l, ''lied''), \texttt{child}(l, c), \texttt{tag}(c, ''composer''), \\
& \texttt{child}(c, cfn), \texttt{tag}(cfn, ''first''), \texttt{text}(cfn, sft), \texttt{child}(c, cmn), \texttt{tag}(cmn, ''maiden''), \texttt{text}(cmn, smt)
\end{aligned}
$$

Now observe that the identity mapping on the variables $r, d, p, pn, pnt, s, sf, sft, sm, smt$ makes the premise of the implication in (17) a subset of B_Q's atoms, namely the one corresponding to its first three lines of B_Q. Also, there is no $\texttt{RV}(pnt, sft, smt)$- atom in B_Q, so the identity mapping cannot be extended to the conlusion of the implication. A chase step applies therefore, and its effect is that of adding the latter atom to those of B_Q. We observe that the effect of chasing with the constraints from RViewConstr is that of bringing into the chase result a `V`-atom. Similarly, by continuing the chase with constraints from MusicMapConstr ∪ GReXMusic ∪ GReX it follows that `Author`, `Song` and `Maiden`-atoms are eventually added. The final result of the chase stage is a large, redundant query U, expressed against both the storage and the published schema, and therefore not executable. Here is part of the chase result:

In order not to clutter the figure, we omit most of the atoms corresponding to the variable binding relations (such as B_O, B_{lied}), Skolem functions (such as G_s, G_t), copy functions (such as C_{lied}), or even `desc` atoms between all pairs of reachable XML elements. All of these are actually added during the chase.

Observe that the queries corresponding to the bindings of R_1, R_2, R_3 can be found as subgraphs of U's query graph. It turns out that *all* binding queries of reformulations of B_Q are subgraphs of the chase result (theorem 6.1).

Phase 2: Reduce the chase result. Since all reformulations are expressed exclusively using storage schema elements, it follows from theorem 6.1 that **minimal** reformulations are actually subqueries of the largest subquery of the chase result that is induced by keeping only storage schema atoms. A reformulation is minimal if we cannot remove any of its atoms without compromising equivalence to the binding query. It makes sense to look only for such reformulations, as they contain the least number of joins. In our example, since `Music.xml` and Xencode(Music.rdb)

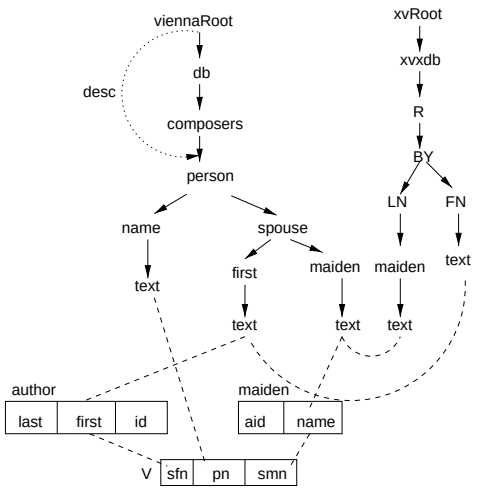

Figure 15: Fragment of chase result

are virtual documents, their root relations belong neither to the storage schema and therefore nor to the reduced chase result. The obtained subquery will therefore contain a subquery SQ consisting of the `child`,`desc`,`tag`,`text` and `attr` atoms corresponding to the navigation in the `Music.xml` and Xencode(Music.rdb) documents. Since the roots were eliminated, SQ cannot contribute to any reformulation, because legal navigation in any XML document (whether virtual or not) can only proceed by starting from some entry element (typically the root) and performing child and descendant navigation from there. This is why we may safely eliminate SQ from the chase result as well, and in general all subqueries corresponding to XML navigation that isn't rooted at some determined element. We call the remaining subquery the *maximal reformulation*.

