SEARCHING FOR EFFICIENT XML-TO-RELATIONAL MAPPINGS

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

Technical Report
TR-2003-01

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

[1]Computer Science and Engineering, Oregon Graduate Institute, Beaverton, Oregon 97006, USA.
[2]KRESIT, IIT Bombay, Mumbai 400076, India.

Searching for Efficient XML-to-Relational Mappings

Maya Ramanath
Indian Institute of Science
maya@dsl.serc.iisc.ernet.in

Juliana Freire
OGI/OHSU
juliana@cse.ogi.edu

Jayant Haritsa
Indian Institute of Science
haritsa@dsl.serc.iisc.ernet.in

Prasan Roy
Indian Institute of Technology
prasan@it.iitb.ac.in

Abstract

Cost-based strategies to derive relational configurations for XML applications have been recently proposed and shown to provide substantially better configurations than heuristic methods. These strategies make use of *schema transformations* to a canonical schema in order to derive various relational configurations.

In this paper, we propose a flexible framework for schema transformations and show how it can be used to explore the search space of relational configurations at various granularities. In particular, we use this framework for designing algorithms to iteratively search the space of configurations. We address several issues which arise in this context including that of propagating accurate statistics in the iterations of the search algorithm; and investigating the effect of the query workload on the quality of the relational configurations that are derived and on the run-time of the algorithm. We also propose optimizations to speed up the search process without significant loss in the quality of the relational configurations.

Our experiments indicate that a judicious choice of transformations and search strategy can lead to relational configurations of substantially higher quality than those recommended by previous approaches.

1 Introduction

XML has become an extremely popular medium of information exchange. 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, 17] and shown to provide substantially better configurations than heuristic methods (*e.g.*, [13]). The general methodology used in these strategies is to define a set of *schema transformations* that derive different relational configurations.

The quality of the relational configurations is evaluated by a costing function applied to a given *XML query workload*. A greedy heuristic or variants thereof are used to search through the associated space of relational configurations.

In this paper, we study, for the first time, the impact of schema transformations and the query workload on *search strategies* for finding efficient XML-to-relational mappings. Specifically, we develop a framework for generating XML-to-relational mappings which incorporates a comprehensive set of schema transformations and is capable of supporting different mapping schemes such as ordered XML and schema-less content. Our framework represents an XML Schema through type constructors and uses this representation to define several schema transformations from the existing literature. We also propose variations of these transformations that lead to a more efficient search process, as well as new transformations that derive additional useful configurations.

In order to study the problem of searching for an efficient relational configuration in depth, we have implemented this framework on top of the LegoDB prototype [1]. Here, we describe a series of greedy algorithms that we have experimented with, and show how the choice of transformations impacts the search space of configurations. The algorithms differ in the number and type of transformations they utilize. Intuitively, the size of the search space examined increases as the number/type 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 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.

An important aspect of cost-based XML-to-relational mapping is evaluating the cost of the input workload for each of the derived configurations. In order to compute precise cost estimates, it is important that accurate statistics are available as transformations are applied. Clearly, it is not practical to scan the base data for each relational configuration derived. As discussed later in this paper, we address this issue by gathering statistics at the appropriate granularity before the search starts; and *deriving* accurate statistics during the search process.

In summary, our contributions are:

- A framework for exploring the space of XML-to-relational mappings.
- More powerful variants of existing transformations and their use in search algorithms.
- 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.
- Optimizations to speed up these algorithms and to prune the search space.

Organization

Section 2 develops the framework for XML-to-relational mappings. Section 3 outlines a methodology for search. Section 4 proposes three different search algorithms based on the greedy heuristic. Section 5 evaluates the search algorithms and Section 6 discusses several optimizations to reduce the search time. Section 7 discusses related work and finally, Section 8 summarizes our results.

