MARS: A System for Publishing XML from Mixed and Redundant Storage

Alin Deutsch
University of California San Diego

Val Tannen
University of Pennsylvania

Abstract

We present a system for publishing as XML data from mixed (relational+XML) proprietary storage, while supporting redundancy in storage for tuning purposes. The correspondence between public and proprietary schemas is given by a combination of LAV- and GAV-style views expressed in XQuery. XML and relational integrity constraints are also taken into consideration. Starting with client XQueries formulated against the public schema the system achieves the combined effect of rewriting-with-views, composition-with-views and query minimization under integrity constraints to obtain optimal reformulations against the proprietary schema. The paper focuses on the engineering and the experimental evaluation of the MARS system.

1 Introduction

In their most basic form, XML publishing systems are analogous to the Global-As-View (GAV) data integration scenario [20]. Corporations publish virtual views of their proprietary business data (the local sources) in the form of XML documents (the global data). Sensitive proprietary information is typically *hidden* by these GAV views. Such systems receive client queries against the virtual XML. To answer them, the system must *reformulate* them as queries on the actual proprietary data. This is done using algorithms performing so-called *composition-with-views*. This basic functionality is provided by systems such as XPeranto [30] and SilkRoute [16].

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

In addition to the basic functionality, it is highly desirable for such a system to allow for operational tuning in order to speed up the execution of certain classes of queries. By tuning we mean, eg., materializing views, creating indexes, and maintaining caches of previously answered queries. For example, portions of XML documents could be also stored relationally in order to exploit mature relational technology (see recent research in [7, 31, 4]). The proprietary storage, therefore, is both *mixed* (relational + XML) and *redundant*.

A shortcoming of the GAV-only approach is that it cannot easily model redundant source data [20]. Redundancy is best modeled by the complementary approach, Local-As-View (LAV), in which the local data (our proprietary data) is described as views of the global data (our published data). Query reformulation is done using algorithms for *rewriting-with-views*. Agora [23], which also handles mixed storage, is an example of a publishing system based on the LAV paradigm. The essential shortcoming of LAV is that all usable proprietary data is in views of the public data. Hence, a LAV-only approach cannot properly support the hiding of sensitive proprietary data. What is really needed, and what MARS provides, is the ability to combine the LAV and GAV approaches, as the following example illustrates.

Example 1.1 Figure 1 shows a proprietary relational database `patient` with two tables: patient name - diagnosis, and patient name - drug - usage. This data is partially published as XML using the GAV approach, through the mapping/view `CaseMap` that produces `case.xml`. We emphasize that `case.xml` is a virtual document; it is the result that would be obtained if `CaseMap` were run. `CaseMap` joins the two tables on the patient name but then it projects the name away, hiding this sensitive information. Clearly, it is not possible to express `patient` as a view of `case.xml`. The LAV approach cannot capture this situation.

The proprietary data is mixed: we also have a native XML document `catalog.xml` which associates to each drug a price and some notes (on side-effects, generic alternatives, etc.). This part of the proprietary data is published in its entirety through the identity

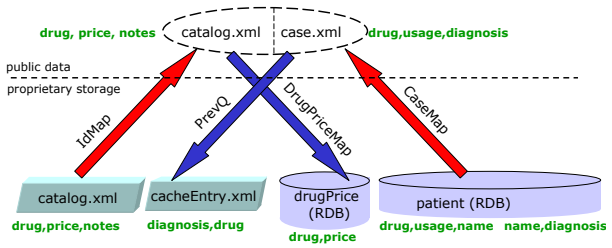

Figure 1: Configuration for Example 1.1

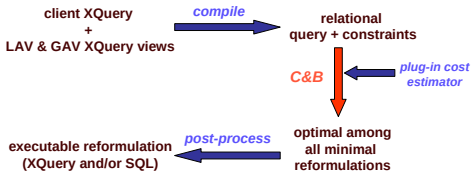

Figure 2: Reduction of XQuery reformulation to minimization of relational queries under constraints.

mapping/view `IdMap`.

The rest of the stored data comes from tuning. To speed up querying involving `catalog.xml` we decide to put some of its data redundantly in relational storage. We choose to copy the drug - price information, in a table `drugPrice` leaving the irregularly structured notes only in XML. Such a choice could be made for example by the STORED system [7] where `drugPrice` would be specified (in LAV style) as a materialized view of `catalog.xml`. We shall call the query defining this view `DrugPriceMap`.

For more tuning, suppose we cache an XML document `cacheEntry.xml` that is the result of a previously answered query, call it `PrevQ`, that retrieves from `case.xml` only the association between drugs and diagnosis disregarding usage. `PrevQ` is another example of LAV view. No GAV view could have done this. Overall, the proprietary-public correspondence in this example needed a combination of two GAV and two LAV views (see the extended version [13] for the code of this example).

Now assume that a client query is posed against the published documents `catalog.xml` and `case.xml`, to find the association between each diagnosis and the corresponding drug's price. Because of the redundancy in the stored data, there are multiple reformulations of this query, notably as queries that:

- access `patient` and `catalog.xml`, or
- access `patient` and `drugPrice`, or
- access `catalog.xml` and `cacheEntry.xml`.

With the `drugPrice` table stored in the same RDBMS as the `patient` information and with current technology, the second one is likely the best. •

This example shows that *redundancy enables multiple reformulations*, some potentially cheaper to execute than others. Hence, MARS should also be able to find and compare the candidates for best reformulation.

GLAV A specific way of combining LAV and GAV, known as GLAV has been introduced in the context of data integration [19]. This approach was developed for relational conjunctive queries but even if it were further adapted to XML publishing we would have the following problem. GLAV reformulation looks for a maximally contained reformulation of the original query even when there is no equivalent reformulation. This is achieved by generating a Datalog program who accesses *all* available data sources. With redundancy in storage, this means redundant data accesses, i.e. a non-optimal reformulation, thus defeating the purpose of tuning in XML publishing. In the context of Example 1.1, this reformulation would access `patient`,`catalog.xml`,`drugPrice` and `cacheEntry.xml`, performing the combined work of all three reformulations we listed.

The GLAV approach relies on previous LAV reformulation techniques. But the reformulation algorithms used for rewriting-with-views are quite different from the algorithms for composition-with-views used for GAV. Hence the inefficient reformulation described above that performs redundant work. To be able to combine GAV and LAV in MARS we switched to a different, "direction-neutral", representation of the views, compiling them into *constraints*. Moreover, in order to avoid reformulations that perform redundant work like the one above, we have a *query minimization* (under constraints) capability in MARS. Thus MARS has a completely different kind of reformulation algorithm leading to a new set of engineering difficulties.

