Searching for Efficient XML-to-Relational Mappings

Maya Ramanath[1], Juliana Freire[2], Jayant R. Haritsa[1], and Prasan Roy[3]

[1] SERC, Indian Institute of Science
{maya,haritsa}@dsl.serc.iisc.ernet.in
[2] OGI/OHSU
juliana@cse.ogi.edu
[3] Indian Institute of Technology, Bombay
prasan@it.iitb.ac.in

Abstract. We consider the problem of cost-based strategies to derive efficient relational configurations for XML applications that subscribe to an XML Schema. In particular, we propose a flexible framework for XML schema transformations and show how it can be used to design algorithms to search the space of equivalent relational configurations. We study the impact of the schema transformations and query workload on the search strategies for finding efficient XML-to-relational mappings. In addition, we propose several optimizations to speed up the search process. Our experiments indicate that a judicious choice of transformations and search strategies can lead to relational configurations of substantially higher quality than those recommended by previous approaches.

1 Introduction

XML has become an extremely popular medium for representing and exchanging information. As a result, efficient storage of XML documents is now an active area of research in the database community. In particular, the use of relational engines for this purpose has attracted considerable interest with a view to leveraging their powerful and reliable data management services.

Cost-based strategies to derive relational configurations for XML applications have been proposed recently [1, 19] and shown to provide substantially better configurations than heuristic methods (*e.g.,* [15]). The general methodology used in these strategies is to define a set of *XML schema transformations* that derive different relational configurations. Given an XML query workload, the quality of the relational configuration is evaluated by a costing function on the SQL equivalents of the XML queries. Since the search space is large, greedy heuristics are used to search through the associated space of relational configurations.

In this paper, we propose **FlexMap**, a framework for generating XML-to-relational mappings which incorporates a comprehensive set of schema transformations. FlexMap is capable of supporting different mapping schemes such as ordered XML and schema-less content. Our framework represents the XML Schema [2] through type constructors and uses this representation to define several schema transformations from the existing literature. We also propose several new transformations and more powerful variations of existing ones. We utilize this framework to study, for the first time, the impact of

schema transformations and the query workload on *search strategies* for finding efficient XML-to-relational mappings.

We have incorporated FlexMap in the LegoDB prototype [1]. We show that the space of possible configurations is large enough to remove the possibility of exhaustive search even for small XML schemas. We describe a series of greedy algorithms which differ in the number and type of transformations that they utilize, and show how the choice of transformations impacts the search space of configurations. Intuitively, the size of the search space examined increases as the number/types of transformations considered in the algorithms increase. Our empirical results demonstrate that, in addition to deriving better quality configurations, algorithms that search a larger space of configurations can sometimes (counter-intuitively) converge *faster*. Further, we propose optimizations that significantly speed up the search process with very little loss in the quality of the selected relational configuration.

In summary, our contributions are:

1. A framework for exploring the space of XML-to-relational mappings.
2. More powerful variants of existing transformations and their use in search algorithms.
3. A study of the impact of schema transformations and the query workload on search algorithms in terms of the quality of the final configuration as well as the time taken by the algorithm to converge.
4. Optimizations to speed up these algorithms.

Organization Section 2 develops the framework for XML-to-relational mappings. Section 3 proposes three different search algorithms based on greedy heuristics. Section 4 evaluates the search algorithms and Section 5 discusses several optimizations to reduce the search time. Section 6 discusses related work and Section 7 summarizes our results and identifies directions for future research.

2 Framework for Schema Transformations

2.1 Schema Tree

We define a *schema tree* to represent the XML schema in terms of the following type constructors: *sequence* (","), *repetition* ("$*$"), *option* ("?"), *union* ("|"), $<$*tagname*$>$ (corresponding to a tag) and $<$*simple type*$>$ corresponding to base types (*e.g.,* integer). Figure 1 gives a grammar for the schema tree. The schema tree is an ordered tree where tags appear in the same order as the corresponding XML schema.

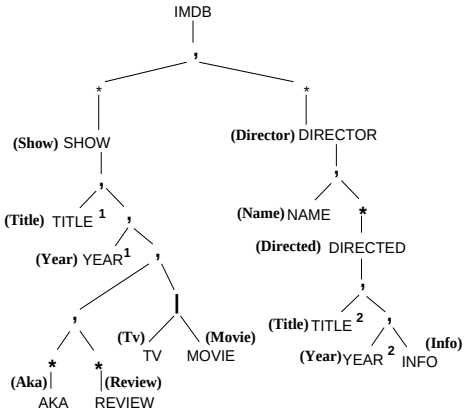
As an example, consider the (partial) XML Schema of the IMDB (Internet Movie DataBase) website [8] in Figure 2(a). Here, **Title**, **Year**, **Aka** and **Review** are simple types. The schema tree for this schema is shown in Figure 2(b) with base types ommitted and tags represented in normal font. Nodes in the tree are *annotated* with the *names* of the types present in the original schema – these annotations are shown in boldface and paranthesized next to the tags. Note that, first, there need not be any correspondence between tag names and annotations (type names). Second, the schema graph is represented as a tree, where different occurrences of equivalent nodes are captured, but

```
<complex type> ::=
        <simple type>
        || <complex type> , <complex type>
        || <complex type> | <complex type>
        || <complex type> *
        || <complex type> ?
        || <tagname> [<complex type>]
```

Fig. 1. Using Type Constructors to Represent XML schema Types

```
define element IMDB {
    type Show*, type Director*, type Actor*
}
define type Show {element SHOW {
    type Title, type Year, type Aka*,
    type Review*, (type Tv | type Movie) }}
define type Director {element DIRECTOR {
    type Name, type Directed*}}
define type Directed {element DIRECTED {
    type Title, type Year, type Info }}
```

(a) The (partial) IMDB Schema



(b) Tree Representation

Fig. 2. The IMDB Schema

their content is assumed to be shared (see *e.g.*, the nodes TITLE^1 and TITLE^2 in Figure 2(b)). Finally, recursive types can be handled similarly to *shared* types, *i.e.*, the base occurrence and the recursive occurrences are differentiated, but share the same content.