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

Figure 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 Movie | type Tv) }}
define type Director { element DIRECTOR {
    type Name, type Directed*}}
define type Directed {
    element DIRECTED {type Title, type Year, type Info }}
```

Figure 2: The (partial) IMDB Schema

2 Framework for Schema Transformations

2.1 Schema Tree

A *schema tree* is a representation of the XML Schema in terms of its type constructors. An XML Schema can be regarded as a *complex type* represented using the type constructors for: *sequence* (","), *repetition* ("$*$"), *option* ("?"), *union* ("|"), $<$*tagname*$>$ (corresponding to a tag) and $<$*simple type*$>$ corresponding to base types (*e.g.,* integer). Figure 1 gives a simplified grammar for the schema tree[1].

To illustrate the representation of a schema tree, consider the partial XML Schema in Figure 2 representing the data from the IMDB (Internet Movie DataBase) website [7]. Here, $Title$, $Year$, Aka and $Review$ are simple types. The schema tree for an excerpt of this schema is shown in Figure 3 (note that base types are not shown). Nodes in the tree are *annotated* with the *names* of the types present in the original schema – these annotations are shown italicized next to the tags (shown in boldface) in Figure 3. Some points are worthy of note. First, there need not be any correspondence between tag names and annotations (type names). Second, the schema graph is actually represented as a tree, where although different occurrences of equivalent nodes are captured, their content is shared (see *e.g.,* the nodes $\mathbf{TITLE^1}$ and $\mathbf{TITLE^2}$ in Figure 3). Finally, recursive types can be handled similarly to *shared* types, *i.e.,* the base occurrence and the recursive occurrence are differentiated, but both share the same content.

Any subtree in the schema tree can be regarded as a type and the node corresponding to that subtree can be annotated *without* changing the structure of the tree. We refer to this annotation as the *name* of the node and use it synonymously with annotation. We also use the terms subtree, node and type interchangeably throughout the paper.

[1]The schema tree is an ordered tree.

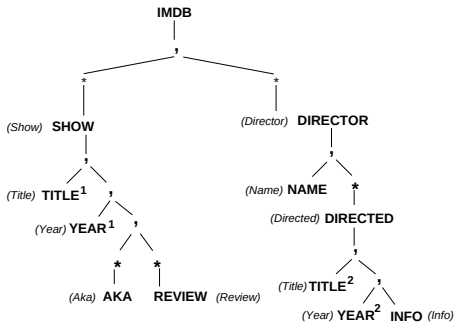

Figure 3: (Partial) Schema Tree for the IMDB Schema

2.2 From Schema Trees to Relational Configurations

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

- If N is an annotation in the schema tree, 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.
- 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.
- If N is the annotation of a node, then T_N contains as many additional columns as the number of non-annotated children of N that are of type <simple type>.

Other rules which may help in deriving efficient schemas are as follows. Note that the mapping can follow any set of rules – not necessarily those presented below.

- Repeated types are stored in a separate table. The alternatives would be to (i) store each occurrence of the repetition as separate columns in its parent table leading to an artificial upper bound on the number of repeats, or (ii) store all occurrences of the repetition in the same column by duplicating the values in the rest of the tuple leading to wasted space (as well as increased complexity for updates).
- Types which are part of a union are stored in a separate table. This rule avoids nulls in the parent table.

The relational configuration corresponding to the naming in Figure 3 for the Director subtree is shown in Figure 4.

It is possible to support a different mapping scheme as well – for example in order to support ordered XML, one or more additional columns have to be incorporated into the relational table [14]. 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 define a rule mapping annotated nodes of that type to a ternary relation (edge table) [4].

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_id]
Table	Year	[Year_key, YEAR, parent_directed_id]
Table	Info	[Info_key, INFO, parent_directed_id]

Figure 4: Relational Schema for the (partial) Schema Tree

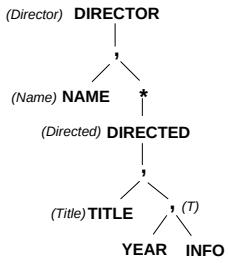

Figure 5: A Subset of Annotations

2.3 Inline and Outline

As described above, any node (type) which is annotated has a relational table associated with it. We call such types as *Outlined*. Note that annotating any node in the schema tree corresponds to outlining the type corresponding to that node.

In contrast, all nodes which are not annotated are *Inlined* since they are stored within the table of the closest outlined ancestor if they are simple types. We can *inline* a previously outlined type by simply deleting its annotation. Thus, through these *naming* operations on the schema tree, we can derive an exponentially large number of possible relational configurations of the given schema.

Inline and outline may also be used to group elements together. Consider Figure 5 in which introducing the annotation T and removing annotations $Year$ and $Info$ results in the new relational schema shown in Figure 6. This configuration *groups* Year and Info together in a single table. Note that the usefulness of this configuration is decided by the system searching the space of configurations.

2.4 Other Transformations

Before we describe additional transformations and how they derive different relational configurations, we introduce a compact notation to describe the type constructors, and using this notation, we define the notion of *syntactic equality*.

Tag Constructor: $E(label, t, a)$, where $label$ is name of the tag (such as TITLE, YEAR, etc.), t is its subtree and a its annotation (if any).

Sequence, Union, Option and Repetition Constructors: Each of the constructors are defined as: $C(t_1, t_2, a)$, $U(t_1, t_2, a)$, $O(t, a)$, and $R(t, a)$, respectively, where t_1, t_2, t are subtrees and a is the annotation.

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

Definition 2.1 (Syntactic Equality) Two types T_1 and T_2 are syntactically equal – denoted by $T_1 \cong T_2$ – if the following holds:

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_id]
Table	T	[T_key, YEAR, INFO, parent_Directed_id]

Figure 6: Relational Schema with Annotation "T"

define type **Show** { element SHOW {type Title, (type TV|type Movie) }}
define type **Director** { element DIRECTOR {type Title, type Directed }}
define type **Title** { element TITLE {string}}

Type split Title →

define type **Show** { element SHOW {type STitle, (type TV|type Movie) }}
define type **Director** { element DIRECTOR {type DTitle, type Directed }}
define type **STitle** { element TITLE {string}}
define type **DTitle** { element TITLE {string}}

Figure 7: Example of a Shared Type and Type Split

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 $b = b'$

2.4.1 Type Split/Merge

The inline and outline operations are analogous to removing and adding annotations to nodes. We now defi ne two transformations based on the *renaming* of nodes: *Type Split* and *Type Merge*. We refer to a type as *shared* when it has distinct annotated parents. In the example shown in Figure 7, the type Title is shared by the types Show and Directed. Consequently, the table corresponding to Title would contain a parent_id column which contains key values from both Directed as well as Show – hence the parent_id column is not a foreign key.

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 confi guration is derived where a separate table is created for each type of title. Conversely, the type merge operation adds identical annotations to types whose *corresponding subtrees* are syntactically equal. The annotation Title would occur twice in the schema tree – once each under Show and Directed.

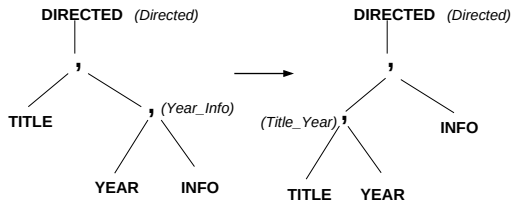

Figure 8: Applying Associativity

2.5 Structural Transformations

All the transformations defined so far are limited to modifying the annotations of nodes, and preserving the structure of the schema tree. We now define operations which change the structure of the schema tree. Some of these transforms were originally described in our earlier work [1]. We redefine and extend these transforms in our new framework.

2.5.1 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. Consider, for example, the type Directed shown in Figure 8. The first tree in this figure yields a relational schema in which the Year and Info about Directed are stored in a single table called Year_Info. We can change this grouping by applying associativity as shown in the second tree and obtain a relational schema in which Title and Year appear in a single table called Title_Year.

Commutativity by itself does not give rise to different relational mappings [2], but when combined with associativity may generate mappings different from those considered in the existing literature. For example, in Figure 8, 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.

2.5.2 Union Distribution/Factorization

Using the standard distribution law for distributing sequences over unions for regular expressions, we can separate out components of a union: $(a,(b|c)) = (a,b)|(a,c)$. 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 →