The MARS system. We present a system called MARS (Mixed And Redundant Storage) for publishing as XML data from mixed (XML+relational) proprietary storage, while supporting redundancy in storage for tuning purposes. Client queries, as well as GAV and LAV views are expressed using the standard XQuery language [34], and MARS finds the *minimal* query reformulations. By minimal we mean (roughly) that there is no redundant data source scan. This is compatible with cost reduction, assuming that the cost model is monotone. In addition, MARS exploits integrity constraints on both the XML and relational data. This may help in finding even better reformulations. The functionality of MARS subsumes and goes beyond that of existing systems such as XPeranto, SilkRoute (no redundancy, no integrity constraints) and Agora (no data hiding, no integrity constraints).[1]

Prior theoretical work. MARS uses a compilation of queries, views and constraints from XML into the relational framework. In [11] the compilation algorithm was introduced and proven complete for a large

[1] Like these other middleware systems, MARS does not manage its own storage. It reformulates queries and then sends them to actual relational or native XML database engines for execution.

and expressive fragment of XQuery and an expressive XML constraint language (see also [12, 13]). The compilation reduces the original reformulation problem to a problem of minimization of relational queries under relational integrity constraints. This problem in turn is solved by the C&B algorithm [8] (see Figure 2). A second result in [11] shows that the C&B algorithm is itself complete, for a large and expressive class of relational queries and constraints. Hence the MARS reformulation algorithm is overall complete (under the restrictions detailed in [13]). Complete means that MARS finds *all* existing minimal reformulations.

The value of completeness. MARS compares all minimal reformulations using a plug-in cost estimator. Assuming a monotone cost model this guarantees that the reformulation with optimal cost will be found. Previous XML publishing systems that handle redundancy do not make such claims. One could ignore theoretical incompleteness if the practical aspects of the problem were well understood and a convincing body of examples were given in which the "desired" reformulations are found anyway. However, the semantics of XQuery is quite complicated and we do not think that the research field has developed enough practical experience for this. Our theoretical guarantee of completeness avoids such worries, assuming of course that the algorithm can be implemented practically!

The focus of this paper: MARS engineering. A priori challenges: several phases of the reformulation algorithm run in worst-case exponential time (this is inherent as the underlying problems are NP-hard). But in practice the input (queries, views, constraints) can be small enough to make the problem feasible. In this paper, we show that this approach is indeed feasible, and that its reformulation effort is worthwhile. Salient contributions:

1. We present a series of **XML-specific optimization techniques** which exploit the fact that the input to the C&B algorithm is not arbitrary relational queries and integrity constraints, but rather results from the reduction of XML query reformulation. The effect of these techniques is to significantly restrict the space explored by the C&B algorithm in search for minimal reformulations.

2. A new, scalable implementation of the C&B algorithm for minimization of relational queries under integrity constraints. A previous, direct implementation of the C&B algorithm was described in [26], and was shown to be practical for query and constraint sizes that arise in relational scenarios (queries with up to 15 joins, and about a dozen constrains). However, even after using the XML-specific optimization techniques, the reduction from XML reformulation produces much larger relational queries (hundreds of joins) and more numerous constraints (hundreds as well). The prototype in [26] does not scale to such input, and we are not aware of any other system providing a module for complete minimization of queries under such a large class of constraints. Consequently, the new C&B implementation is a contribution of independent interest even to the area of optimization of relational queries under semantic integrity constraints.

The new implementation is not based on incremental engineering of the original one, but rather on an entirely new paradigm. The main operation used in the C&B algorithm is the *chase*. The new implementation exploits the observation in [25] that chasing a query Q with a constraint c can be viewed as evaluating a relational query obtained from c over a small database obtained from Q. This enables us to apply classical query processing techniques (such as compiling constraints to join trees, pushing selection into them, etc.) to obtain a scalable implementation. We measure the improvements in several experiments and find that the new implementation is between 30 and 100 times faster.

3. We introduce **schema specialization**, a new technique which exploits regularity in the structure of XML documents to obtain a reduction to more concise relational queries and constraints. Since minimization is NP-hard in the query and constraint size, we obtain significant savings this way. The technique can be applied to any XML document but its benefit grows with the degree of regularity exhibited, and the best case scenario is reached when the document is really an XML dump of relational data (which is quite common in XML publishing). Practice has shown that it is reasonable to expect a quite high degree of regularity in the structure of most XML documents.

4. We conduct **experiments** that show the practicality of the approach. We use synthetic configurations to measure separately the improvements due to the new C&B implementation, and those due to schema specialization. The synthetic configurations go beyond realistic situations in order to study the correlation between the complexity of the reformulation problem and MARS performance. The reformulation times turn out to be highly worthwhile when related to the net savings in execution time by leading XQuery engines. We use a scenario based on the XML benchmark [27] to demonstrate the feasibility of overall reformulation on realistic queries and views that exercise various features of XQuery. The reformulation times are well within feasibility range (with an average of about 350 ms). This scenario also allows us to discuss qualitatively how MARS supports the XML publishing paradigm by relating reformulation times to execution times reported in [27].

Other related work. There has been much work in LAV reformulation, see the references in [20]. [17] reformulates simple XPath expressions using extended access support relations, which are LAV views in a broader sense. The LegoDB project [4] performs re-

formulation of XQueries against a relational schema chosen to best support a certain application. The reformulation algorithm is similar to the one performed in [31], which picks the storage schema according to the XML DTD. Since we focus on publishing, MARS must support storage that may not conform to either of the forms chosen above, which calls for more flexible storage mappings. At the other extreme, we find systems storing XML in native storage, such as Natix [24] and TIMBER [21]. The reformulation happening here is mostly from the logical XQuery level to algebraic operator plans (such as the TAX algebra for TIMBER). [3] solves a particular case of MARS minimization, namely minimization of XPath queries defined by tree patterns, under a simple class of constraints inferred from a DTD.

Paper organization. In section 2 we describe MARS' architecture and we review the XML-to-relational compilation algorithm [11] as well as the principles of the C&B algorithm [8]. In section 3 we describe the new implementation of the C&B algorithm and the XML-specific optimizations that we add to C&B. In section 4 we describe the experiments, except those related to schema specialization. Schema specialization and its experimental evaluation are the subject of section 5. Conclusions and further work are in section 6.

2 Architecture of the MARS System

As shown in Figure 2, the strategy behind the design of MARS is to reduce the input to the MARS system to a problem of reformulation of relational queries under relational integrity constraints, and then use the C&B algorithm to solve the latter. More details are shown in the architecture diagram in Figure 3. We start our overview of the MARS system by describing its input.

