Abstract

Systems for managing and querying semistructured-data sources often store data in proprietary object repositories or in a tagged-text format. We describe a technique that can use relational database management systems to store and manage semistructured data. Our technique relies on a mapping between the semistructured data model and the relational data model, expressed in a query language called STORED. When a semistructured data instance is given, a STORED mapping can be generated automatically using data-mining techniques. We are interested in applying STORED to XML data, which is an instance of semistructured data. We show how a document-type-descriptor (DTD), when present, can be exploited to further improve performance.

1 Introduction

Semistructured data is becoming ubiquitous. The emergence of XML, which is a data format for semistructured data, will increase the availability of semistructured data. Such data is best defined as a graph-based, self-describing object instance model. Data consists of a collection of objects; each object is either atomic (e.g., integer, string, image, audio, video), or complex (i.e., a set of (attribute, object) pairs). Since attribute names are stored with the data, the data is self-describing.

Existing systems for managing and querying semistructured data sources store the schema with the data. Lorel [17] and Tsimmis [16] store their data as graphs; the schema is stored as attributes labeling the graph’s edges. Strudel [6] stores the data externally as structured text, and internally as a graph. XML often is stored in proprietary object repositories or in text files, in which tags encodes the schema. Storing the schema with the data provides the flexibility required by semistructured data. In data integration, for example, data from new sources can be loaded immediately, regardless of its structure, and changes to the structure of old sources can be handled seamlessly. This flexibility, however, incurs a space cost, because the schema is replicated at each data item, and a time cost, because of the additional processing of the replicated schema. A more fundamental disadvantage however, is that we cannot use a commercial RDBMS for managing the semistructured data.

We describe a technique for using an RDBMS to store, query and manage semistructured data. Semistructured data can always be stored as a ternary relation, since the data is an edge-labeled graph, but this is no better than storing the schema with the data. Instead, our technique relies on an aggressive mapping from the semistructured data model to the relational model. The mapping is expressed in STORED (Semistructured TO Relational Data), a declarative query language. A relational schema is chosen, then the STORED mapping translates the semistructured data instance into that schema. The mapping is always lossless; parts of the semistructured data that do not fit the schema are stored in an “overflow” graph.

We expect this technique to be used (1) to store and manage efficiently existing semistructured data sources, and (2) to convert relational sources into a semistructured format, such as XML.

In the first application, the semistructured-data instance exists, e.g., it might be a large XML file. The main issue is generating the relational schema and the STORED mapping automatically from patterns discovered in the data instance. Subsequently, queries and updates over the semistructured view are automatically rewritten into queries and updates over the relational store. If some query mix is known in advance, it may be used during the generation phase. As the data or the query mix changes, the performance of the relational storage may degrade, and a new mapping should be generated (and the relational data reorganized). In the second application, a relational data source is exported in a semistructured view, e.g., in an XML view. In this case, the STORED mapping is defined by the application writer. This application is easier than the first, because the mapping need not be generated automatically. We expect this application will become more important as information providers export data in XML.

Given a semistructured data instance, we have to gen-
erate a “good” relational schema and STORED mapping to that schema. The meaning of “good” depends on the application, but usually includes minimizing disk space, reducing data fragmentation, and satisfying constraints of the RDBMS (e.g., maximum number of attributes per relation). When a query mix on the semistructured data is known, a “good” relational storage reduces the weighted cost of those queries. Hence, this can be modeled as a cost optimization problem. Unlike other optimization problems (of query plans [20] or data warehouse design [18]), our input is the data instance, not a set of queries. This problem is NP-hard in the size of the data. For that reason, we did not pursue the cost bases approach, but instead developed a heuristic algorithm. Wang and Li [21] have described a data-mining algorithm for semistructured data, and we adapt their algorithm to our problem. The result of the data-mining phase is used to produce a reasonable relational schema and STORED mapping.