```
define type Show {
    (element SHOW { type Title, type TV }) |
    (element SHOW { type Title, type Movie }) }}
```

Outline →

[2] Note that commuting the children of a node no longer retains the original order of the XML schema.

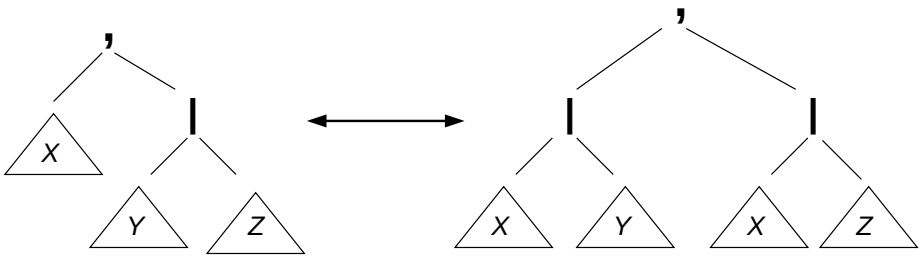

Figure 9: Union Distribution and Factorization

```
define type Show { type TVShow | type MovieShow}
define type TVShow {
    element SHOW { type Title, type TV }}
define type MovieShow {
    element SHOW { type Title, type Movie }}
```

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 would enable 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, *i.e.*, one partition is created for TV shows and another for movies. Conversely, the union factorization transform would factorize the union. For the example above, this would be done by first doing a type merge of TVTitle and MovieTitle and then doing a factorization of the union to get back the original schema. In order to determine whether there is potential for a union distribution we search the schema tree for the pattern shown in the left-hand side of Figure 9. Similarly, in order to locate a potential union factorization, we search for the pattern shown in the right-hand side of Figure 9. [3] 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. Due to lack of space, we do not enumerate them here.

2.5.3 Repetition Split/Merge

According to the rules in Section 2.2, a repeated type is always stored in a separate table. However, it is possible to inline some of these values by a transformation which *splits* the repetition. For example:

[3] The problem of finding these patterns on schema trees is an instance of non-linear tree pattern matching and can be performed in time $O(m * n)$, where m is the number of nodes in the pattern and n the number of nodes in the target, and has been shown to have a linear expected time behavior [10].

define type **Show** {element SHOW {type Title, type Aka*} }

Split Repetition →

define type **Show** {element SHOW {type Title, type Aka1?, type Aka2*} }

By *splitting* the repetition Aka*, the new type Aka1 may be inlined into Show. Aka2* may be split over and over again. [4] In order to prevent an infinite number of splits, the number of splits must be fixed during the search. *Repetition merge* is the inverse of split — it merges the types Aka1 and Aka2 into Aka.

Candidates for repetition split and merge can be identified in the schema tree by identifying the patterns $R(X, -)$ and splitting it into $C(O(X, -), R(X, -), -)$ and conversely looking for the latter transform and merging it into the former.

Many other transforms such as *simplifying unions* [13] (a lossy transform which enables the inlining of one or more of the components of the union), etc. can be defined similarly, and the patterns to be matched in order to discover the potential for such a transform can be enumerated. Next, we outline search algorithms which consider the transformations discussed in this section, namely, Inline/Outline, Type Split/Merge, Union Distribution/Factorization and Repetition Split/Merge. In the remaining part of the paper, we refer to Type Merge, Union Factorization and Repetition Merge as *merge transforms* and Type Split, Union Distribution and Repetition Split as *split transforms*.

3 Evaluating Configurations

As mentioned in the Introduction, it is important that during the search process precise cost estimates are computed for the query workload under each of the derived configurations — this, 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.

Collection and Propagation of Statistics

For ease of exposition, we describe the collection and propagation of statistics at the XML Schema level, and later show how to translate these into relational statistics.

An important observation about the transformations defined in Section 2 is that whereas merge operations preserve the accuracy of statistics, split operations do not. Intuitively, if two types $T1$ and $T2$ are merged into T, precise statistics for T can be derived by summing/unioning the statistics of $T1$ and $T2$. However, when a type T is split into $T1$ and $T2$, in general it is not possible to determine precisely the statistics for the new types. (Note that in some special cases, *e.g.*, for outline transform, it may be possible to accurately *infer* the statistics of the new type from the statistics of the parent type.)

Consequently, in order to preserve the accuracy of the statistics, before the search procedure starts, *all* possible split operations are applied to the user XML schema.

[4] Note that Aka1 and Aka2 share the same content.

Statistics are then collected for the *fully decomposed* schema. Subsequently, during the search process, only merge operations are considered.

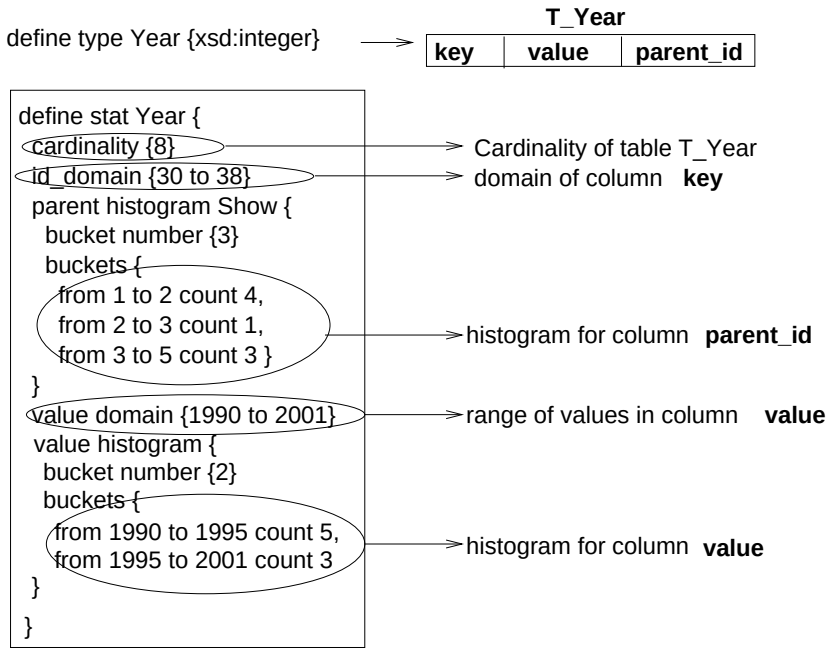

Figure 10: Statistics Translation

In our prototype implementation, we use StatiX [5] to collect statistics for *annotated types*. The StatiX system provides concise and accurate summaries which can be easily translated into relational statistics. The translation procedure is illustrated through the example in Figure 10.

The derived relational statistics are used as input to a relational optimizer, which in turn computes cost estimates for the (also appropriately translated) query workload under a given relational configuration. In our implementation, we use the Volcano-based optimizer described in [11]. Since the accuracy of this optimizer has been verified by the execution of queries over a commercial RDBMS, given precise statistics at all stages of transformations, this optimizer is *expected* to provide cost estimates that are consistent with those of commercial RDBMS.

Due to space limitations, we do not discuss the translation procedure from XQuery to SQL. Details about this process can be found in the literature [13, 14, 3].

4 Search Algorithms

In this section we describe three greedy algorithms we have implemented using our framework. They differ in the choice of transformations that are selected and applied

at each iteration of the search. In the remainder of the section, we describe these algorithms in detail.

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
```