2.1 Input to the MARS system

`XBind` queries. Like [5, 23] we follow [16] in splitting *XQuery = navigation part + tagging template*. According to [34], in a first phase the navigation part searches the input XML tree(s) binding the query variables to nodes or string values. In a second phase that uses the tagging template a new element of the output tree is created for each tuple of bindings produced in the first phase. Previous research has addressed the efficient implementation of the second phase [14, 30], and we adopt the *sorted outer union* approach from [30]. Only the first phase depends on the schema correspondence so we focus in this paper on reformulating the navigation part of XQueries. To describe this first navigation stage, we introduce `XBind` queries, a notation that abstracts away the process of generating XML tags, describing only the navigation and binding of variables. Their general form is akin to conjunctive queries. Their head returns a tuple of variables, and the body atoms can be purely relational

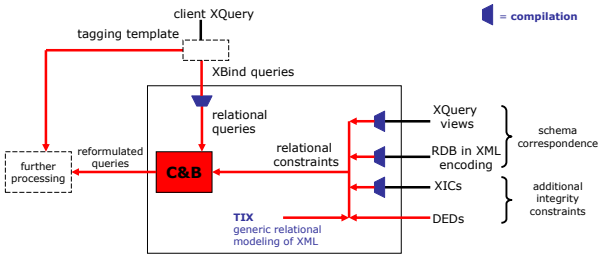

Figure 3: MARS architecture

or are predicates defined by XPath expressions [33]. The predicates can be binary, of the form $[p](x, y)$, being satisfied whenever y belongs to the set of nodes reachable from node x along the path p. Alternatively, predicates are unary, of form $[p](y)$, whenever p is an absolute path starting from the root.

Example 2.1 Consider a document containing `book` elements, each of whom contain a `title` and some `author` subelements. The query Q below restructures the data by grouping the book titles with each author.

```
Q:  <result>
      for    $a in distinct(//author/text())
      return
            <item>
              <writer>$a</writer>
              {for   $b in //book
                     $a1 in $b/author/text()
                     $t in $b/title
               where $a = $a1
               return $t}
            </item>
    </result>
```

A naive way of evaluating Q is via nested loops. However, research in evaluating correlated SQL queries suggests an alternative strategy that consists in breaking the query into two decorrelated queries which can be efficiently evaluated separately and then putting together their results using an outer join [29]. We will borrow this technique, obtaining for Q the two decorrelated `XBind` queries below. Xb_o computes the bindings for the variables introduced in the outer `for` loop ($a), while Xb_i computes the bindings of the variables from the inner loop ($b,$a1,$t) that agree with some value for $a as computed by Xb_o. Notice that this value of $a is output by Xb_i as well, in order to preserve the correlation between variable bindings. In the query definitions below, we drop the $ signs from the variable names.

$$Xb_o(a) :- [//author/text()](a)$$
$$Xb_i(a, b, a1, t) :- Xb_o(a), [//book](b),$$
$$[./author/text()](b, a1), [./title](b, t), a = a1 \bullet$$

In summary, we describe the navigational part of an XQuery by a set of decorrelated `XBind` queries, each of them to be reformulated by MARS.

The schema correspondence. To make view specification user-friendly we allow the DB administrators to think of the stored relational schema through an XML "lens" and therefore (as in XPeranto [5]) to specify the views in XQuery. Any of several straightforward RDB in XML schema-level encoding can be picked, which is then compiled into a set of relational constraints (Figure 3). Then, RDB→XML or XML→RDB mappings/views (e.g., `CaseMap`, respectively `DrugPriceMap` in Figure 1) can be defined just as XML→XML mappings written in XQuery. The schema correspondence is hence given by several **XQuery views** (in both directions, LAV+GAV) between published and proprietary schemas.

Integrity constraints: XICs and DEDs. We support integrity constraints on both the XML and the relational data. The relational integrity constraints are expressed as *disjunctive embedded dependencies* (DEDs) introduced in [10]. DEDs extend classical embedded dependencies [1] with disjunction and nonequalities and they can express all common relational integrity constraints such as referential integrity and key constraints. All constraints in the set `TIX` presented in section 2.2 are DED examples.

For XML, MARS accepts as input the class of XML Integrity Constraints (`XICs`) introduced in [9]. `XICs` have the same general form as DEDs, in which relational atoms are replaced by predicates defined by XPath expressions, just like for `XBind` queries. `XICs` allow us to express key and inclusion constraints such as given by XML Schema, but also more expressive integrity constraints. For example, (2) below says that each `person` element has a `ssn` child element, and (1) says that the `ssn` element is a key for `person` elements.

$$\forall p,q,s \quad [//person](p) \wedge [./ssn](p,s) \wedge [//person](q) \wedge [./ssn](q,s) \rightarrow p = q \quad (1)$$

$$\forall p \quad [//person](p) \rightarrow \exists s\ [./ssn](p,s) \quad (2)$$

2.2 Basics of XML-to-relational compilation

The relational framework we compile to. The above input is compiled to the relational input of the C&B module as follows. We define a *generic relational schema encoding XML*, called `GReX`

$$\texttt{GReX} = [\texttt{root}, \texttt{el}, \texttt{child}, \texttt{desc}, \texttt{tag}, \texttt{attr}, \texttt{id}, \texttt{text}].$$

The relations `child`, `desc`, `root`, etc. model respectively the parent-child, ancestor-descendant and root relationships in the XML document. `el` contains the set of element nodes, and `tag`, `attr`, `id`, `text` associates to each element node respectively a tag, attribute, identity and `text`. We emphasize that the XML data is not stored according to `GReX`. `GReX` is just a logical representation for reasoning about XQueries.

Of course, `child` and `desc` are not independent relations (the second is the transitive closure of the first). We try to capture such relationships using a set of *built-in integrity constraints* called `TIX` (True In XML). The full list is given in [11]. We show only the most interesting ones below. For example, (base),(trans),(refl) state that `desc` is a reflexive, transitive relation which contains `child`. (line) says that all ancestors of an element reside on the same root-leaf path. Similarly, `TIX` contains key constraints which say that elements have only one tag or attribute of given name, etc., in all, there are 13 such built-in constraints. Note that all constraints in `TIX` are DEDs.

$$\text{(base)} \quad \forall x,y\ \texttt{child}(x,y) \rightarrow \texttt{desc}(x,y)$$

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

$$\text{(refl)} \quad \forall x\ \texttt{el}(x) \rightarrow \texttt{desc}(x,x)$$

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

The compilation. The relational framework consisting of the schema `GReX` and the constraints `TIX` is what we compile the MARS input to. In particular, we compile:

(i) each `XBind` query describing the navigational part of the client XQuery into a relational conjunctive query with inequality and disjunction over schema `GReX`. The compilation is done by a straightforward syntax-directed translation. For instance, the `XBind` query Xb_0 from Example 2.1 compiles to

$$B_o(a) :- \texttt{root}(r), \texttt{desc}(r,d), \texttt{child}(d,c), \texttt{tag}(c,''author''), \texttt{text}(c,a) \quad (3)$$

(ii) the `XICs` into sets of relational DEDs over `GReX`. This is done using the same translation of path atoms into relational atoms over `GReX`, used in compiling `XBind` queries. For instance, the `XIC` (2) compiles to

$$\forall r,d,p\ \texttt{root}(r) \wedge \texttt{desc}(r,d) \wedge \texttt{child}(d,p) \wedge \texttt{tag}(p,''person'') \rightarrow \exists s\ \texttt{child}(p,s) \wedge \texttt{tag}(s,''ssn'')$$

(iii) the XQuery views in the schema correspondence into sets of DEDs (as detailed in section 2.4).