Notice that the maximal reformulation is really an executable query (its atoms belong exclusively to the storage schema and all XML navigation is legal). However, it is redundant (it performs the work of R_1, R_2, R_3 simultaneously). In scenarios in which we know that the redundancy is expected to be small, and/or when the query involves a single data source whose query optimizer and execution engines are known to be powerful, we may chose to stop here, check equivalence of the maximal reformulation with the original query, and execute the maximal reformulation as is if the answer is positive (otherwise there is no reformulation against the given storage schema). This aproach works for the particular case of XPERANTO, whose application scenario coincides to the latter case (no redundancy, single source, good optimizer) and the obtained query is simply passed to the relational optimizer. But we may not always be that fortunate: our simple example shows that it is likely that the stored data is spread across several sources, such that no single source optimizer is aware of the other sources and schema mapping involving them. In such cases, we have to minimize the maximal reformulation ourselves in order to minimize access to sources, and, if possible, skip some storage sources altogether (as R_2 and R_3 do). The benefit of optimizing the maximal reformulation grows if the data sources have only weak optimization and processing capabilities. In such cases, we can follow the first stage with a minimization stage.

Phase 3: backchase. Minimize the maximal reformulation. In this stage, the subqueries of the maximal reformulation are explored. Doing so allows us for example to identify R_2 as a valid reformulation, rather than joining together the results of R_1, R_2 and R_3 and accessing all available data sources. Of course we are not going to explore the exponential search space of all subqueries exhaustively, but theorem 6.1 guarantees that, if we did, we would find all minimal reformulations. There are several ways of exploring this search space. All of them have one operation in common: regardless of which subqueries we explore, we have to check their equivalence under all available constraints to the original query (B_Q in our example) which is done again using the chase [2].

Cost-based pruning. This strategy is borrowed from [23]. It proceeds bottom-up, starting from subqueries consisting of one atom, going on to subqueries of two atoms, and so on, until it hits the first subquery R that is equivalent to B_Q under all available constraints. R is a reformulation of B_Q. Compute its cost and continue the exploration, pruning away all subqueries whose cost is larger than the best cost found so far. If the cost model is *monotonic* (i.e. the cost of a query increases when adding an atom to it), cost-based pruning is guaranteed to end up keeping the cheapest reformulation. Otherwise, the theoretical guarantee does not hold, but this pruning strategy becomes a useful heuristic. The implementation of our algorithm offers a plug for a cost module, which is called during the cost-based pruning.

Heuristic: cost-based pruning using number of atoms as cost. The development of a reasonable cost model for XML navigation is still a research issue (but see [10, 3] for initial work, which however does not yet cover the full XQuery navigation). We found however that the heuristic of using the number of scans as cost tends to favor reformulations using relational sources rather than XML documents whenever possible. This is because usually relational storage is *in-lined* [27], i.e. one single relational tuple contains data that corresponds to several leaves of an XML tree, who can only be reached in several navigation steps. A typical example is `V`, which allows access to the person name and to that of its spouse using a single atom, while accessing the same information in the XML document requires 15 atoms (recall B_Q). Under this heuristic, the backchase stage picks R_2 as the best reformulation, and this is likely to be a reasonable choice.

6 A Completeness Result for our Reformulation Algorithm

The schema mappings used to configure and tune the MARS system is given as a collection of XQueries, SQL queries, and constraints, all of whom are translated to a set Σ_{Map} of constraints. Any incoming XQuery Q is broken into several relational binding queries $B_1, \ldots, B_n$ (one for every nested subquery of Q), as well as a tagging template that creates XML out of the tuples obtained by combining the results of the B_i's. What we reformulate is the B_i's, and we do so without trying to preserve document order.

Well-behaved XQueries. We are interested in a significant subclass of XQuery, which we call well-behaved for reasons shown below. Such queries allow no user-defined functions, no `filter` operation, no aggregates, no navigation to a child of unspecified tag or to a parent (`*`, respectively `..` in abbreviated XPath syntax). `range` and `BEFORE` predicates are disallowed. Moreover, they use no arbitrary negation, but inequalities are allowed.

Theorem 6.1 *The chase of binding query B_i with the constraints in $\Sigma_{Map} \cup GReX$ terminates. If Q and all XQueries used to express mappings are well-behaved, then all minimal reformulations of B_i are subqueries of its maximal reformulation. Moreover, if neither Q nor the XQueries contain disjunction or inequalities, the chase terminates in time polynomial in the size of B_Q and exponential in the maximum size of a constraint in Σ_{Map}.*

One of the pleasant surprises contributing to this result is that the equivalence of well-behaved binding queries *over XML trees* coincides with that *under the constraints in GReX*. This is non-obvious, as the treeness-property as well as the fact that the descendant relation `desc` is the transitive closure of the child relation `child`, are notoriously not expressible in First Order logic, and in particular in constraints. It turns out however that what we *do* express about these properties is sufficient for characterizing equivalence if the queries are well-behaved.