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 conversion from 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 is also very flexible. This flexibility is achieved by varying the strategies to select applicable transformations at each iteration (function a$pplicableTransforms$ in line 8). In the experiments described in [1], only inline and outline were considered as the applicable transformations and the utility of the other transformations (*e.g.*, union distribution and repetition split) were shown independently. Below, we describe variations to the basic greedy algorithms that allow for a richer set of transformations.

As discussed in Section 3, it is important to perform all splits and then the merges on the schema to preserve the accuracy of statistics. It is worth pointing out that fixing this order is also important to avoid re-generating the same relational configuration in different iterations of the search. In the rest of the paper, we assume that the starting schema for all search algorithms is the *fully decomposed schema* and only merge operations are applied during the greedy iterations.

4.1 InlineGreedy

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

4.2 ShallowGreedy: Adding Transforms

The *ShallowGreedy* 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 2.1), it only performs single-level or *shallow* merges.

The notion of syntactic equality, however, can be too restrictive for effective exploration of the search space. 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 ShallowGreedy, these two type merges by themselves may not reduce the cost, but taken in conjunction with the union merge would make a substantial impact. If that is the case, ShallowGreedy is handicapped by the fact that a union merge will never be be applied since the two type merges will not be chosen by the algorithm. In order to overcome this problem, we design a new algorithm called *DeepGreedy*, which we describe below.

4.3 DeepGreedy: *Deep* merges

Before we proceed to describe the DeepGreedy algorithm, we first introduce the notions of *Valid Transforms* and *Logical Equivalence*. The set of *valid transformations* for a given schema tree S is a subset of all the applicable transformations in S.

Definition 4.1 (Logical Equivalence) Two types T_1 and T_2 are *logically equivalent* under a 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 $V = \{Inline\}$; $t_1 := E(TITLE, S(string, -), Title_1)$, and $t_2 := E(TITLE, S(string, -), Title_2)$. 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 ShallowGreedy can be identified as candidates, provided the necessary operations can be fired to *enable* the transform. That is, if T_i and T_j are identified as a potential type merge, then to perform this type merge, t_1 and t_2 are *recursively* type merged in order to enable the type merge of T_i and T_j. 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 DeepGreedy allows the same transforms as ShallowGreedy, *except* that potential transforms are identified not by syntactic equality, but by logical equivalence with the set of valid transforms containing all the merge operations (including inline). This allows DeepGreedy to perform *deep* merges. Note that although not covered in this paper, additional variations of the search algorithms are possible, *e.g.*, by restricting the set of valid transforms.

5 Performance Evaluation

In this section we present a performance evaluation of the three algorithms proposed in this paper: InlineGreedy, ShallowGreedy and DeepGreedy. The purpose of this evaluation is twofold: (1) to analyze the relative performance of the algorithms on different kinds of query workloads, and (2) to establish the competitiveness of the proposed algorithms.

5.1 Query Workloads

We evaluated each of the algorithms on several query workloads based on (1) the efficiency of the derived relational configuration, and (2) the efficiency of the search algorithm. These are the same metrics used in [1]. Note that the latter is the same as 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 earlier in the paper, notice that the behavior of each algorithm 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 DeepGreedy nor ShallowGreedy is expected to perform better than InlineGreedy.