Solving the relational problem. We now have a relational query that needs to be reformulated modulo equivalence under the set of all relational constraints obtained from compiling the schema correspondence, the `XICs`, and from adding the built-in constraints from `TIX`. For this we use the **C&B algorithm** described in Section 2.3. For details on the restrictions under which we can provide theoretical guarantees for this approach, see [11, 12].

2.3 The C&B Algorithm

We explain C&B through an example involving views. To deal with view uniformly as with constraints, we model conjunctive query views with DEDs relating the input of the defining query with its output.

For example, consider the view defined by $V(x,z) : -A(x,y), B(y,z)$. The following dependencies state the inclusion of the result of the defining query in the relation V (c_V), respectively the opposite inclusion (b_V).

$$(c_V) \qquad \forall x \forall y \forall z \; [A(x,y) \wedge B(y,z) \rightarrow V(x,z)]$$
$$(b_V) \qquad \forall x \forall z \; [V(x,z) \rightarrow \exists y \; A(x,y) \wedge B(y,z)]$$

Phase 1: the chase. Universal Plan. Assume that in addition, the following semantic constraint is known to hold on the database (it is an inclusion dependency):

$$(ind) \qquad \forall x \forall y \; [A(x,y) \rightarrow \exists z \; B(y,z)]$$

Let $Q(x) : -A(x,y)$ be the query to reformulate. In the first phase, the query is chased [1] with all available dependencies, until no more chase steps apply (the chase definition is recalled in section 3.1, where the implementation is discussed as well). The resulting query is called the *universal plan*. In our example, a chase step with (ind) applies, yielding $Q_1(x) : -A(x,y), B(y,z)$ which in turn chases with (c_V) to the universal plan $Q_2(x) : -A(x,y), B(y,z), V(x,z)$. Notice how the chase step with (c_V) brings the view into the chase result, and how this was only possible after the chase with the semantic constraint (ind).

Phase 2: the backchase. Subqueries. In this phase, the *subqueries* of the universal plan are inspected and checked for equivalence with Q. Subqueries are induced by a subset of the atoms in the body of the universal plan, using the same variables for the head. For example, $S(x) : -V(x,z)$ is a subquery of Q_2 which turns out to be equivalent to Q under the available constraints, as can be checked by chasing S "back" to Q_2 using a chase step with (b_V).

Minimal reformulations. Notice that when applied to the particular case when (i) $\{A, B\}$ is the public schema, (ii) $\{V\}$ is the storage schema, and (iii) the schema correspondence is given in LAV style by the query defining V, the C&B algorithm discovers the reformulation S of Q exploiting the semantic constraint (ind). It is not accidental that a reformulation of Q was found among the subqueries of the chase result: in [11], we prove that *all minimal* reformulations can be found this way. A reformulation is minimal if no atoms can be removed from its body without compromising equivalence to the original query. More intuitively, minimal reformulations perform no redundant data accesses, and this is why it makes sense to restrict our attention to them. Clearly, if a reformulation exists then a minimal one will exist too.

Cost-based pruning. Even so, a query may have exponentially many minimal reformulations, and we do not want to inspect all of them to find one of minimum cost. To this end, the backchase stage performs cost-based pruning as implemented in [25]: start by examining all subqueries of one atom, next all those of two atoms, etc. in a bottom-up fashion. When a subquery is hit which is equivalent to the original query, a reformulation is found, and the best cost seen so far is updated. Whenever the cost of a subquery S is higher than the best found so far, S is discarded, as well as all subqueries containing the atoms of S. Note that a subquery is not yet an execution plan, it only specifies which relations are to be joined. To cost a subquery, the algorithm performs join reordering using dynamic programming (see [25] for details).

Initial reformulation. In scenarios when the proprietary storage features no significant redundancy, or when finding any reformulation fast is more important than finding a non-redundant one, we can simply "switch off" the backchase minimization, and return the largest subquery M induced by proprietary schema atoms. It follows from [11] that, if there exists some reformulation, then M is a reformulation as well. We call this the *initial reformulation*. Notice that all minimal reformulations are subqueries of it, so it is in general not minimal, especially when the proprietary storage is redundant. It so happens that for the above example, there is no redundant storage, and the initial reformulation is minimal. We know from [11] that only polynomially many chase steps (in the size of the original query) apply until the universal plan is reached. However, each step is exponential in the size of the constraint, so it is a priori not clear how the chase behaves before running extensive experiments. We do so in Section 4, showing that M is always found fast. In contrast, obtaining a minimal reformulation can take worst case exponential time in the size of the universal plan, if the backchase has to inspect many subqueries before finding it. In section 4 we show that for common scenarios there is significant benefit to nevertheless search for the best cost minimal reformulation.

2.4 Compiling XQuery Views to DEDs

We have seen in section 2.3 how we model a relational view defined by a conjunctive query with two DEDs stating the inclusions between the extent of the view and the result of its defining query. Adapting this idea to views defined by XQueries is not straightforward because these are more expressive than conjunctive queries in the following ways. (i) XQueries contain nested, correlated subqueries in the return clause. (ii) XQueries create new nodes, which do not exist in the input document, so there is no inclusion relationship between input and output node sets. (iii) XQueries return deep, recursive copies of elements from the input. Due to space constraints, we only sketch the solution for (i) and (ii) here (see [11, 13] for (iii)).

Nested, Correlated Subqueries. Recall that the navigation part of an XQuery is described by a set of decorrelated XBind queries (Xb_o,Xb_i in example 2.1), which can be straightforwardly translated to relational queries over schema `GReX`(recall B_0 from

equation (3)). These are (unions of) conjunctive queries, which can be modeled with two DEDs, as shown in section 2.3. For B_0, here is one of the resulting DEDs:

$$\forall r, d, c, a \quad \texttt{root}(r) \wedge \texttt{desc}(r, d) \wedge \texttt{child}(d, c) \wedge \texttt{tag}(c, ''author'') \wedge \texttt{text}(c, a) \rightarrow B_o(a) \quad (4)$$