Note that any subtree in the schema tree is a type regardless of whether it is annotated. We refer to annotations as the *name* of the node and use it synonymously with annotation. We also use the terms subtree, node and type interchangeably in the remainder of the paper.

2.2 From Schema Trees to Relational Configurations

Given a schema tree, a relational configuration is derived as follows:

1. If N is the annotation of a node, then there is a relational table T_N corresponding to it. This table contains a *key* column and a *parent_id* column which points to the key column of the table corresponding to the *closest named ancestor* of the current node if it exists. The key column consists of ids assigned specifically for

Table **Director**	[director_key]
Table **Name**	[Name_key, NAME, parent_director_id]
Table **Directed**	[Directed_key, parent_director_id]
Table **Title**	[Title_key, TITLE, parent_directed_show_id]
Table **Year**	[Year_key, YEAR, parent_directed_show_id]
Table **Info**	[Info_key, INFO, parent_directed_id]

Fig. 3. Relational Schema for the **Director** Subtree

the purpose of identifying each tuple (and by extension, the corresponding node of the XML document).

2. If the subtree of the node annotated by N is a simple type, then T_N additionally contains a column corresponding to that type to store its values.
3. If N is the annotation of a node, and no descendant of N is annotated, then T_N contains as many additional columns as the number of descendants of N that are simple types.

Other rules which may help in deriving effi cient schemas could include storing repeated types and types which are part of a union in separate tables. The relational confi guration corresponding to the naming in Figure 2(b) for the **Director** subtree is shown in Figure 3. Since **Title** and **Year** are shared by **Show** and **Directed**, their parent id columns contain ids from both the **Show** and **Directed** tables. Note that the mapping can follow rules different from the ones listed above (for example, storing types which are part of unions in the same table by utilizing null values).

It is possible to support different mapping schemes as well – for example, in order to support ordered XML, one or more additional columns have to be incorporated into the relational table [16]. By augmenting the type constructors, it is also possible to support a combination of different mapping schemes. For example, by introducing an ANYTYPE constructor, we can defi ne a rule mapping annotated nodes of that type to a ternary relation (edge table) [5].

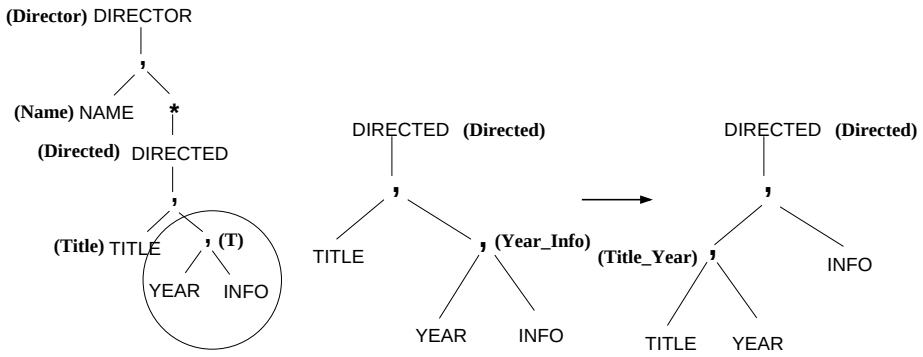
2.3 Schema Transformations

Before we describe the schema transformations we introduce a compact notation to describe the type constructors, and using this notation, we defi ne the notion of *syntactic equality* between two subtrees. In the following, t_i and t are subtrees and a is the annotation.

Tag Constructor: $E(label, t, a)$, where $label$ is the name of the tag (such as TITLE, YEAR, etc.)

Sequence, Union, Option and Repetition Constructors: The constructors are defi ned as: $C(t_1, t_2, a)$, $U(t_1, t_2, a)$, $O(t, a)$, and $R(t, a)$, respectively.

Simple Type Constructor: Simple types are represented as $S(base, a)$ where $base$ is the base type (*e.g.,* integer).



(a) Introducing Annotation *T*

(b) Applying Associativity

Fig. 4. Grouping Elements Using Annotations

Definition 1. Syntactic Equality *Two types* T_1 *and* T_2 *are syntactically equal – denoted by* $T_1 \cong T_2$ *– if the following holds:*

case T_1, T_2 *of*

$| E(label, t, a),\ E(label', t', a') \rightarrow label = label'$ *AND* $a = a'$ *AND* $t \cong t'$

$| C(t_1, t_2, a),\ C(t_1', t_2', a') \rightarrow a = a'$ *AND* $t_1 \cong t_1'$ *AND* $t_2 \cong t_2'$

$| U(t_1, t_2, a),\ U(t_1', t_2', a') \rightarrow a = a'$ *AND* $t_1 \cong t_1'$ *AND* $t_2 \cong t_2'$

$| R(t, a),\ R(t', a') \rightarrow a = a'$ *AND* $t \cong t'$

$| O(t, a),\ O(t', a') \rightarrow a = a'$ *AND* $t \cong t'$

$| S(b, a),\ S(b', a') \rightarrow a = a'$ *AND* $b = b'$

Inline and Outline An annotated node in the schema tree is *Outlined* and has a separate relational table associated with it. All nodes which do not have an annotation associated with them are *Inlined*, *i.e.,* form part of an existing table. We can outline or inline a type by respectively annotating a node or removing its annotation. Inline and outline can be used to group elements together as shown in Figure 4(a) where a new annotation T is introduced. The corresponding relational configuration will "group" **Year** and **Info** into a new table **T**.