Note that the algorithm is applicable also to the case when the XQueries are not well-behaved, and it will produce some reformulations, but it is not guaranteed to produce all of them, thus potentially missing the optimal one. In fact, we have the theoretical guarantee that some reformulations *will* be missed in that case, unless $NP = \Pi_2^p$. This is a corollary of the result in [15] showing that containment for ill-behaved XQueries is Π_2^p-complete in the size of the query, even in the absence of any constraints, and in the absence of disjunction. The lower bound transfers to

reformulation, which is at least as hard as checking containment. But theorem 6.1 gives an NP upper bound for reformulation in the well-behaved, disjunction-free, constraintless case.

7 Experimental Evaluation

Our reformulation algorithm for XML queries over mixed and redundant storage was implemented in the MARS system. We investigate the behavior of the algorithm in a likely scenario: when the storage is mixed, but mostly relational, and only the highly unstructured XML data is stored as character large objects (clobs) reachable from some tuples via unique identifiers.

The published schema. This mixed data is published as an XML document conforming to the DTD of the XML Benchmark project [8]. This project offers a DTD, a set of 20 queries and an XML document generator, intended to measure XQuery *execution time* for XQuery engines. Since our focus is on query *translation and minimization* time, we ignore the data generator and use only the queries and the schema. This schema can be viewed at [24] and it corresponds to the scenario we consider most likely: it consists of a large, highly structured part, augmented with some unstructured XML information. The schema models entities of an auction application, in which various items are bidden for (in auctions that are still open), or bought (appearing in the result of closed auctions). Information is also kept on the sellers, bidders and eventual buyers, who are all persons with standard attached information such as address, name, email, credit card, etc., all of it being perfectly structured. The information on auctioned items is both structured (name, provenance, seller, etc.) and unstructured (a description and various annotations containing an arbitrary mix of text, arbitrarily nested within bold and emphatic qualifiers).

The mixed storage schema. The space constraints of this submission do not permit us to show the storage schema, which can be viewed at [31]). Suffice it to say that the structured part of the data is sufficiently clear from the DTD to allow us to pick a common-sense relational schema, without needing to first perform any involved analysis of what relational storage to choose (such as shown in [13, 27]). We simply store such well-structured entities as person, item, open auction, closed auction, etc. in a total of 7 relations, and keep the less structured descriptions and annotations as clobs (containing XML text), referenced from these relations.

The queries. The 20 XQueries are designed to exercise those features of the language whose execution is non-trivial, and can be viewed at [25]. In our graphs, we will refer to them using their official names, Q_1 through Q_{20}.

The redundancy. We introduce redundancy according to an idealized scenario, in which the system is tuned to answer these 20 queries as fast as possible. To do so, we materialize 20 views, each containing the tuples of variable bindings needed by one of the 20 queries. Since these views can be seen as access suport relations in a broader sense, we shall call them ASR_1 through ASR_{20}. In their presence, each query may be answered by simply scanning its ASR and streaming the tuples to the tagger. [10]

The mapping. It turns out that this relatively simple scenario is alreay inexpressible by existing systems. As discussed previously, even in the absence of the redundant ASRs, the fact that the description and annotation data is stored as XML rather than relationally takes us beyond systems like SilkRoute and XPERANTO. Agora cannot be used either, because the storage schema contains a few attributes hidden in the published schema. They are internal keys and foreign keys, introduced for normalization purposes to preserve the relationships such as between items and the XML clobs containing their description, etc. The mapping is therefore expressed as follows: the proprietary relational data is mapped to the published schema via an XQuery. The XML descriptions and annotations are mapped from the published schema to the storage using constraints. The redundant ASRs are described by SQL

[10] Of course this is a best-case scenario for query execution, but note that from the point of view of reformulation, detecting which ASR is relevant and finding the reformulation is just as hard if the ASR covers all of the query's navigation or not.