Creation of new elements. For every binding for $a, a new element node tagged `item` is created whose identity does not exist anywhere in the input document, but rather is an invented value. This value depends on the binding of $a, as distinct bindings of $a result in distinct invented `item` elements. In other words, the node identities of the `item` element nodes are the image of the bindings for $a under some injective function F_{item}. We capture this function by extending the schema with the relational symbol G_{item}, intended to hold the graph of F_{item}: $G_{item}(x, y) \Leftrightarrow y = F_{item}(x)$. Similarly, we introduce the relational symbol G_{writer} to capture the invention of `writer` element nodes. Recall that Q constructs a *unique* `result` element node, whose identity does not depend on the bindings of Q's variables (it is a constant). This constant is represented as a function of no arguments F_{result} whose graph is the unary relation G_{result}. We use constraints to enforce the intended meaning for $G_{item}, G_{writer}, G_{result}$. Here are a few of them (Q's input and output are XML documents encoded over schema $\texttt{GReX}_1$, respectively $\texttt{GReX}_2$):

$$\forall x_1, x_2, y \; [G_{item}(x_1, y) \wedge G_{item}(x_2, y) \rightarrow x_1 = x_2] \quad (5)$$

$$\forall x, y_1, y_2 \; [G_{item}(x, y_1) \wedge G_{item}(x, y_2) \rightarrow y_1 = y_2] \quad (6)$$

$$\forall x \; [B_o(x) \rightarrow \exists y \; G_{item}(x, y)] \quad (7)$$

$$\forall x, c \; [G_{item}(x, c) \rightarrow \exists r \; G_{result}(r) \wedge \texttt{child}_2(r, c) \wedge \texttt{tag}_2(c, ''item'')] \quad (8)$$

$$\forall a, w \; [G_{writer}(a, w) \rightarrow \texttt{text}_2(w, a)] \quad (9)$$

$$\forall a, w \; [G_{writer}(a, w) \wedge \texttt{desc}_2(w, d) \rightarrow d = w] \quad (10)$$

(5) and (6) state that F_{item} is an injective function. By (7), the domain of F_{item} contains the set of bindings for $a. The range of F_{item} consists of `item` nodes that are children of the `result` node (8). The contents of the `writer` elements is the text $a was bound to (9), and the `writer` node has no children (10).

The important thing to note is that we can compile any behaved XQuery view to a set of DEDs. The number of DEDs is linear in the size of the view, but still significantly larger than the 2 DEDs needed for relational views.

3 Engineering the MARS System

The implementation of the reformulation algorithm described in the previous section raises difficult problems. These problems are caused by the fact that the compilation of the XML views and queries produces large numbers of relational dependencies as well as relational conjunctive queries and dependencies with many atoms.

"Stress" test for the chase phase. We chose an XPath query fragment that is probably too complex to arise often in practice but that we would definitely want to be able to handle: $//a/b/c/d/e/f/g/h/i/j$. This compiles to a conjunctive query with 20 atoms (9 `child`, 1 `desc`, 10 `tag`). We want to chase this query with a set of DEDs consisting mostly of the ones from `TIX`. None of the DEDs has more than 2 atoms in the premise. The chase result should contain 270 atoms (representing a 270-way join).

We used the first implementation of the C&B algorithm [26, 25] on this problem (chase only) without converging after running for more than 12h. This first implementation was able to handle relational and object-oriented query reformulation problems of practical sizes. But this stress test proved that it would not scale up to handling XML query reformulation problems. Therefore, we set forth to

- build a novel, significantly more efficient, implementation of the C&B algorithm, specifically of its critical chase component, and
- incorporate into the C&B implementation optimizations that take advantage of the specific nature of the queries and dependencies resulting from XML compilation.

3.1 A Novel C&B Implementation

A chase step of a query Q with a constraint (c) *applies* only if (i) we can find a mapping h from the universally quantified variables of (c) into the variables of Q, such that the image under h of each atom of the premise of the implication in (c) is in the body of Q. Such a mapping is called a *homomorphism* from the premise of (c) into the body of Q. (ii) Moreover, there should be no extension of h to a homomorphism from the conclusion of (c) into the body of Q. The *effect* of a chase step is to add to the body of Q the image of (c)'s conclusion under h (if the conclusion has existentially quantified variables, fresh variables are used).

Example 3.1 Assume we want to chase the query $Q(a, g) \; : \; -R(a, b), R(b, c), R(c, d), S(d, e), S(e, f), S(f, g)$ with the following constraint, call it (c): $\forall x, y, z, u, v \; R(x, y) \wedge R(y, z) \wedge S(z, u) \wedge S(u, v) \rightarrow T(x, v)$. It is easy to see that the mapping $m = \{x \mapsto b, y \mapsto c, z \mapsto d, u \mapsto e, v \mapsto f\}$ is a homomorphism from (c)'s premise, but not its conclusion. The effect of the chase step is to add $T(m(x), m(v))$, i.e. $T(b, f)$, to the body of Q. Notice that, when attempting to chase the resulting query with (c), the chase step does not apply since condition (ii) is violated. •

Notice how a chase step can alternatively be seen as evaluating a query obtained from (c) on a symbolic

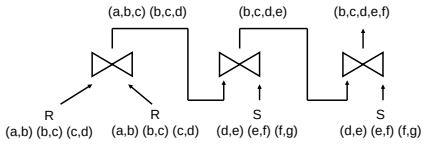

Figure 4: Evaluating JT_P on *Inst(Q)*

database instance whose constants are the variables of Q, and whose tuples are the atoms in Q's body. For Example 3.1, we represent Q internally as a symbolic database instance *Inst(Q)* consisting of the relations R, S of extents $\{(a,b),(b,c),(c,d)\}$ respectively $\{(d,e),(e,f),(f,g)\}$. (c) is represented as the query $T(x,v) :- R(x,y), R(y,z), S(z,u), S(u,v)$. Consequently, we can employ standard relational optimization techniques to speed up this evaluation. For example, we compile constraints down to operator trees, whose nodes are relational algebra operators. Continuing Example 3.1, we compile (c)'s premise into the join tree JT_P from Figure 4. (c)'s conclusion is precompiled to a join tree JT_C (this is a single node tree containing a scan of the T relation). This compilation step is done once and for all, when the constraints are read into the system. Figure 4 shows the intermediate results obtained at every node when evaluating JT_P on *Inst(Q)*. The only tuple propagating bottom-up to the root is (b,c,d,e,f), which corresponds to the homomorphism m from Example 3.1. Next, we need to check that m cannot be extended to a homomorphism from (c)'s conclusion into the body of Q. The homomorphisms that extend to the conclusion are easily computed in bulk using the same idea: compute the *semijoin* of the result of evaluating JT_P with that of evaluating JT_C. In our example, the semijoin is empty, therefore m has no extension to (c)'s conclusion, and the chase step applies. Its effect is adding the tuple $T(b,f)$ to *Inst(Q)*.

We implemented joins as hash-joins, and pushed selections into them. Using such set-oriented processing techniques reduced the time to chase tremendously. Clearly, since in the backchase phase we chase subqueries "back" to the universal plan, this phase benefits from the chase speedup as well. In section 4, we measure this speedup experimentally.

As a first encouraging sign, the new MARS C&B implementation *runs in 2.6s* through the chase stress-test problem described earlier that required more than 12h from the implementation from [26].