Given the relational mapping of the STORED query, the system automatically generates the overflow mapping necessary to ensure that any semistructured instance is stored losslessly. This part of the STORED mapping specifies which objects or object parts are stored in the overflow graph. The mapping must be lossless for any data instance, because we support updates, which are propagated to the relational store or the overflow graph. These overflow mappings can be unnecessarily conservative; for XML data, we show that a DTD can be used to simplify the overflow mappings.

Given the complete STORED mapping, our system accepts queries and updates over the semistructured source and rewrites them into queries and updates on the relational source. Rewriting of relational or dialog queries is a well-understood problem [11, 14]. Since STORED is a new query language with novel features, we revisit query rewriting. We show that arbitrary queries on semistructured data, with regular expressions and tree-like patterns, can be rewritten in terms of STORED mappings. Updates are important as well. We show that insertions in the semistructured data can be automatically rewritten as insertions into the relational and overflow stores.

This paper makes the following contributions:

- STORED, a declarative language for specifying storage mappings from the semistructured-data model to the relational model plus overflow graphs.

- A schema-generation algorithm, which constructs a relational schema and STORED mappings for a semistructured data instance and, possibly, a query mix.

- An algorithm for automatic generation of STORED overflow mappings for a given relational mapping, which can exploit a DTD.

- A query- and an update-rewriting algorithm.

---

1 We may also consider as input a query mix, but that size is negligible compared to the data.

---

**Figure 1:** An instance of semistructured data. Values and object identifiers are omitted.

**Figure 2:** Textual representation of data

**Example Mapping.** Our semistructured model is an ordered version of the OEM model [16]. Data consists of a collection of objects, in which each object is either complex or atomic. A complex object is an ordered set of (attribute, object) pairs, and an atomic object is an atomic value of type int, string, video, etc. Hence, data is a graph, with edges labeled by attributes and some leaves labeled with atomic values. Data is exchanged in a text representation: the data graph in Fig. 1 is represented textually in Fig. 2. The order of an object’s attributes is the only difference from the OEM model, and we use the order only when storing the data. Any order will do; it can be the order in the text representation or obtained in some other way.

The text representation specifies the data in a tree-like format. To specify arbitrary graphs, we write references to object identifiers, e.g., the value of the Audit.company.owner attribute in Fig. 2 is the object &q24. In this paper, we consider the data to be a tree. Object identifiers in the text representation are optional. If no object identifier is specified, an object is assigned a unique identifier automatically. These assumptions are consistent with XML.
The following is an example of the kind of mapping to a relational schema on which we base our technique. One choice of a relational schema and its corresponding instance is in Fig. 3. We separate objects by their "types": taxpayers and companies are stored separately. We separate taxpayers with a complex address from those with a string-valued address. Even after this decomposition, objects are not uniform: there are many nulls. Table taxpayer1 has two attributes audit1 and audit2 to accommodate objects with two occurrences of the audit attribute. Most object identifiers from the semistructured data are omitted. The actual "mapping" is not explicitly defined, but implied by the choice of attribute and table names. For instance, the path Audit.taxpayer.name is mapped to both name in Taxpayer1 and name in Taxpayer2, and the path Audit.taxpayer.audited is mapped to audit1 and audit2 in Taxpayer1, and to audit in Taxpayer2.

Data like that in Fig. 3 can be managed by any RDBMS. Unlike semistructured data, the schema is not stored with the data. Of course, this choice is not unique or necessarily the best. For example, we could store all taxpayers in one relation, and split their addresses, depending on their structure. Some updates to the semistructured data instance cannot be accommodated by the chosen relational storage. For example, we cannot add a new taxpayer with a phone attribute. Instead, we store that data in an overflow graph. Any semistructured data repository can store the overflow graph. Efficiency is not crucial, because the overflow graph should be small. System issues arise from the integration of a relational storage with a semistructured overflow, but they are not addressed in this paper.

In Sec. 2, we introduce the STORED language, and in Sec. 3, we describe an algorithm for automatically generating the relational STORED mappings from a data instance. Sec. 4 shows how to generate automatically the overflow mappings. Sec. 5 describes query and update rewriting algorithms, and Sec. 6 reports some experimental results.