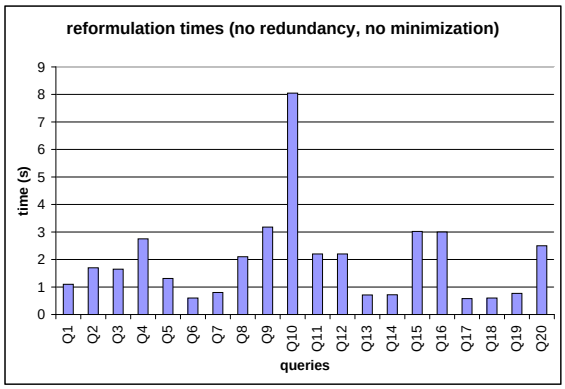

Figure 16: suite1

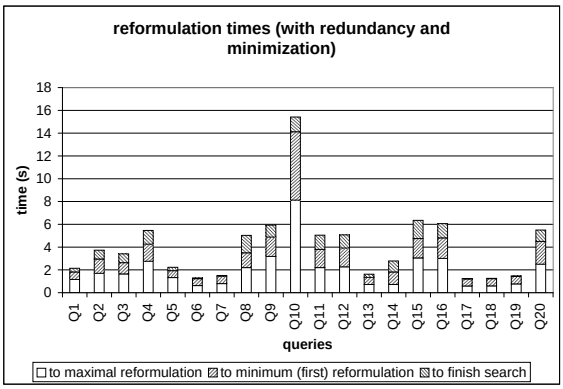

Figure 17: suite2

views over the published schema (seen in the generic relational encoding of XML). All of these compile to about 200 constraints .

The experiments. We measured the reformulation times for each query, both in the absence of ASRs and in their presence. In the latter case, the maximal reformulation is redundant, joining some proprietary relations with the ASR. The maximal reformulation is minimized according to the backchase stage of our algorithm, by exploring the the subqueries bottom-up. We obtain for each query two minimal reformulationss, one that uses the ASR and one that does not. The time to find the first reformulation coincides with the time to explore the entire search space using cost-based pruning, when the cost is the number of atoms in the query. In order to show how much more the search would go on without pruning, we also show the time to fully explore the search space (during which we also obtain the other plan).

Conclusions Notice that the time for reformulation is the same whether we add the redundant ASRs or not. Time penalty is incurred only when we switch on the minimization stage. The most time was spent reformulating query Q_{10}, which is highly nested (it has 10 correlated subqueries in the `return` clause).

The measured times should be considered in light of the time needed to execute queries of this complexity. According to [16], typical query execution times for realistically-sized documents published from databases via mappings that are no more complex than the present one, can last up of 100 seconds. The times we measured are therefore within practicality bounds. As for the execution of Q_{10}, the benchmark record so far is held by the MONET *native* XML query engine [8], and it measures 25 minutes.

A heuristic we employ to reduce the search space during minimization, is cost-based pruning using the number of atoms in the query as cost. Consequently, the first plan found is always the one using the ASR, since it contains a single atom. The time for finding this first plan is shown to be reasonable as well (no more than 3 seconds). In general, this heuristic choice of cost gives preference to plans joining as few relations as possible (one cost unit per relation), over plans that perform XML navigation. This is because reaching selected XML fragments typically means performing several navigation steps, where each navigation step contributes a unit to the cost, while the relational storage of the same information is usually *in-lined* [27] allowing direct access through the tuple's attributes.

Some engineering we employed to speed up reformulation. Note that every chase step consists of matching the premise of a constraint against the query, and adding its conclusion to it. The chase can therefore be seen as the execution of a generalized Datalog program on a small database instance corresponding to the query itself: the query atoms are the tuples in the instance. Therefore, one can employ standard relational optimization techniques to

speed up this evaluation. For example, we compile constraints down to join trees, whose nodes are relational algebra operators. The query atoms are treated as database tuples and pipelined into the join tree, where they undergo selection and joins. We implemented joins as hash-joins, and pushed selections into them. Using such set-oriented processing techniques to "optimize the optimizer" brought down reformulation time from several hours (in a first, direct implementation) to merely seconds. Of course, the bag of tricks of relational optimization is not exhausted and we plan to use more of them for further speedup.

Platform The MARS system is implemented in JDK1.2, and the experiments were run on a Dell Inspiron8000 laptop, with a 650Mhz PentiumIII processor and 256MB of RAM, running the Millenium Edition of Windows.

8 Extensions

We briefly touch on several application scenarios to which both our theoretical and practical work extend, not mentioned so far for the sake of presentation simplicity.