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 lookup. For example:

```
SQ1:  for $i in /IMDB/SHOW
      where $i/TV/CHANNEL = 9
      return $i/TITLE

SQ2:  for $i in /IMDB/DIRECTOR
      where $i/DIRECTED/YEAR = 1994
      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. The performance of the proposed algorithms on S-query workloads is analysed in Section 5.2.

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 ask 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, $k/SOURCE,
            $k/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 a 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). Thus the query may benefit from reduced fragmentation. 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 queries - a union of SW1 and SW2 above.
- **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 Section 5.2 and Section 5.3 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. 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 MQ1 would reduce the number of tuples. Such queries thus benefit from split transformations as well as merge transformations. However, in case the two types of transformations conflict, we need to analyze the balance between the two. But considering arbitrary queries is unlikely to give much insight because the impact of split transformations vs. merge transformations would be different for different queries. Thus, we chose to work instead on a workload containing a *mix* of S- and M-queries, where the impact of split transformations vs. the merge transformations is controlled using a parameter. The performance of the proposed algorithms on one such workload as a function of the control parameter is studied in Section 5.4. Finally, in Section 5.5 we demonstrate the competitiveness of the configurations derived using the proposed algorithms against those derived using certain baselines/prior approaches.

5.2 Performance on S-Query Workloads

Recall that DeepGreedy does "deep" merges, ShallowGreedy does "shallow" merges and InlineGreedy allows only inline. S-Queries do not fully exploit the potential of DeepGreedy since they do not benefit from too many merge transformations. So, DeepGreedy could possibly be considering transformations which are useless. This would make it more inefficient in the run time without any major advantages in the cost of the derived schema. We present results for the 3 workloads: SW1, SW2 and SW3.

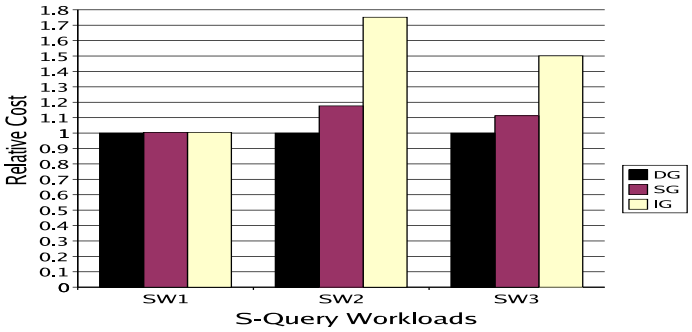

Figure 11: Cost of Workloads containing S-Queries

The cost difference, shown in Figure 11 of the derived configuration between DeepGreedy and ShallowGreedy was less than 1% for SW1 and ShallowGreedy and InlineGreedy gave the same cost for SW1. But the cost difference between DeepGreedy and ShallowGreedy for SW2 jumped up to around 17% due to the return values benefiting from merge, and the difference between DeepGreedy and InlineGreedy was around 48%.

Figure 12: No. of configurations Examined for Workloads Containing S-Queries

The relative number of configurations examined by each of DeepGreedy, ShallowGreedy and InlineGreedy are shown in Figure 12. In terms of number of schemas examined, DeepGreedy examined a much larger set of configurations than ShallowGreedy, while ShallowGreedy examined more number of configurations than InlineGreedy except in the case of SW2 where the ShallowGreedy was comparable to DeepGreedy.

5.3 Performance on M-Query Workloads

Figure 13 shows the relative costs of the 3 algorithms for the 3 workloads, MW1, MW2 and MW3. As expected DeepGreedy performs extremely well compared to ShallowGreedy and InlineGreedy since DeepGreedy is capable of performing deep merges which benefit MW1. Note that the effect of adding selective predicates reduces the magnitude of difference in the costs between DeepGreedy, ShallowGreedy and InlineGreedy.

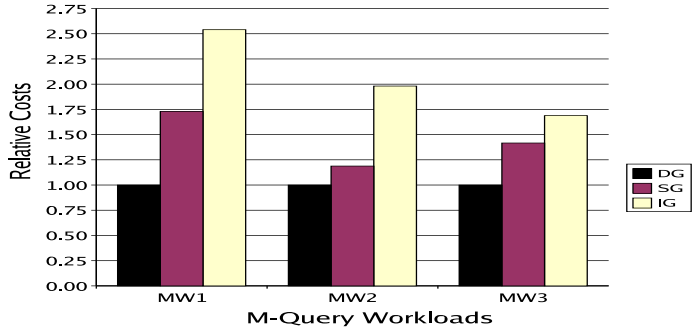

Figure 13: Cost of Workloads Containing M-Queries

In terms of the number of configurations examined, DeepGreedy performed the best as compared to ShallowGreedy and InlineGreedy. This would seem counter-intuitive – we would expect that since DeepGreedy is capable of examining a superset of trans-

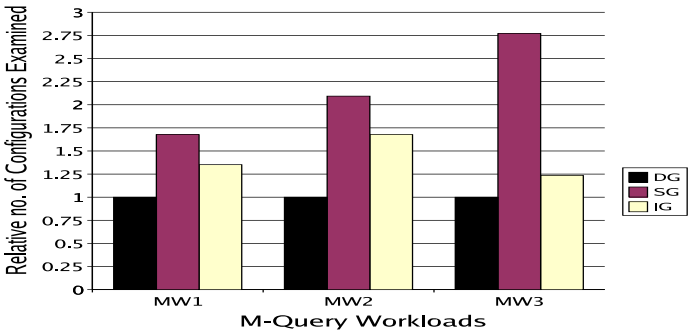

Figure 14: No. of configurations Examined for Workloads Containing M-Queries

formations as compared to ShallowGreedy and InlineGreedy, it would take longer to converge. However, this did not turn out to be the case since DeepGreedy picked up the cost saving recursive merges (such as union factorization) fairly early on in the run of DeepGreedy which reduced the number of lower level merge and inline candidates in the subsequent iterations. This enabled DeepGreedy to converge faster. By the same token, we would expect ShallowGreedy to examine less number of configurations than InlineGreedy, but that was not the case. This is because ShallowGreedy was not able to perform any major cost saving merges since the "enabling" merges were never chosen individually (note the cost difference between DeepGreedy and ShallowGreedy). Hence, the same set of merge transforms were being examined in every iteration without any benefit, while InlineGreedy was not burdened with these candidate merges. But note that even though InlineGreedy converged faster, it was mainly due to the lack of useful inlines as reflected by the cost difference between InlineGreedy and ShallowGreedy.