2 STORED

We describe STORED (Semistructured TO Relational Data). A relational schema is a collection of relation names $R_1, \ldots, R_m$ with attributes $A_1, \ldots, A_m$. An overflow schema is a collection of graph names, $G_1, \ldots, G_k$. Conceptually, each graph $G_i$ contains a unary relation $G_i.root(X)$ and a ternary one $G_i.edges(X,Y,L)(X,Y$ object identifiers, $L$ attribute name). A mixed schema contains a relational schema and an overflow schema. A STORED mapping translates semistructured data into instances of some mixed schema. STORED is more restrictive than other query languages for semistructured data: it doesn't have joins or regular-path expressions. This ensures losslessness: it must be possible to reconstruct any semistructured data instance from its transformation over the mixed schema.

Simple Storage Queries. A STORED mapping consists of one or more FROM WHERE STORE queries. The FROM clause is a single pattern that binds several variables. The STORE clause states how values bound to variables are stored. In this example, nested addr objects are flattened into the 4-ary relation Taxpr1:

```
M1 = FROM Audit.taxpayer : $X
    { name : $N, addr : {street : $S, zip : $Z}}
STORE Taxpr1($X, $N, $S, $Z)
```

Each FROM clause defines a unique key variable; by default, it is the first variable in the pattern, e.g., $X$ above. Each binding of the key variable that matches the pattern causes one tuple to be stored by the STORE clause. Patterns are sequences of attribute constants. All variables in the FROM clause must be used in the STORE clause, but intermediate variables may be removed from the pattern, i.e., it's not necessary to write $A$ in $addr:A$ {street:$S$, zip:$Z$}.

We can also store optional attributes:

```
M2 = FROM Audit.taxpayer : $X
    { name : $N, addr : {street : $S, zip : $Z} }.
    OPT { audited : $A, taxamount : $T}
STORE Taxpr2($X, $N, $S, $Z, $A, $T)
```

Here name, addr.street and addr.zip are required attributes, therefore the first four columns of Taxpr2 are always non-null, but audited and taxamount are optional. When one of them is missing, the last two attributes in Taxpr2 are null. We can check for audited and taxamount independently:

```
M3 = FROM Audit.taxpayer : $X
    { name : $N, addr : {street : $S, zip : $Z} }
    OPT {audited : $A, taxamount : $T}
STORE Taxpr3($X, $N, $S, $Z, $A, $T)
```

Each STORED pattern has a required subpattern and an arbitrary number of OPT subpatterns; the latter can be nested (i.e., each OPT subpattern has its required subpattern and other OPT subpatterns). The matching succeeds at an object $x$ if the required subpattern matches that object; a successful matching binds the required variables. The OPT subpatterns are tentatively matched too, starting from their required subpatterns.

A STORED mapping may contain several queries. This example shows how to cluster taxpayers into two relations, using combinations of their attributes:
M4a = FROM Audit.taxpayer : $X
   { name : $N, addr : $P,
     OPT{audited : $A}, OPT{taxamount : $T} }
   WHERE typeOf($P, "string")
STORED Taxpr4a($X, $N, $P, $A, $T)

M4b = FROM Audit.taxpayer : $X
   { name : $N, addr : { street $S, city $C, OPT{zip $Z}} }
   OPT{audited : $A}, OPT{taxamount : $T} 
STORED Taxpr4b($X, $N, $S, $A, $Z, $A, $T)

Taxpayers with an addr attribute of type string are stored in Taxpr4a and the others in Taxpr4b. In the latter case, street must be present, but city is optional. When city is present, then zip is optional. Each STORE clause must refer to a distinct relation (Taxpr4a and Taxpr4b in the example), to simplify reconstructing the semistructured data.