Integrity constraints. Our algorithm is oblivious of the source of the constraints it uses in reformulation. Therefore it is natural to provide it with integrity constraints known to hold on the published and storage schemas, in addition to the constraints our mappings compile to. Integrity constraints are useful because their presence enlarges the number of good reformulations. It turns out that for large classes of integrity constraints, our algorithm still finds all minimal reformulations (in other words, theorem 6.1 still holds). Examples are the *full relational dependencies* of [2] for the relational storage schema, and the *bounded simple XML integrity cnstraints* [15] for the published and XML part of the storage schema.

Using Skolem functions as explicit primitives for mapping queries. The mappings we have illustrated are expressed either in SQL or in XQuery. However, the constraints they compile to use Skolem functions implicitly. Again, our algorithm is oblivious of whether the Skolem functions are implicit or appear as explicit query primitives in some mapping language. Our technique of compiling queries to constraints applies seamlessly to languages using these functions explicitly (such as RXL). The benefit of using a mapping language that enriches say XQuery with Skolem functions is that it allows us to express strictly more schema mappings. In particular, we can use the Skolem functions to invent values of storage schema attributes hidden in the published schema (recall `aid` in our motivating example) In particular, we can specify in LAV-style schema mapping that were previously definable only in the GAV approach.

Using LDAP as storage. Recent research has focussed on storing XML in LDAP directories, in order to use their capabilities to answer queries [11]. The mappings used in these cases are typically not specified by the administrator, but rather a default scheme is chosen. Using constraints, we can express these default schemes, thus allowing LDAP as part of the storage mix we support.

9 Related work

Storing XML as RDB vs. native storage techniques.

Given the variety of storage schemes, it is natural to ask whether one can use a single approach for applications of all above types. In particular, early research explored the obvious avenue of eliminating heterogeneity by storing all XML data relationally, with the added benefit of exploiting the mature relational technology. The emerging consensus is that, while relational storage can (and should) be used in many important cases, it is not always sufficient. This is because highly unstructured XML is harder to store "meaningfully" in relations, therefore sometimes we may have to store only part of the XML, keeping the rest as is ([13] only used relations for the frequently encountered XML subtree types, and stored the outliers in overflow graphs.) [27] used the DTD information to generate the relational

storage schema, and therefore faces the problem that DTDs are too permissive to fully capture relationally in a natural way. Moreover, as the study in [5] reveals, DTDs are often misused, underspecifying the document structure for the sake of simplicity. Of course, there is always the generic mapping storing the edge relations of the XML tree in RDB tables ([18]). In this scheme however, a path of n navigation steps translates into just as many joins of the edge relations, thus simulating the graph navigation that is typical of DOM-based systems and therefore taking only limited advantage of the RDBs set-oriented processing capabilities (by allowing join reordering, which is impossible in DOM-based navigation). *Conclusion: mixed storage is needed.*

Research in relational data integration suggests two ways of specifying schema mappings. Each of the mentioned systems picks one of these approaches and adapts it to the case when the published schema is XML. One approach is called the *global as view (GAV)* approach to data integration, in which the global (published) schema is defined as a view over the local (storage) schema. The complementary approach is called *local as view (LAV)* because it defines the local schema as view of the global schema. They each have their advantages [20]: GAV is considered less flexible in adding new storage schema elements, requiring the redefinition of all published schema elements, to take into account the added elements. In a LAV approach, one would only have to add a new view specification, describing the added sources in terms of the published schema, independently of the pre-existing sources. On the other hand, if the mapping from storage to published schema is lossy, i.e. it hides some attributes, it cannot be specified in LAV style. **Example** The hidden `id` and `aid` attributes, as well as the integrity constraints on the source schema make a LAV description of the `Music.xml` document impossible: we cannot specify each relation as a view over it, as we need to invent values for the hidden fields. Also, defining each relation as view, in isolation from the other relations, will lose the correspondence between keys and foreign keys. A GAV-style mapping is however quite natural, as shown above. On the other hand, we can express `V` LAV-style, but not in GAV fashion, since the latter needs the view to be lossless. *Conclusion: both mapping styles need to coexist.*

Note that this LAV drawback was not an issue in the first system proposing LAV-style mappings, the Information Manifold [20], which was modeling web information sources that were typically queried iby filling out forms. In this scenario, each source is one form, all of whose fields are visible, so it can always be described as lossless view over the published schema.