3.2 XML-specific Optimizations

Short-cutting the chase. For particular sets of constraints, we can predict the chase outcome up front. Doing so allows us to skip the chase entirely and construct its result directly. For example, we observe that the result of chasing a query solely with the (refl),(base) and (trans) DEDs from `TIX` (recall section 2) adds to this query those `desc` atoms missing from the reflexive, transitive closure of the `child` atoms. We can think of the chase as proceeding according to the following conceptual implementation:

repeat until no more chase step applies:
(1) chase with (refl),(base),(trans) until termination
(2) continue with all other DEDs until termination
end

Since we know up front what the result of phase (1) in every iteration is, (namely the reflexive, transitive closure) we do not have to use the chase to compute it. Instead, we jump directly to the beginning of phase (2) by computing the closure using a standard adjacency matrix-based algorithm. This trick cuts the time to chase considerably.

Example Consider a chain of n atoms

$$\texttt{root}(x_1), \texttt{child}(x_1,x_2), \ldots, \texttt{child}(x_{n-1},x_n) \qquad (11)$$

Chasing with (refl),(base),(trans) will add atoms $\texttt{desc}(x_2,x_2)$, $\texttt{desc}(x_2,x_3)$, $\texttt{desc}(x_2,x_4)$ etc., resulting in $\frac{n(n+1)}{2}$ chase steps. Their effect can be simulated by directly computing the transitive closure of `child` in the symbolic instance associated to the chain. •

Recall the stress-test in which the time to chase was cut from over 12 hours to 2.6 seconds using the new implementation with join trees. If we add this shortcut technique, the same result is obtained in 640ms.

Pruning the Backchase. Recall that during the backchase phase, we inspect subqueries of the universal plan U. By removing atoms from U before starting the backchase, we decrease the number of inspected subqueries. Treating the `GReX` atoms as interpreted symbols, we developed a few criteria for decreasing the size of the universal plan while guaranteeing that the optimal reformulation won't be missed.

1. Assume that the universal plan U contains both an atom $\texttt{child}(x,y)$ and an atom $\texttt{desc}(x,y)$. Clearly, by removing $\texttt{desc}(x,y)$ we preserve equivalence to the original query, and also we do no lose optimality of the reformulation, since in any reasonable cost model accessing the descendants of a node is at least as expensive as accessing its children. We therefore eliminate all `desc` atoms that are "parallel" to a chain of `child` and `desc` atoms. For the example chain (11), this means removing $\frac{n(n+1)}{2}$ `desc` atoms introduced in the chase phase, thus reducing the number of subqueries from $O(2^{n^2})$ to $O(2^n)$.

Criteria 2–3 prune subqueries that do not correspond to legal XQuery navigation.

2. Child and descendant navigation steps must be contiguous. For example, we won't consider the subquery induced by the atoms $\texttt{root}(x_1), \texttt{child}(x_2,x_3)$ of the chain (11) because it involves jumping directly from x_1 to some element x_2, instead of navigating there via the missing $\texttt{child}(x_1,x_2)$ atom. This pruning criterion reduces the search space size from

$O(2^{n+1})$ to $O(n^2)$ subqueries. Notice the drop from exponential to polynomial size.

3. Subsets of atoms that do not contain at least an atom describing the root of a document, or some other valid entry point into it, are not considered. there is no point considering the subquery induced by the atoms $\texttt{child}(x_1, x_2)$, $\texttt{child}(x_2, x_3)$ of the chain (11). This reduces the number of eligible subqueries further, from $O(n^2)$ to $O(n)$.

Of course, a naive implementation that first generates the subquery and then checks the criteria still results in exponential work. Instead, we eliminate the redundant `desc` atoms from criterion 1 by working directly on the universal plan. More interestingly, we avoid generating subqueries that violate criteria 2 and 3 by constructing a directed reachability graph of the atoms: the nodes represent atoms, and e.g. there is an edge from a `child` atom a_1 to a `desc` atom a_2 iff the second variable of a_1 coincides with the first of a_2. Similar rules are defined for all pairs of relational symbols from `GReX`. The roots of the graph are given by `root` atoms. We traverse this graph starting from the roots to generate the legal subsets of atoms.[2]

4 Experiments

The MARS system is implemented in JDK1.2, and the experiments were run on a Windows XP Dell 4450 desktop with a 2.4GHz PentiumIV and 1GB of RAM.

4.1 Synthetic Experimental Configuration

XML star queries. We designed the following class of XML reformulation problems, based on the XML equivalent of relational star queries.

The public schema describes `R` elements (children of the root element) with `K`-, `A1`,... `An`-, subelements. Similarly, for $1 \leq i \leq N_C$ there are S_i elements with subelements A and B. R's subelements A_i are foreign keys referencing subelement A in S_i, and K is a key for R (see [13] for the `XIC`s expressing these constraints).

The proprietary schema contains the public schema as well as redundantly materialized XML star views V_l ($1 \leq l \leq N_V$) which join the hub (R) with two if its corners (S_l and S_{l+1}). The joins are along the foreign keys, and the view projects on the B subelements of both corners, as well as R's subelement K. Note that there is considerable redundancy in the storage: view V_l redundantly stores the data from R and S_l, S_{l+1} elements, but there is also overlap among the views themselves: V_l and V_{l+1} for each l.

The query has a similar shape as the queries defining the views, but it joins R-elements with *all* N_C S_i elements. Notice that in the absence of integrity constraints, there is no reformulation using the views, but if we take into account the key constraint on R, then the star join can be rewritten using any subset of the views corresponding to it. There are hence 2^{N_V} possible reformulations, all of which are discovered by the C&B algorithm. Consequently, the universal plan will contain all N_V views, and the backchase stage will have to deal with 2^{N_V} possible reformulations.

4.2 Scalability Experiments

We study the behavior of MARS reformulation as the number of views increases in a situation that puts the backchase implementation under significant stress, forcing it to pick from exponentially many minimal reformulations.

The configuration we used to obtain such a worst-case scenario is the XML star query, in which we allowed $N_V = N_C - 1$ views per star:

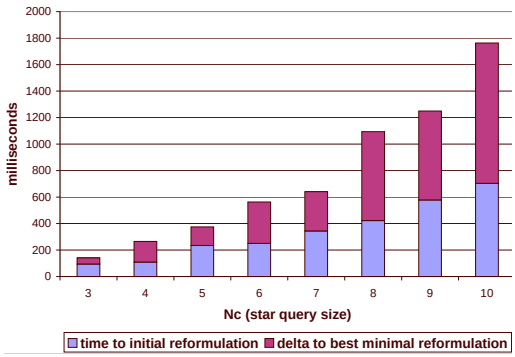

Figure 5: Scalability of Reformulation

We measured the time to find the initial reformulation, and the additional time to find the best minimal reformulation. It turns out that these are negligible compared to the execution times of leading XQuery engines for these queries, even on small documents. For example, for $N_C = 3$ and a toy document of 60 elements, the Galax [15] engine takes 1.5s to execute the query as is, and 128ms to execute the reformulation using views. The 141ms invested by MARS to find this reformulation hence gives a net saving of 1.331s. The net saving increases with increasing complexity of the query and views. For $N_C = 6$ and the same document, it reaches 20.532s, and for $N_C = 10$, it reaches 2 minutes (when spending 2s in reformulation). Larger document sizes only increase the net saving. We tried the same experiment with the Enosys [32] engine, obtaining net savings of the same order of magnitude.