Type Split/Merge The inline and outline operations are analogous to removing and adding annotations to nodes. *Type Split* and *Type Merge* are based on the *renaming* of nodes. We refer to a type as *shared* when it has distinct annotated parents. In the example shown in Figure 2(b), the type **Title** is shared by the types **Show** and **Directed**.

Intuitively, the type split operation distinguishes between two occurrences of a type by renaming the occurrences. By renaming the type **Title** to **STitle** and **DTitle**, a relational configuration is derived where a separate table is created for each title. Conversely, the type merge operation adds identical annotations to types whose *corresponding subtrees* are syntactically equal.

So far, we have defined transformations which do not change the structure of the schema tree. Using only these transformations, the number of derived relational configurations is already exponential in the number of nodes in the tree. That is, since any subset of nodes in the schema tree can be named (subject to the constraints of having to name certain nodes which should have separate relations of their own as outlined in Section 2.2), and the corresponding relational configuration derived, for a schema tree with N nodes, we have a maximum of 2^N possible relational configurations.

We next define several transformations which do change the structure of the schema tree which in turn leads to a further expansion in the search space. Some of these transforms were originally described in our earlier work [1]. We redefine and extend these transforms in our new framework.

Commutativity and Associativity Two basic structure-altering operations that we consider are: *commutativity* and *associativity*. Associativity is used to *group* different types into the same relational table. For example, in the first tree of Figure 4(b), Year and Info about the type **Directed** are stored in a single table called **Year_Info**, while after applying associativity as shown in the second tree, Title and Year appear in a single table called **Title_Year**.

Commutativity by itself does not give rise to different relational mappings[4], but when combined with associativity may generate mappings different from those considered in the existing literature. For example, in Figure 4(b), by first commuting Year and Info and then applying associativity, we can get a configuration in which Title and Info are stored in the same relation.

Union Distribution/Factorization Using the standard distribution law for distributing sequences over unions for regular expressions, we can separate out components of a union: $(a,(b|c)) \equiv (a,b)|(a,c)$. We derive useful configurations using a combination of union distribution, outline and type split as shown below:

define type **Show** { element SHOW { type **Title**, (type **Tv**|type **Movie**) }}

Distribute Union $\rightarrow$ *Outline* $\rightarrow$ *Type split* $Title \rightarrow$

define type **TVShow** { element SHOW { type **TVTitle**, type **Tv** }}
define type **MovieShow** { element SHOW { type **MovieTitle**, type **Movie** }}

The relational configuration corresponding to the above schema has separate tables for **TVShow** and **MovieShow**, as well as for **TVTitle** and **MovieTitle**. Moreover, applying this transformation enables the inlining of **TVTitle** into **TVShow** and **MovieTitle** into **MovieShow**. Thus the information about TV shows and movie shows is separated out (this is equivalent to horizontally partitioning the **Show** table). Conversely, the union factorization transform would factorize a union.

In order to determine whether there is potential for a union distribution, one of the patterns we search the schema tree is: C(X,U(Y,Z)) and transform it to U(C(X,Y),C(X,Z)).

[4] Commuting the children of a node no longer retains the original order of the XML schema.

We have to determine the syntactic equality of two subtrees before declaring the pattern to be a candidate for union factorization. Note that there are several other conditions under which union distribution and factorization can be done [11].

Other transforms, such as *splitting/merging repetitions* [1], *simplifying unions* [15] (a lossy transform which could enable the inlining of one or more components of the union), etc. can be defined similarly to the transformations described above.

We refer to Type Merge, Union Factorization and Repetition Merge as *merge transforms* and Type Split, Union Distribution and Repetition Split as *split transforms* in the remainder of this paper.

2.4 Evaluating Configurations

As transformations are applied and new configurations derived, it is important that precise cost estimates be computed for the query workload under each of the derived configurations – which, in turn, requires accurate statistics. Since it is not practical to scan the base data for each relational configuration derived, it is crucial that these statistics be accurately propagated as transformations are applied.

An important observation about the transformations defined in this section is that while merge operations preserve the accuracy of statistics, split operations do not [6]. Hence, in order to preserve the accuracy of the statistics, before the search procedure starts, *all* possible split operations are applied to the user-given XML schema. Statistics are then collected for this *fully decomposed* schema. Subsequently, during the search process, only merge operations are considered.

In our prototype, we use StatiX [6] to collect statistics and to accurately propagate them into the relational domain. The configurations derived are evaluated by a relational optimizer [12]. Our optimizer assumes a primary index on the key column of each relational table. For lack of space, we do not elaborate on these issues. More information can be found in [11].

3 Search Algorithms

In this section we describe a suite of greedy algorithms we have implemented within FlexMap. They differ in the choice of transformations that are selected and applied at each iteration of the search.

First, consider Algorithm 1 that describes a simple greedy algorithm – similar to the algorithm described in [1]. It takes as input a query workload and the initial schema (with statistics). At each iteration, the transform which results in the minimum cost relational configuration is chosen and applied to the schema (lines 5 through 19). The translation of the transformed schema to the relational configuration (line 11) follows the rules set out in Section 2.2. The algorithm terminates when no transform can be found which reduces the cost.