XPERANTO [9, 26]. This system is intended to publish proprietary relational data as XML, and is most suited for scenarios in which there is no redundancy, no mixed storage, and the relational source is unique. The mapping is specified GAV-style, by using a default encoding of the relations as XML, and writing an XQuery against this. The reformulation algorithm performs *composition-with-views.* As a consequence of the GAV-style mapping, this system cannot integrate `V`, `XView`, `Vienna.xml` from the motivating example. In the particular scenarios of XPeranto, the MARS reformulation algorithm acheives the effect of composition-with-views. MARS borrows one of the tagging approaches proposed in XPeranto.

SilkRoute [17, 16]. This system is applicable whenever XPeranto is, and shares with it the benefits and drawbacks of the GAV approach to schema mapping. A unique feature of its mapping language, RXL, is that allows Skolem functions explicitly as language primitives. They can express the outer joins and nested correlated queries of XPeranto, and even mappings that go beyond the latter. We capture them with constraints just like we did for XQueries in section 4. We favor an RXL-like syntax for expressing schema mappings.

Agora [22] is a system intended to query mixed relational and XML sources, by specifying mappings in LAV style, and thus it inherits the limitations of this approach, not being applicable when the published schema hides proprietary data fields. But as argued very well in SilkRoute, the actual data source typically has information that we do not want to expose. The system uses a generic relational encoding of XML, which is roughly the same as GReX in MARS, *but without the constraints*, and sources are specified as SQL views against this encoding. MARS captures such mappings with constraints as particular case. Agora's reformulation algorithm performs *rewriting-*

with-views but is oblivious of constraints and only finds rewritings that hold in all relational instances. Therefore, it misses important reformulations, such as `desc-child= child-desc`, which hold in all XML documents, but not in all possible interpretations of `childand desc`!

Stored [13] is a system whose goal is to store XML losslessly, using relational storage. It defines views in LAV-style, and uses an overflow graph for the highly unstructured XML fragments. The queries are therefore reformulated to mixed queries over the relations and the XML overflow graph. The system is not applicable for publishing of proprietary relational data.

MARS's reformulation algorithm borrows from [14], but the latter was used for rewriting simpler, flat (non-nested) queries over relational/complex-valued data, under disjunction-free constraints. The work gives a limited completeness result, in which the only constraints allowed come from views, and belong to the class of *full dependencies*, which cannot capture Skolem functions. The main added value of he MARS system is to show how nested XQueries can be captured with a larger class of constraints, and to extend the algorithm to work with them, according to the challenges listed in the introduction. [15] introduces the generic relational schema of XML, and the containment decision procedures used in MARS.

10 Conclusions and Future Work

We have argued in this paper that in the most common scenarios for publishing proprietary data, this data resides in mixed XML and relational storage. This storage typically becomes redundant once the systems are tuned for performance. Note that redundancy typically occurs even within a single source, for example when certain integrity constraints are satisfied. The MARS framework is best justified in these scenarios, because it provides tools for configuring and tuning such a publishing system and it is able to exploit the storage redundancy to obtain better reformulations, from queries against the published schema, to executable queries against the storage. The nature of the challenges in doing so are both *descriptive* (providing a language that can specify the necessary schema mappings) and *computational* (exploiting redundancy in reformulation).

Constraints turn out to be a very expressive formalism, allowing us to specify the mappings supported by existing systems, and beyond. One side benefit of this work is the capturing with constraints (i.e. in First-Order Logic) of an important part of the semantics of XQueries. The resulting specification is more declarative and easier to reason with than the algorithmic one provided in the XQuery1.0 recommendation.

The chase with constraints, initially developed for optimization of relational queries, is surprisingly useful and practically relevant in our context, in which query execution times are typically high, and spending a small fraction thereof in optimization can bring spectacular improvements.

A matter of independent interest is the fact that our chase-based reformulation algorithms achieves the combined effect of existing algorithms for composition-with-views and rewriting-with-views.

We are considering several strategies/heuristics for reducing the search space for reformulations. Some are applicable even in the absence of cost information: the number of subqueries is sensitive to the number of elements in the storage schema as well as the relationships between them. It makes sense therefore to attempt to partition the storage schema into clusters of elements that are highly interconnected, and minimize separately the subqueries of the maximal reformulation induced by the elements in these clusters. A trivial clustering method would be to put in the same cluster all relations stored at the same source. This partition could be refined for example by taking into account key-foreign key relationships across relations.