More Experiments. The extended version [13] shows that the speedup of the new C&B implementation compared to the original one is at least two orders of magnitude. Furthermore, it presents a suite of experiments which study the behavior of the MARS system on more realistic queries and views, using a configuration based on the XMark benchmark [27]. The experiments show that MARS performs well within

[2]This technique is related to that of *sideways information passing* used in System R's optimizer [28] to avoid plans that compute Cartesian products, which are precisely the plans violating criterion (2) (such as $\texttt{root}(x_1), \texttt{child}(x_2, x_3)$).

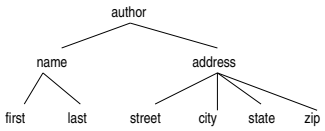

Figure 6: XML representation of `author` entities

feasible range for realistic size queries. The optimal reformulated queries (exploiting the redundancy we added to the schema) run very fast and it is well worth spending the time to find them (on average 350ms).

5 Schema Specialization

This technique taps a completely different source of potential improvements in MARS performance: using an application-specific schema as a part of the target for the XML-to-relational compilation. Its effect is to reduce the number of atoms in the query and constraints, thus obtaining a faster chase, a smaller universal plan, and consequently a faster backchase stage.

5.1 Exploiting Regularity in the Structure of Documents

The basic idea is to model a set of atoms corresponding to an XML navigation pattern by a single tuple of a virtual relation from a special schema. We will call the technique of associating this relational schema to the XML schema *specialization*. For example, if we know that the relatively complex XML tree pattern in Figure 6 represents `author` entities, we can model these as tuples of the schema

```
Author(id,pid,first,last,street,city,state,zip)
```

Note that the identities of internal nodes of the tree (`name, address`), as well as the parent-child relationships are abstracted away in this relational view. The `id` attribute holds the identity of the `author` nodes, and `pid` that of their parents. These are needed in translating queries against the XML schema to relational queries against the special schema. Now consider the `XBind` query returning the names of authors living in a city where a publisher is located:

$$\begin{aligned} Xb(l) \quad \leftarrow \quad & [//author](id), [./name/last/text()](id, l), \\ & [./address/city/text()](id, c), \\ & [//publisher/address/city/text()](c) \end{aligned}$$

Assume the existence of a relational materialized view associating author last names with the city they live in:[3]

$$\begin{aligned} V(l, c) \quad \leftarrow \quad & [//author](id), [./name/last/text()](id, l), \\ & [./address/city/text()](id, c) \end{aligned}$$

[3]For brevity, we give the view definition using an `XBind` query, rather than an XQuery outputting the XML encoding of the relation, as would be done in XPeranto.

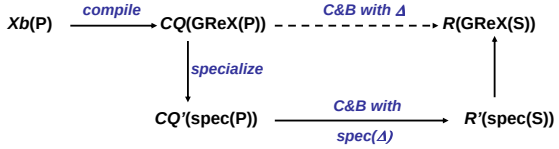

Figure 7: Reformulation by schema specialization

The C&B algorithm will find a reformulation of `Xb` using `V` by chasing the relational compilation of `Xb` (which we shall denote `c(Xb)`) with the constraints capturing `V`. Here is one of them:

$$\begin{aligned} \forall r \forall d \forall id \forall n \forall x \forall l \forall c \; & \texttt{root}(r) \wedge \texttt{desc}(r, d) \wedge \texttt{child}(d, id) \wedge \\ & \texttt{tag}(id, '' author'') \wedge \texttt{child}(id, n) \wedge \texttt{tag}(n, '' name'') \wedge \\ & \texttt{child}(n, x) \wedge \texttt{tag}(x, '' last'') \wedge \texttt{text}(x, l) \wedge \texttt{child}(id, a) \\ & \wedge \texttt{tag}(a, '' address'') \wedge \texttt{child}(a, y) \wedge \texttt{tag}(y, '' city'') \wedge \\ & \texttt{text}(y, c) \rightarrow V(l, c) \end{aligned} \quad (12)$$

The verbosity of this constraint makes for a relatively expensive chase step, since any such step involves finding a homomorphism from the premise of (12) into the body of `c(Xb)`, which is NP-hard in the size (number of atoms) of the premise. The premise size can be reduced by using the above relational specialization of `author` elements. (12) turns into (13) below, with 13 less atoms in the premise:

$$\begin{aligned} & \forall d \forall id \forall f \forall l \forall str \forall c \forall sta \forall z \\ & \qquad \texttt{Author}(id, d, f, l, str, c, sta, z) \rightarrow V(l, c) \end{aligned} \quad (13)$$

Similar concise translations hold for `c(Xb)`.

Figure 7 shows our specialization strategy. As before, the correspondence between proprietary schema S and public schema P is compiled to the set Δ of DEDs. Let $spec(P)$,$spec(S)$ denote the specializations of schemas P, S. We again start by compiling the `XBind` query Xb (formulated against P) to a relational conjunctive query CQ over schema $\texttt{GReX}(P)$, which is the generic relational encoding of XML schema P. However, instead of applying the C&B algorithm to CQ and Δ to directly obtain a reformulation `R`, we first *specialize* CQ, translating it to a query CQ' against $spec(P)$. Similarly, we specialize the DEDs in Δ, to obtain the more concise DEDs $spec(\Delta)$ defining the correspondence between schemas $spec(P)$ and $spec(S)$. We next reformulate CQ' under $spec(\Delta)$ to a query R' over $spec(S)$. Finally, R' is post-processed, which means substituting the relational specializations from $spec(S)$ with the original XML entities from S (encoded relationally in $\texttt{GReX}(S)$). Notice that, while the specialization of the query must be done on-line, that of the schemas P, S and constraints Δ can be done once and for all, off-line. For our example, `Author` belongs to schema $spec(P)$, CQ is $c(Xb)$, and its specialization is

$CQ'(l,c) \leftarrow$ `Author`(id,d,f,l,str,c,sta,z),
`root`(r), `desc`(r,d'), `child`(d',p), `tag`$(p,''publisher'')$,
`child`(p,a'), `tag`$(a',''address'')$, `child`(a',u),
`tag`$(u,''city'')$, `text`(u,c)

Notice how only the navigation of Xb pertaining to author data could be specialized, while that pertaining to publisher could not. The C&B reformulation of CQ' involves chasing it with constraint (13), (instead of chasing $c(Xb)$ with (12)). We obtain, among others, a reformulation using V.

Finding the schema specialization. The specializations can be specified explicitly by domain experts. A more desirable alternative is to infer them automatically, by detecting the parts of the XML document which are highly structured, and associating a relation to them. Tools solving exactly this task have been developed, albeit with a different motivation, namely that of *storing* XML data relationally. We can therefore adapt tools like STORED [7] or hybrid inlining [31]. The specialization in the above example would be found by hybrid inlining using a DTD or XML Schema associated with the document.