Though this algorithm is simple, it can be made very flexible. This flexibility is achieved by varying the strategies to select applicable transformations at each iteration (function applicableTransforms in line 8). In the experiments described in [1], only inline and outline were considered as the applicable transformations. The utility of the

Algorithm 1 Greedy Algorithm

```
1: Input: queryWkld, S {Query workload and Initial Schema}
2: prevMinCost ← INF
3: rel_schema ← convertToRelConfig(S, queryWkld)
4: minCost ← COST(rel_schema)
5: while minCost < prevMinCost do
6:    S' ← S {Make a copy of the schema}
7:    prevMinCost ← minCost
8:    transforms ← applicableTransforms(S')
9:    for all T in transforms do
10:      S'' ← Apply T to S' {S' is preserved without change}
11:      rel_schema ← convertToRelConfig(S'', queryWkld)
12:      Cost ← COST(rel_schema)
13:      if Cost < minCost then
14:         minCost ← Cost
15:         minTransform ← T
16:      end if
17:   end for
18:   S ← Apply minTransform to S {The min. cost transform is applied}
19: end while
20: return convertToRelConfig(S)
```

other transformations (*e.g.*, union distribution and repetition split) were shown qualitatively. Below, we describe variations to the basic greedy algorithm that allow for a richer set of transformations. All search algorithms described use the fully decomposed schema as the start point, and only merge operations are applied during the greedy iterations.

3.1 InlineGreedy

The first variation we consider is *InlineGreedy* (IG), which only allows the inline transform. Note that IG differs from the algorithm experimentally evaluated in [1], which we term *InlineUser* (IU), in the choice of starting schema: IG starts with the fully decomposed schema whereas InlineUser starts with the original user schema.

3.2 ShallowGreedy: Adding Transforms

The *ShallowGreedy* (SG) algorithm defines the function applicableTransforms to return *all* the applicable merge transforms. Because it follows the transformation dependencies that result from the notion of syntactic equality (see Definition 1), it only performs single-level or *shallow* merges.

The notion of syntactic equality, however, can be too restrictive and as a result, SG may miss efficient configurations. For example consider the following (partial) IMDB schema:

define type **Show** {type **Show1** | type **Show2**}
define type **Show1** {element SHOW { type **Title1**, type **Year1**, type **Tv** }}
define type **Show2** {element SHOW { type **Title2**, type **Year2**, type **Movie** }}

Unless a type merge of **Title1** and **Title2** and a type merge of **Year1** and **Year2** take place, we cannot factorize the union of **Show1** | **Show2**. However, in a run of SG, these two type merges by themselves may not reduce the cost, whereas taken in conjunction with union factorization, they may lead to a substantial cost reduction. Therefore, SG is handicapped by the fact that a union factorization will only be applied if both type merges are independently chosen by the algorithm. In order to overcome this problem, we design a new algorithm called *DeepGreedy* (DG).

3.3 DeepGreedy: *Deep* merges

Before we proceed to describe the DG algorithm, we first introduce the notions of *Valid Transforms* and *Logical Equivalence*. A *valid transform* is an element of the set of all *applicable* transforms, S. Let V be a set of valid transforms.

Definition 2. Logical Equivalence *Two types T_1 and T_2 are* logically equivalent *under the set V of valid transforms, denoted by $T_1 \sim_V T_2$, if they can be made syntactically equal after applying a sequence of valid transforms from V.*

The following example illustrates this concept. Let $V = \{Inline\}; t_1 := E(\mathsf{TITLE}, S(string, -), \mathbf{Title_1})$, and $t_2 := E(\mathsf{TITLE}, S(string, -), \mathbf{Title_2})$. Note that t_1 and t_2 are not syntactically equal since their annotations do not match. However, they are *logically equivalent*: by *inlining* them (*i.e.*, removing the annotations $Title_1$ and $Title_2$), they can be made syntactically equal. Thus, we say that t_1 and t_2 are logically equivalent under the set $\{Inline\}$.

Now, consider two types T_i and T_j where $T_i := E(l, t_1, a_1)$ and $T_j := E(l, t_2, a_2)$ with t_1 and t_2 as defined above. Under syntactic equality, T_i and T_j would not be identified as candidates for type merge. However, if we relax the criteria to logical equivalence with (say) $V = \{TypeMerge\}$, then it is possible to identify the *potential* type merge of T_i and T_j. Thus, several transforms which may never be considered by SG can be identified as candidates by DG, provided the necessary operations can be fired (like the type merge of t_1 and t_2 in the above example) to *enable* the transform. Extending the above concept, we can enlarge the set of valid transforms V to contain all the merge transforms which can be fired recursively to *enable* other transforms.

Algorithm DG allows the same transforms as SG, *except* that potential transforms are identified not by direct syntactic equality, but by logical equivalence with the set of valid transforms containing all the merge operations (including inline). This allows DG to perform *deep* merges. Additional variations of the search algorithms are possible, *e.g.*, by restricting the set of valid transforms. But, they are not covered in this paper.

4 Performance Evaluation

In this section we present a performance evaluation of the three algorithms proposed in this paper: *InlineGreedy* (IG), *ShallowGreedy* (SG) and *DeepGreedy* (DG). We used a

synthetically generated subset of the IMDB dataset (≈60MB) for the experiments. The user schema consisted of 22 types, with 2 unions, 3 repetitions and 2 shared types. We describe the query workloads used next. For a more detailed discussion on the schema, dataset and the query workloads, please refer to [11].