An important tool for reducing the search space is cost-based pruning. A realistic cost model would of course have to consider shipping times from the sources together with estimates of the size of the result. The latter estimate is an open research issue, some initial work was done in [3, 10], but it doens't yet cover full XQueries. Our algorithm

is equippend with a plug for an arbitrary cost module, and currently we are looking for the best candidate to use. Cost-based pruning however works well only when the cost model is monotonous (any query has larger cost than its superqueries), and it is not clear to what degree cost models for XML satisfy this property.

References

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : From Relations to Semistructured Data and XML* . Morgan Kaufmann, 1999.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[3] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *VLDB*, 2001.

[4] S. Adali, K. Selcuk Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD*, pages 137 – 148, 1996.

[5] Arnaud Sahuguet. Everything You Ever Wanted to Know About DTDs, But Were Afraid to Ask. In *WebDB-2000*, 2000.

[6] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[7] BizTalk. Microsoft BizTalk Server. http://www.microsoft.com/biztalk/home, 2001.

[8] Ralph Busse, Mike Carey, Daniela Florescu, Martin Kersten, Ioana Manolescu, Albrecht Schmidt, and Florian Waas. The XML Benchmark Project. http://monetdb.cwi.nl/xml/index.html, 2001.

[9] Michael Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene Shekita, and Subbu Subramanian. XPERANTO: Middleware For Publishing Object-Relational Data as XML Documents. In *VLDB*, Sep 2000.

[10] Zhiyuan Chen, H. V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond T. Ng, and Divesh Srivastava. Counting Twig Matches in a Tree. In *ICDE*, 2001.

[11] Sophie Cluet, Olga Kapitskaia, and Divesh Srivastava. Using LDAP Directory Caches. In *PODS*, 1999.

[12] Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *VLDB*, 2001.

[13] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *SIGMOD*. ACM Press, 1999.

[14] Alin Deutsch, Lucian Popa, and Val Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *International Conference on Very Large Databases (VLDB)*, September 1999.

[15] Alin Deutsch and Val Tannen. Containment and integrity constraints for xpath fragments. In *KRDB 2001*, Sep 2001.

[16] Mary Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient Evaluation of XML Middle-ware Queries. In *SIGMOD 2001*.

[17] Mary Fernandez, WangChiew Tan, and Dan Suciu. SilkRoute: Trading between Relations and XML. In *WWW9*, 2000.

[18] Daniela Florescu and Donald Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report 3680, INRIA, 1999.

[19] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97*.

[20] Alon Halevy. *Answering Queries Using Views: a survey.* Available from `http://www.cs.washington.edu/homes/alon/site/files/view-survey.ps`, 2001.

[21] A. Kemper and G. Moerkotte. Access support relations in object bases. In *SIGMOD 1990*.

[22] Ioana Manolescu, Daniela Florescu, and Donald Kossman. Answering XML Queries on Heterogeneous Data Sources. In *Proc. of VLDB 2001*, 2001.

[23] Lucian Popa. *Object/Relational Query Optimization with Chase and Backchase.* PhD thesis, University of Pennsylvania, CIS Department, 2000.

[24] The XML Benchmark Project. DTD for XML Benchmark. http://monetdb.cwi.nl/xml/Assets/auction.dtd, 2001.

[25] The XML Benchmark Project. XML Benchmark Queries. http://monetdb.cwi.nl/xml/Assets/xmlquery.txt, 2001.

[26] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying XML Views of Relational Data. In *VLDB*, Sep 2001.

[27] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*. Morgan Kaufmann, 1999.

[28] W3C. Extensible Markup Language (XML) 1.0. W3C Recommendation 10-February-1998. Available from `http://www.w3.org/TR/1998/REC-xml-19980210`.

[29] W3C. XML Schema Part 0: Primer. Working Draft 25 February 2000. Available from `http://www.w3.org/TR/xmlschema-0`.

[30] W3C. XQuery: A query Language for XML. W3C Working Draft 15 February 2001. Available from `http://www.w3.org/TR/xquery`.

[31] XXXXXX. Mixed and Redundant Storage Schema for XML Benchmark Data. URL hidden to preserve anonimity, 2001.