5.4 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 InlineGreedy is the best algorithm to run since it returns a configuration with marginal difference in cost compared to DeepGreedy and in less time (reflected in the results for SW1), while if the query workload consists of M-Queries, then DeepGreedy is the best algorithm to run.

Since InlineGreedy performs well for workloads with mainly S-Queries, it is reasonable to expect that if the query workload is dominated with S-Queries, then InlineGreedy would perform well. Similarly, if M-Queries dominate the workload, then DeepGreedy would perform best. Note that in either case, DeepGreedy would provide the best results in terms of cost.

We considered several different "mixed" workloads, but present here results for only one 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 $k/7$ to each of the 7 S-queries and weight $(1 - k)/4$ to each of the 4 M-queries.

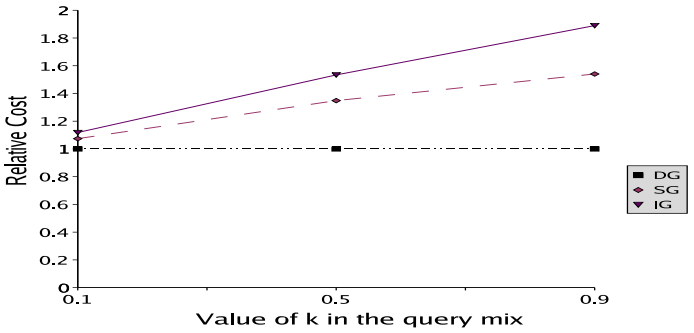

Figure 15: Cost of Workloads Containing both M- and S-Queries

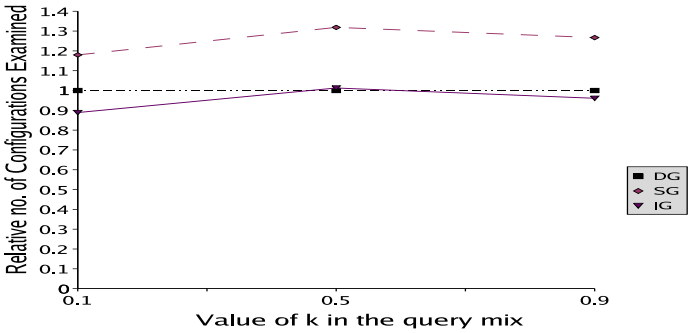

Figure 16: No. of Configurations Examined for Workloads Containing M- and S-Queries

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 15. Expectedly, when S-Queries dominates, InlineGreedy performs quite competitively with DeepGreedy (with the cost of InlineGreedy being within just 15% of DeepGreedy). 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 16. DeepGreedy examines more configurations than InlineGreedy when S-Queries dominates, but the gap is almost closed for the other cases.

Note that both ShallowGreedy and InlineGreedy 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.

5.5 Comparison with Baselines

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

- **Fully Decomposed, All Outlined (FDAO):** Fully decompose the schema and outline all its types.
- **Fully Decomposed, All Inlined (FDAI):** Fully decompose the schema and inline as many types as possible.
- **Fully Merged, All Outlined (FMAO):** Retain the original schema and outline all its types.
- **Fully Merged, All Inlined (FMAI):** Inline as many types as possible in the original schema.
- **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 DeepGreedy gives configurations of the best quality among the 3 algorithms evaluated, the algorithm to compute the lower bound consisted of transforms available to DeepGreedy. 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 configurations increases exponentially with the number of types. The exhaustive search algorithm typically examined several orders of magnitude more configurations than DeepGreedy.

Note that the first 4 baselines are non cost-based. 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.

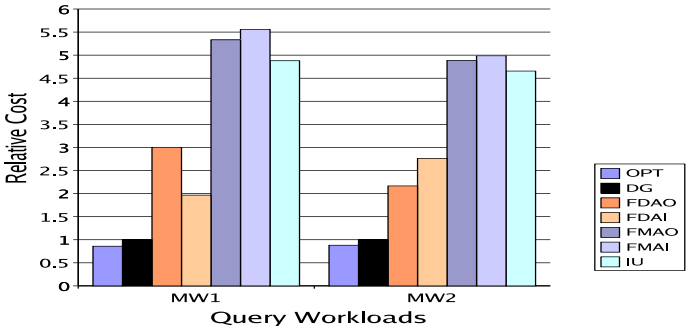

Figure 17: Comparison of DeepGreedy with the Baselines and Inline (User)

The relative cost for each baseline is shown in Figure 17. As expected, none of the non-cost-based baselines are competitive with DeepGreedy. Moreover, InlineUser

also compares unfavorably with DeepGreedy. 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 or part of unions since there will not be too many types left to inline. Note also that the results of InlineUser confirms the results in [1], since it derives configurations up to 50% better than the all-inlined configuration on the original user schema. The figures for the optimal configuration also show that DeepGreedy is within around 15% of the optimal.

6 Optimizations

There are several different optimizations that can be done to speed up the search algorithms. We propose a few of them here and outline their drawbacks and advantages.

6.1 Grouping Transformations Together

Algorithm 2 GroupGreedy 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:   prevMinCost ← minCost
 7:   transforms ← applicableTransforms(S)
 8:   sortedTransforms = SORT(transforms)
 9:   for all T in sortedTransforms do
10:     if applicable(T) then
11:       S' ← Apply T to S
          {S is preserved without change}
12:       rel_schema ← convertToRelConfig(S', queryWkld)
13:       Cost ← COST(rel_schema)
14:       if Cost < minCost then
15:         minCost ← Cost
16:         S ← S' {Retain the merge}
17:       else
18:         Goto step 5
19:       end if
20:     else
21:       Goto step 5
22:     end if
23:   end for
24: end while
```