4.1 Query Workloads

A query workload consists of a set of queries with a weight (in the range of 0 to 1) assigned to each query. These weights reflect the relative importance of the queries in some way (for example, the query with the largest weight might be the most frequent). We evaluated each of the algorithms on several query workloads based on (1) the quality of the derived relational configuration in terms of the cost for executing the query workload, and (2) the efficiency of the search algorithm measured in terms of the time taken by the algorithm. These are the same metrics as those used in [1]. Note that the latter is proportional to the number of distinct configurations seen by the algorithm, and also the number of distinct optimizer invocations since each iteration involves constructing a new configuration and evaluating its cost using the optimizer.

From the discussion of the proposed algorithms in Section 3, notice that the behavior of each algorithm (which runs on the fully decomposed schema) on a given query depends upon whether the query benefits more from merge transformations or split transformations. If the query benefits more from split, then neither DG nor SG is expected to perform better than IG.

As such, we considered the following two kinds of queries: **S-Queries** which are expected to derive benefit from split transformations (Type Split, Union Distribution and Repetition Split), and **M-Queries** which are expected to derive benefit from merge operations (Type Merge, Union Factorization and Repetition Merge).

S-Queries typically involve simple lookups. For example:

```
SQ1:  for $i in /IMDB/SHOW          SQ2:   for $i in /IMDB/DIRECTOR
      where $i/TV/CHANNEL = 9               where $i/DIRECTED/YEAR = 1994
      return $i/TITLE                       return $i/NAME
```

The query SQ1 is specific about the Title that it wants. Hence it would benefit from a type split of **Title**. Moreover, it also specifies that TV Titles only are to be returned, not merely Show Titles. Hence a union distribution would be useful to isolate only TV Titles. Similarly, query SQ2 would benefit from isolating Director Names from Actor Names and Directed Year from all other Years. Such splits would help make the corresponding tables smaller and hence lookup queries such as the above faster. Note that in the example queries above, both the predicate as well as the return value benefit from splits.

M-queries typically query for subtrees in the schema which are *high up* in the schema tree. When a split operation is performed on a type in the schema, it propagates downwards towards the descendants. For example, a union distribution of **Show** would result in a type split of **Review**, which would in turn lead to the type split of **Review**'s children. Hence queries which retrieve subtrees near the top of the schema tree would benefit from merge transforms. Similarly predicates which are high up in the tree would also benefit from merges. For example:

```
MQ1: for $i in /IMDB/SHOW,
         $j in $i/REVIEW
     return $i/TITLE, $i/YEAR, $i/AKA,
            $j/GRADE, $j/SOURCE,
            $j/COMMENTS
```

```
MQ2: for $i in /IMDB/ACTOR,
         $j in /IMDB/SHOW
     where $i/PLAYED/TITLE = $j/TITLE
     return $j/TITLE, $j/YEAR, $j/AKA,
            $j/REVIEW/SOURCE, $j/REVIEW/GRADE,
            $j/REVIEW/COMMENTS, $i/NAME
```

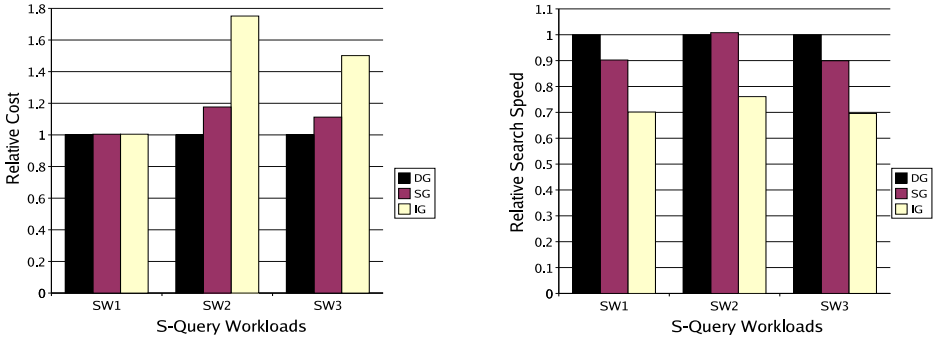
Query MQ1 asks for full details of a Show without distinguishing between TV Shows and Movie Shows. Since all attributes of Show which are *common* for TV as well as Movie Shows are requested, this query is likely to benefit from reduced fragmentation, *i.e.*, from union factorization and repetition merge. For example, a union factorization would enable some types like **Title** and **Year** to be inlined into the same table (the table corresponding to Show). Similarly, query MQ2 would benefit from a union factorization of **Show** as well as a repetition merge of **Played** (this is because the query does not distinguish between the Titles of the first Played and the remaining Played). In both the above queries, return values as well as predicates benefit from merge transformations.

Based on the two classes of queries described above and some of their variations, we constructed the following six workloads. Note that each workload consists of a set of queries as well as the associated weights. Unless stated otherwise, all queries in a workload are assigned equal weights and the weights sum up to 1.
SW1: contains 5 distinct S-queries, where the return values as well as predicates benefit from split transforms.
SW2: contains 5 distinct S-queries, with multiple return values which do not benefit from split, but predicates which benefit from split.
SW3: contains 10 S-queries with queries which have: i) return values as well as predicates benefitting from split and ii) only return values benefitting from split.
MW1: contains a single query which benefits from merge transforms.
MW2: contains the same single query as in MW1, but with selective predicates.
MW3: contains 8 queries which are M-Queries as well as M-Queries with selective predicates.

The performance of the proposed algorithms on S-query workloads (SW1-3) and M-query workloads (MW1-3) is studied in Sections 4.2 and 4.3, respectively.