Specifying the specializations using mappings. Regardless of how the specializations are found, and especially if a domain expert defines them, we need a notation for describing the corresponding XML-to-relational mapping. This can be done using various syntaxes (see [13] for an example in XBind syntax).

Restrictions on the specialization mappings to ensure efficient specialization. Specializing a query means reformulating it according to the specialization mappings. This can be done in two ways. In one approach we could use independent techniques. For example, if the specialization mappings are the result of hybrid inlining, then the query reformulation algorithm from [31] could be used. A more desirable approach is to *reuse* the C&B algorithm. Since the specializations can be expressed as mappings, we can compile them to constraints and use the C&B reformulation to obtain the specialized queries as reformulations against the intermediate specialization schema I. However, if the specialization mappings are arbitrary, this approach shifts the complexity from the C&B reformulation to the specialization step itself, thus defeating its purpose. We identified reasonable restrictions on the expressive power of specialization mappings which lead to polynomial running time of the C&B when used for specialization (see [13]).

Proposition 5.1 *Given a restricted specialization mapping, there is a compilation of this mapping to a set of constraints ρ such that the C&B algorithm applied with ρ reformulates any* `XBind` *query in PTIME in its size.*

The restrictions still allow interesting specialization mappings. It turns out that the specialization mapping we would obtain by hybrid inlining satisfies them (space does not allow us to show details). This suggests borrowing the hybrid inlining technique (developed for *storing* the XML data) in order to *specialize* XML queries and integrity constraints.

Corollary 5.2 *If the specialization mappings correspond to the mapping discovered by hybrid inlining, the specialization step can be performed in PTIME in the size of the query using the C&B algorithm itself.*

5.2 Improvements due to Specialization

In this experiment, we use the same family of public schema and query as introduced in the XML star query scenario. However, the proprietary schema contains only the views now, and we measure the ratio of the times to find all reformulations without and with schema specialization. This ratio is again measured as a function of the N_C parameter, shown in Figure 8. Note the exponential increase of the benefit. The ben-

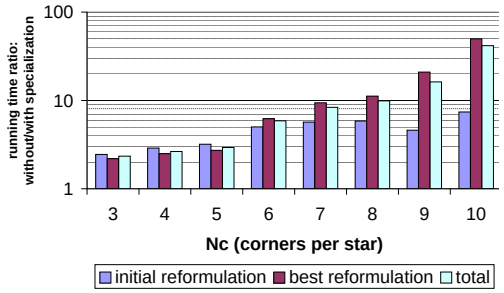

Figure 8: Effect of schema specialization

efit is broken down for (1) the time to obtain the initial reformulation using all views, (2) the time to perform backchase minimization on this reformulation, to obtain all minimal reformulations, and (3) the sum of these times. Recall from Section 2.3 that the initial reformulation obtained in (1) is computed by first chasing the query to the universal plan, and then picking the largest subquery that mentions only proprietary schema elements (in this case views). To see that the optimization time is negligible when compared to the time to execute the unoptimized query, note that the actual running time for the C&B with specialization is 1.1sec for $N_C = 9$, while the running time for the unoptimized query is over 10 minutes.

6 Further Work

Of course, there are more relational evaluation / optimization techniques than the ones we have used to speed up the implementation of the chase. We are particularly interested in applying multi-query optimization techniques, given that during chasing we need to

evaluate in parallel all queries corresponding to constraints. We would like to investigate the applicability of the reformulation techniques presented here for XQuery optimizers of the future. Also, we plan to exploit the completeness of MARS reformulation to create a testbed for various cost models and XQuery optimization heuristics. Cost models for XQuery are still under research, and in order to check their performance, we need to have a good understanding of what reformulations they discard and how close they come to the optimal one. This can be achieved only by enumerating all reformulations.

Acknowledgements This work owes much to our prior collaboration with Lucian Popa. We are indebted to Peter Buneman, Susan Davidson, Mary Fernandez, Dan Suciu and Scott Weinstein for extensive and valuable feedback on this work.

References

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

[2] A. Aboulnaga, A. Alameldeen, and J. F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *VLDB*, 2001.

[3] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[4] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, 2002.

[5] M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Middleware For Publishing Object-Relational Data as XML. *VLDB*, 2000.

[6] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, Raymond T. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *ICDE*, 2001.

[7] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *SIGMOD*, 1999.

[8] A. Deutsch, L. Popa, and V. Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *VLDB*,1999.

[9] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath fragments. In *KRDB*, 2001.

[10] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *DBPL*, 2001.

[11] A. Deutsch and V. Tannen. Reformulation of xml queries and constraints. In *ICDT*, 2003.

[12] A. Deutsch and V. Tannen. XML queries and constraints, containment and reformulation. To appear in *JTCS*, 2003.

[13] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage (extended version). `http://db.ucsd.edu/people/alin`.

[14] M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middle-ware Queries. In *SIGMOD*'01.

[15] M. Fernandez, J. Simeon, P. Wadler, and B. Choi. Galax. `http://db.bell-labs.com/galax`.

[16] M. Fernandez, W.-C. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. In *WWW9*, 2000.

[17] T. Fiebig and G. Moerkotte. Evaluating queries on structure with extended access support relations. In *WebDB*, 2000.

[18] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Simon. Statix: making xml count. In *ICDE*. 2002.

[19] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *AAAI/IAAI*, p. 67–73,1999.

[20] Alon Halevy. Logic-based techniques in data integration. In *Logic Based Artificial Intelligence*, 2000.

[21] H.V.Jagadish, S.Al-Khalifa, A.Chapman, L.V.S.Lakshmanan, A.Nierman, S.Paparizos, J.Patel, D.Srivastava, N.Wiwatwattana, Y.Wu, and C.Yu. Timber:a native xml database. *VLDB Journal*, 11(4), 2002.

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

[23] I. Manolescu, D. Florescu, and D. Kossman. Answering XML Queries on Heterogeneous Data Sources. *VLDB*'01.

[24] G. Moerkotte and C.-C. Kanne. Efficient storage of xml data. In *ICDE*, 2000.

[25] L. Popa. *Object/Relational Query Optimization with Chase and Backchase*. PhD thesis, Univ. of Pennsylvania, 2000.

[26] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A Chase Too Far? In *SIGMOD*, May 2000.

[27] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, 2002.

[28] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[29] P. Seshadri, H. Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In *ICDE*, 1996.

[30] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB*, 2001.

[31] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents. In *VLDB*, 1999.

[32] Enosys Software. http://www.enosyssoftware.com.

[33] W3C. XML Path Language (XPath) 1.0. W3C Recommendation. `http://www.w3.org/TR/xpath`.

[34] W3C. XQuery: A query Language for XML. W3C Working Draft. `http://www.w3.org/TR/xquery`.

[35] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for xml queries. In *EDBT*, 2002.