Recall that in DeepGreedy, 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 save several iterations. Using this observation, we developed a variation of Algorithm 1, called *GroupGreedy* (Algorithm 2).

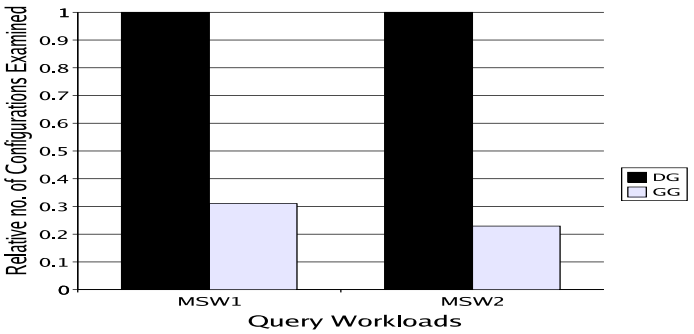

Figure 18: No. of Configurations Examined by DeepGreedy and GroupGreedy

We tried this optimization for DeepGreedy on several workloads and the results were very encouraging. The cost of the final configuration of GroupGreedy was within 1% of DeepGreedy and the number of configurations examined by GroupGreedy were a fraction of those examined by DeepGreedy, as shown in Figure 18.

6.2 Early Termination

One obvious optimization is to stop the algorithm once the decrease in the estimated cost goes below a small δ. This would save 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 DeepGreedy on such a workload, W, is shown in Figure 19.

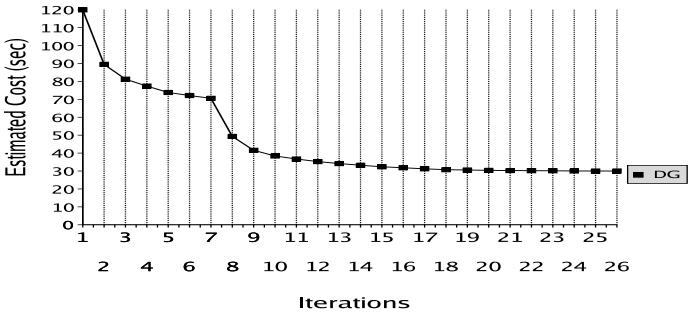

Figure 19: Progress of DeepGreedy on Workload W

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.

An discussion of why we see such sudden cost decreases is available in the Appendix.

6.3 Applying Profitable Transforms

Though it is possible to decompose the schema fully by performing *all* union distributions, repetition splits and type splits, many of them may not be useful and so would simply increase the number of types in the schema. This is especially true for M-Queries where many of the splits ultimately proved to be useless. Thus it would help if an apriori analysis of the query workload and the statistics can be used to cut down on the number of split transforms. This would reduce the number of combinations of merge transforms during search and result in faster execution times.

As an example of such a heuristic, consider a repetition split in the following snippet:

```
define type Show {element SHOW {type Title, type Review*}}
```

It is beneficial to split Review* into Review1? and Review2* only if a majority of the shows have only one review and a very few have more than one. Splitting Review in this way allows the inlining of the first review thus giving rise to potential cost benefits. However, if it is known that most Shows have at least 10 reviews, then it is unlikely that inlining just one Review into Show would yield benefits. Nor would splitting Review help if most Shows had no reviews, but a small number of Shows had lots of reviews.

An interesting direction of future work would be to come up with heuristics based on the statistics available for the XML Schema to decide whether or not perform a particular split operation.

6.4 Reducing the Search Space by Query Analysis

In our metric for the lower bound on the optimal, an exhaustive search was performed on single queries. This was made possible because the number of types relevant to the query was within reasonable levels. The same principle can be applied for the greedy algorithms as well. If the queries in the workload are concentrated to one particular part of the schema, then only those types need to be taken into consideration for the search. Or if the queries in the workload can be partitioned such that each partition has a disjoint set of types relevant to it, then the search can be performed separately for each partition.

7 Related Work

There has been significant interest in the database community to develop efficient storage schemes for XML data. The proposed techniques can be broadly classified into: *generic* (*e.g.*, the edge mapping of [4]); *data-centric*, where the structure of the XML document is *mined* to guide the mapping process (*e.g.*, [3, 12, 16]); 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.*, [13, 14, 15]).

The LegoDB system [1] was the first cost-based approach for automatically generating XML-to-relational mappings. In addition to the XML document and schema,

LegoDB also takes the query workload into account in order to derive a low-cost relational configuration. In this paper, we extend the work presented in [1] in many significant ways. Most notably, we study in depth the problem of searching for XML-to-relational mappings in the presence of a comprehensive set of transformations.

More recently, a cost-based approach was also described by Zheng et al [17]. Zheng et al propose a hill-climbing algorithm and a set of four transformations (similar to inline/outline and type split/merge) to move from one state to the other during a run of the algorithm. They present an evaluation of their algorithm starting from different states, and show experimentally that their derived configurations have lower cost than configurations produced by heuristic-based approaches. Although similar in focus, *i.e.*, the search problem in cost-based storage design, our work differs from theirs in important ways. We use a comprehensive set of transformations (a superset of theirs) that leverage the structure of the document schema in order to derive *useful* configurations. We also develop a series of search strategies, and discuss their performance both in terms of efficiency and quality of derived configuration.

In [8], Krishnamurthy et al take a first step in formalizing the problem of finding an *optimal* XML-to-relational mapping. They show that there is an interplay between the choice of decomposition and the choice of query translation algorithm, i.e., a given mapping is not the *best* for all possible translations; and analyze the interaction between mapping and translation for a restricted set of XML schemas and XML queries, under two simple cost metrics. In contrast, our goal in this paper is to develop *practical algorithms* for selecting *good* decompositions. Although in our experiments we use a fixed query translation strategy and a System-R-style cost model, our system's cost estimation and translation are performed by independent modules. Thus, it is possible to experiment with different cost models and translation strategies. We intend to further investigate the interactions between these components in our future work.