There are many queries which cannot be conclusively classified as either an S-query or an M-query. For example, an interesting variation of S-Queries is when the query contains return values which do not benefit from split, but has predicates which do (SW2). For M-Queries, adding highly selective predicates, may reduce the utility of merge transforms. For example, adding the highly selective predicate $YEAR > 1990$ (Year ranges from 1900 to 2000) to query MW1 would reduce the number of tuples.

We study workloads where the two types of transformations conflict in Section 4.4. Arbitrary queries are unlikely to give much insight because the impact of split transformations vs. merge transformations would be different for different queries. Hence, we chose to work with a *mix* of S- and M-queries where the impact of split and merge transformations is controlled using a parameter. Finally, in Section 4.5 we demonstrate the competitiveness of our algorithms against certain baselines.



(a) Final Configuration Quality

(b) Efficiency of the Algorithms

Fig. 5. Performance of Workloads containing S-Queries

4.2 Performance on S-Query Workloads

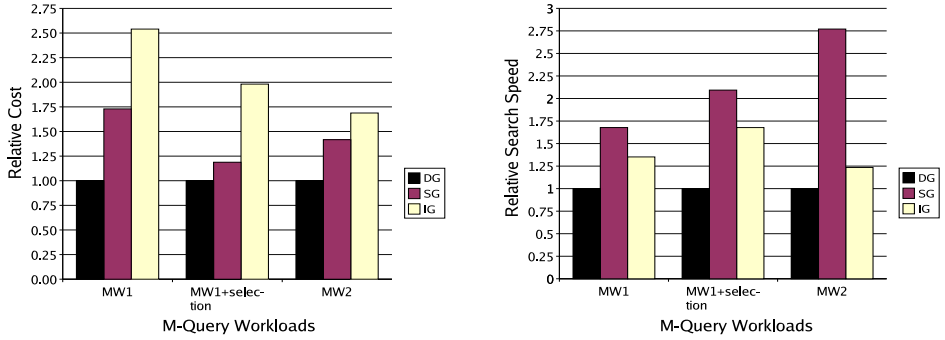
Recall that DG does "deep" merges, SG does "shallow" merges and IG allows only inlinings. S-Queries do not fully exploit the potential of DG since they do not benefit from too many merge transformations. So, DG can possibly consider transformations which are useless, making it more inefficient – *i.e.*, longer run times without any major advantages in the cost of the derived schema. We present results for the 3 workloads: SW1, SW2 and SW3.

As shown in Figure 5(a), the cost difference of the configurations derived by DG and SG is less than 1% for SW1, whereas SG and IG lead to configurations of the same cost for SW1. This is because of the fact that all queries of SW1 benefit mainly from split transforms – in effect, DG hardly has an advantage over SG or IG. But for SW2, the cost difference between DG and SG jumped up to around 17% – this is due to the return values benefiting from merge which gives DG an advantage over SG because of the larger variety of merge transforms it can consider. The difference between DG and IG was around 48%, expectedly so, since even the merge transforms considered by SG were not considered in IG.

The relative number of configurations examined by each of DG, SG and IG are shown in Figure 5(b). In terms of the number of relational configurations examined, DG searches through a much larger set of configurations than SG, while SG examines more configurations than IG. DG is especially inefficient for SW1 where it considers about 30% more configurations than IG for less than 1% improvement in cost.

4.3 Performance on M-Query Workloads

Figure 6(a) shows the relative costs of the 3 algorithms for the 3 workloads, MW1, MW2 and MW3. As expected DG performs extremely well compared to SG and IG since DG is capable of performing deep merges which benefit MW1. Note that the



(a) Final Configuration Quality

(b) Efficiency of the Algorithms

Fig. 6. Performance of Workloads containing M-Queries

effect of adding selective predicates reduces the magnitude of difference in the costs between DG, SG and IG.

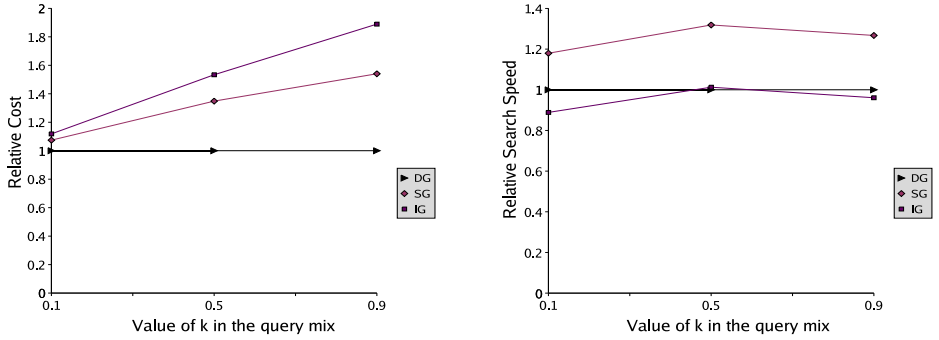
In terms of the number of configurations examined also, DG performed *the best* as compared to SG and IG. This would seem counter-intuitive – we would expect that since DG is capable of examining a superset of transformations as compared to SG and IG, it would take longer to converge. However, this did not turn out to be the case since DG picked up the cost saving recursive merges (such as union factorization) fairly early on in the run of DG which reduced the number of lower level merge and inline candidates in the subsequent iterations. This enabled DG to converge faster. By the same token, we would expect SG to examine fewer configurations than IG, but that was not the case. This is because SG was not able to perform any major cost saving merges since the "enabling" merges were never chosen individually (note the cost difference between DG and SG). Hence, the same set of merge transforms were being examined in every iteration without any benefit, while IG was not burdened with these candidate merges. But note that even though IG converges faster, it is mainly due to the lack of useful inlines as reflected by the cost difference between IG and SG.