Support for XML storage is currently provided by all major commercial RDBMSs, including SQLServer, DB2, and Oracle. For example: Oracle XML DB [9] provides native XML storage and retrieval; DB2 [6] and SQLServer [2] allow mappings to be defined by users through proprietary languages. 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.

8 Conclusions

In this paper, we describe a framework for exploring the space of XML-to-relational mappings. This framework is able perform a comprehensive search on this space by using previously defined schema transformations, extensions to these transformations, and new transformations (see Section 2). 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 several techniques to prune the search space that lead to faster convergence and little or no loss in the quality of the selected 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.

There are several interesting directions we intend to pursue in future work. We would like to explore mappings that allow for mixed-storage strategies. For example, a mapping that use the generic edge strategy for a part of a schema (*e.g.*, a subset of the schema which is updated often, or elements with ANYTYPE content), and LegoDB-style mapping for the subset of the schema that is well-defined and static. We also intend to investigate the interactions between the choice of mapping and the choice of query translation algorithm; and how to integrate the functionality of design tools that perform index selection and view materialization into our system.

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*, pages 64–75, 2002.

[2] 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.

[3] M. Fernandez, W. C. Tan, and D. Suciu. Silkroute: trading between relations and XML. *WWW9/Computer Networks*, 33(1-6):723–745, 2000.

[4] D. Florescu and D. Kossman. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[5] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: Making XML count. In *Proc. of SIGMOD*, pages 181–191, 2002.

[6] IBM DB2 XML Extender. http://www-3.ibm.com/software/data/db2/extenders/xmlext/library.html.

[7] Internet movie database. http://www.imdb.com.

[8] 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. To Appear.

[9] Oracle XML DB: An oracle technical white paper. http://technet.oracle.com/tech/xml/content.html, 2003.

[10] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *JACM*, 39(2):259–316, 1992.

[11] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD*, pages 249–260, 2000.

[12] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proc. of WebDB*, pages 47–52, 2000.

[13] 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*, pages 302–314, 1999.

[14] 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*, pages 204–215, 2002.

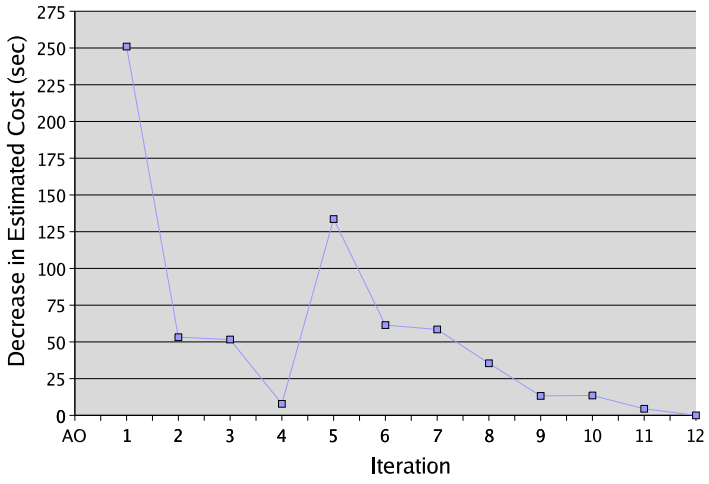

Figure 20: Non-monotonic Decrease in Cost

[15] 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.

[16] 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):110–141, 2001.

[17] S. Zheng, J-R. Wen, and H. Lu. Cost-driven storage schema selection for XML. In *Proc. of DASFAA*, 2003.

Explanation for sudden cost jumps

To explain this behaviour, consider the following simple join query:

```
for $i in IMDB/actor
return $i/name, $i/played/title, $i/played/year,
       $i/played/character, $i/biography/birthdate,
       $i/biography/text
```

The graph showing the *cost decrease* in a run of DeepGreedy for a workload consisting only of this query is shown in Figure 20. The magnitude of the cost decrease is huge at iterations 4 to 5. The relevant transforms are the inline of Played_Year and Played_Title at iterations 4 and 5 respectively. To explain the cost jump, consider the following size of attributes: Title (T) = 30 bytes, Year (Y) = 4 bytes and Character (C) = 25 bytes, Played (P) = 8 bytes (key + parent_key). Note that initially each of these types is outlined.

Cost at iteration 3 (that is before inline of Y) is: $8 * (4 + 25 + 30) + (33 * 12) + (45 * 38) = 2578$

Then, at iteration 4, the first-order query cost after:

1. Inlining Y is $(8 + 4)(30 + 25) + 42 * 37 = 2214$

2. Inlining T is $(8 + 30)(25 + 4) + 63 * 42 = 3748$

3. Inlining C is $(8 + 25)(30 + 4) + 63 * 37 = 3453$

Therefore Y is chosen to be inlined. Note that costs for inlining T or C is *more* than before any inline.

Then, at iteration 5, the tuple size of P with inlined Y is $(8 + 4) = 12$ bytes. Again, the first order query cost after:

1. Inlining T is $(12 + 30)(25) = 1050$

2. Inlining C is $(12 + 25)(30) = 1110$

Therefore T is chosen to be inlined. Note that the cost reduction after the first inline (that is, Y) is *much less* than the cost reduction of the second inline (that is, T) and this matches the observed jump in the cost decrease.

The intuition is that if T was inlined early, while it would certainly reduce the join cost for itself with P, yet it would seriously increase the cost of the joins of P with Y and C since P will now have large tuples. However, once Y is inlined, this degradation has less impact and it becomes a shootout between T and C as per above. That is, for T's cost reduction to happen, it needs to get Y out of the way.