4.4 Performance on Controlled S-Query and M-Query Mixed Workloads

The previous discussion highlighted the strengths and weaknesses of each algorithm. In summary, if the query workload consists of "pure" S-Queries, then IG is the best algorithm to run since it returns a configuration with marginal difference in cost compared to DG and in less time (reflected in the results for SW1), while if the query workload consists of M-Queries, then DG is the best algorithm to run.

In order to study the behaviour of mixed workloads, we used a workload (named MSW1) containing 11 queries (4 M-Queries and 7 S-Queries).

In order to control the dominance of S-queries vs. M-queries in the workload, we use a control parameter $k \in [0, 1]$ and give weight $(1 - k)/7$ to each of the 7 S-queries and



(a) Final Configuration Quality

(b) Efficiency of the algorithms

Fig. 7. Performance of Workloads containing both M- and S-Queries

weight $k/4$ to each of the 4 M-queries. We ran workload MSW1 with 3 different values of k ={0.1, 0.5, 0.9}. The cost of the derived configurations for MSW1 are shown in Figure 7(a). Expectedly, when S-Queries dominate, IG performs quite competitively with DG (with the cost of IG being within just 15% of DG). But, as the influence of S-Queries reduce, the difference in costs increases substantially.

The number of configurations examined by all three algorithms are shown in Figure 7(b). DG examines more configurations than IG when S-Queries dominate, but the gap is almost closed for the other cases. Note that both SG and IG examine more configurations for k = 0.5 than in the other two cases. This is due to the fact that when S-Queries dominate (k = 0.1), cost-saving inlines are chosen earlier; while when M-queries dominate (k = 0.9), both algorithms soon run out of cost-saving transformations to apply. Hence for both these cases, the algorithms converge faster.

4.5 Comparison with Baselines

From the above sections, it is clear that except when the workload is dominated by S-queries, DG should be our algorithm of choice among the algorithms proposed in this paper. In this section we compare the cost of the relational configurations derived using DG with the following baselines:

InlineUser (IU): This is the same algorithm evaluated in [1].

Optimal (OPT): A lower bound on the optimal configuration for the workload given a specific set of transformations. Since DG gives configurations of the best quality among the 3 algorithms evaluated, the algorithm to compute the lower bound consisted of transforms available to DG. We evaluated this lower bound by considering each query in the workload individually and running an *exhaustive* search algorithm on the subset of types relevant to the query. Note that an exhaustive search algorithm is possible only if the number of types involved is very small since the number of possible relational

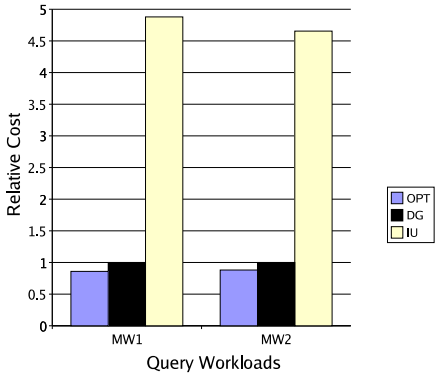


Fig. 8. Comparison of DG with the Baselines

configurations increases exponentially with the number of types. The exhaustive search algorithm typically examined several orders of magnitude more configurations than DG.

We present results for two workloads, MSW1 and MSW2 (MSW1 contains 4 M- and 7 S-Queries and MSW2 contains 3 M- and 5 S-Queries). The proportion of queries in each workload was 50% each for S-Queries and M-Queries. The relative cost for each baseline is shown in Figure 8. InlineUser compares unfavorably with DG. Though InlineUser is good when there are not many shared types, it is bad if the schema has a few types which are shared or repeated since there will not be too many types left to inline. The figures for the optimal configuration show that DG is within 15% of the optimal, *i.e.*, it provides extremely high quality configurations. This also implies that the choice of starting schema (fully decomposed) does not hamper the search algorithm in finding an efficient configuration.

5 Optimizations

Several optimizations can be applied to speed up the search process. In what follows, we illustrate two such techniques. For a more comprehensive discussion, the reader is referred to [11].

5.1 Grouping Transformations Together

Recall that in DG, in a given iteration, *all* applicable transformations are evaluated and the best transformation is chosen. In the next iteration, all the remaining applicable transformations are evaluated and the best one chosen. We found that in the runs of our algorithms, it was often the case that, in a given iteration in which n transforms were applicable, if transformations T_1 to T_n were the best n transformations in this order (that is, T_1 gave the maximum decrease in cost and T_n gave the minimum decrease), other transformations up to T_i, for some $i <= n$, were chosen in subsequent iterations. This being the case, grouping transformations T_1 to T_i together has the potential to

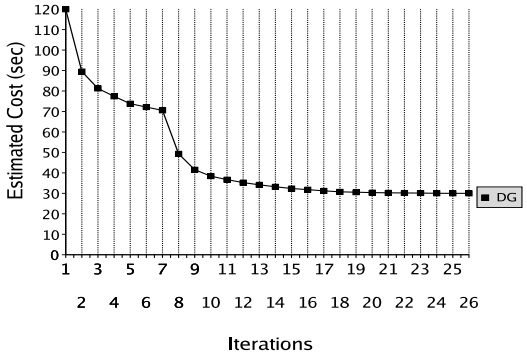


Fig. 9. Progress of DeepGreedy on Workload W

save several iterations. Using this observation, we developed a variation of Algorithm 1, called *GroupGreedy* (GG).

We tried this optimization for DG on several workloads and the results were very encouraging. The cost of the final configuration of GG was within 1% of DG and the number of configurations examined by GG were about 30% for MW1 and about 20% for MW2 compared to DG.

5.2 Early Termination

One obvious optimization is to stop the algorithm once the decrease in the estimated cost goes below a small threshold δ. This saves several iterations which are costly to perform, but do not give substantial decrease in cost. This optimization would be possible if the *decrease* in cost is monotonic. However, during the course of our experiments, we came across several workloads which did not exhibit this behavior. The progress of DG on such a workload, W, is shown in Figure 9. Reasons for this behaviour are analyzed in [11].

Clearly, with an unfortunate value of δ, the algorithm would terminate at iteration 7 and miss the big cost decrease at iteration 8. Thus, while this optimization would result in improved execution times, the derived schema may be suboptimal.

6 Related Work

Existing techniques for XML-to-relational storage can be broadly classified into: *generic* (*e.g.,* the edge mapping of [5]); *data-centric*, where the structure of the XML document is *mined* to guide the mapping process (*e.g.,* [4, 14, 18]); and *schema-centric*, which make use of schema information in the form of DTD or XML Schema in order to derive an efficient relational storage design for XML documents (see *e.g.,* [15–17]).

The LegoDB system [1] was the first schema-centric cost-based approach for automatically generating XML-to-relational mappings and took into account the schema,

statistics and the query workload to derive a low-cost relation configuration. In this paper, we examine the search problem in detail. More recently, a cost-based approach was also described in [19] where a hill-climbing algorithm and a set of four transforms are used. Though the goals of our work and theirs is the same, we differ in the set of transformations used (they consider transforms similar to inline/outline and type split/merge). Also, we have developed a series of search strategies and proposed optimizations to prune the search space.

Formalizing the problem of finding the *optimal* XML-to-relational mapping was considered in [9]. They analyze the interaction between mapping and query translation for a subset of XML queries and XML Schemas under two simple cost metrics. In contrast, our goal in the paper is to develop *practical algorithms* for selecting *good* decompositions.

Support for XML storage is currently provided by all major commercial RDBMSs, including SQLServer [3], Oracle XML DB [10] and DB2 [7]. Although different kinds of mappings are available, these mappings either need to be defined by the user or are fixed. These systems could benefit from a cost-based approach such as the one described in this paper.

7 Conclusions and Future Work

In this paper, we described a framework for exploring the space of XML-to-relational mappings and defined several transformations which exploit the regular expressions in XML Schema (such as unions and repetitions). These transformations encompass physical database design strategies such as vertical and horizontal partitioning – through the use of inline/outline and union distribution respectively. The framework is extensible and new transformations such as some of the OO-to-relational mapping techniques [13] can be added.

We designed and implemented three greedy algorithms and studied how the quality of the final configuration is influenced by the transformations used and the query workload. We have also proposed optimizations to speed up the time taken by the search algorithm with little loss in the quality of the final relational configuration. Experimental results show that our new algorithms provide significantly improved relational schemas as compared to those derived by previous approaches in the literature.

This study can serve as a platform for further investigation into the problem of efficient storage of XML in relational backends. There are several directions for future work. For example:

1. The query workloads considered in this paper contain queries which retrieve data from the database. It would be interesting to investigate a broader range of workloads such as those which involve updates to the database. Update queries are especially significant if indexes and views are considered in the relational configuration.
2. Another scenario not so far addressed is what would happen if the application's query workload changes significantly in terms of the queries being asked. The challenge would then be to find a "minimum change" relational configuration which is efficient for the new workload as well as efficient in terms of the changes needed to the existing configuration (*i.e.,* schema evolution).

Acknowledgements This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India.

References

1. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of ICDE*, 2002.
2. A. Brown, M. Fuchs, J. Robie, and P. Wadler. XML Schema: Formal description, 2001. W3C working draft available at http://www.w3.org/TR/2001/WD-xmlschema-formal-20010320/.
3. A. Conrad. A survey of Microsoft SQL Server 2000 XML features. http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp?frame=true, July 2001.
4. M. Fernandez, W. C. Tan, and D. Suciu. Silkroute: trading between relations and XML. *WWW9/Computer Networks*, 33(1-6), 2000.
5. D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3), 1999.
6. J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: Making XML count. In *Proc. of SIGMOD*, 2002.
7. IBM DB2 XML Extender. http://www-3.ibm.com/software/data/db2/extenders/xmlext/library.html.
8. Internet movie database. http://www.imdb.com.
9. R. Krishnamurthy, V. Chakaravarthy, and J. Naughton. On the difficulty of finding optimal relational decompositions for XML workloads: a complexity theoretic perspective. In *Proc. of ICDT*, 2003.
10. Oracle XML DB: An oracle technical white paper. http://technet.oracle.com/tech/xml/content.html, 2003.
11. M. Ramanath, J. Freire, J. Haritsa, and P. Roy. Searching for efficient XML to relational mappings. Technical Report TR-2003-01, DSL/SERC, Indian Institute of Science, 2003.
12. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD*, 2000.
13. M. Rys. *Materialisation and Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System*. PhD thesis, ETH, Zurich, 1997.
14. A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proc. of WebDB*, 2000.
15. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, 1999.
16. I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of SIGMOD*, 2002.
17. Wang Xiao-ling, Luan Jin-feng, and Dong Yi-sheng. An adaptable and adjustable mapping from XML data to tables in RDB. In *First VLDB Workshop on EEXTT*, 2002.
18. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. Xrel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1), 2001.
19. S. Zheng, J-R. Wen, and H. Lu. Cost-driven storage schema selection for XML. In *Proc. of DASFAA*, 2003.