In this example, only one of M4a or M4b will succeed for a given object, because the conditions on addr are mutually exclusive. In general, several queries may succeed, and a single object may be stored in multiple relations. This replication can be desirable when rewriting queries, because it permits the system to choose the target relation that best matches the query, but it incurs a higher cost in disk space.

Multiple Attributes. One characteristic of semistructured data is that objects may have multiple occurrences of the same attribute; for example, a person may have two phone attributes and several subordinate attributes. We distinguish two classes of multiplicity occurring attributes. A small-set attribute usually has low cardinality and most often is one; e.g., most people have one phone, but some may have two. A collection attribute denotes a collection of objects, usually with high cardinality; e.g., bosses usually can have many subordinates. For small-set attributes, we may increase the number of columns in a relation to accommodate all occurrences, whereas for collection attributes, we may store the attributes in a nested or separate relation.

We show how to express both classes in STORED. Below, the audited attribute occasionally occurs twice:

M7 = FROM Audit.taxpayer : $X
   { name : $N, audited : $A1, OPT{audited : $A2} }
STORED Taxpr7($X, $N, $A1, $A2)

M7 stores objects with at least one audited attribute; the value of $A2 may be null. In a declarative semantics, when some object has two audited attributes with values u, v, then both u, v and v, u are valid bindings for $A1, $A2. However, in our context, it suffices to store a single permutation in Taxpr7. Here, we use the order in the data model. Conceptually, STORED queries are rewritten such that each occurrence of an attribute name a is uniquely indexed as a[1], a[2], a[3], ...:

M8 = FROM Audit.taxpayer : $X
STORED Taxpr8($X, $A1, $A2)

Only attributes below the key variable are enumerated: Audit and taxpayer are not. For each data object x, we enumerate all its attributes; e.g., its audited attributes become audited[1], audited[2], etc. This guarantees the matching between the pattern and the data is unique.

Collection attributes can be stored in nested relations or as separate relations, if nested relations are not supported. In this example, we assume that each irscennter has a collection attribute hearing:

M11a = FROM Audit.irscenter : $X
   { centername : $N, centeraddress : $A}
STORED IrsCenter($X, $N, $A)

M11b = FROM Audit.irscenter:$X.hearing:$Y
   { hearingdate : $D, taxpayername : $TN, 
     auditorname : $AN, decision : $Z} 
KEY $Y
STORED Hearings($Y, $X, $D, $TN, $AN, $Z)

This stores a many-to-one mapping from Hearings to IrsCenter. M11b’s key variable is $Y, which is not the first variable, so it must be declared explicitly. STORED requires that the key variable be outside any { ...} subpattern: e.g. $TN cannot be declared to be the key variable in M11b.

Label Variables. Some instances of semistructured data store data as attributes. For example a person’s name could be an attribute name, like in:

Data: { john : { phone : 5551234, fax : 5551236},
       Joe : { phone : 5552345, fax : 5552346}, ... }

STORED supports label variables which are stored in relations as values. Label variables must occur before the key variable, for example:

FROM Data.$L: $X { phone : $P, fax : $F}
STORE R($X, $L, $P, $F)

Overflow Queries. So far we have described relational queries, whose target are the relations of the mixed schema. Next, we describe overflow queries, targeting the overflow graphs. For example, consider the relational query below:

M15a = FROM Audit.taxpayer : $X
   { name : $N, OPT{address : {street $S, city $C}}} 
STORED R($X, $N, $S, $C)

We could complement it with the following two overflow queries, storing other attributes besides name, address.street, and address.city:

M15b = FROM Audit.taxpayer : $X
   { name : $N, OPT{address : $A}. 
     $L: _} 
OVERFLOW G1($L)

M15c = FROM Audit.taxpayer : $X
   { name : $N, address : {street $S, city $C, $X: _}} 
OVERFLOW G2($C)

Syntactically, overflow queries resemble relational ones, except that they must have one attribute variable that occurs last in the pattern, which is “stored” in the overflow graph by the OVERFLOW statement. We

3To be precise: a zip without a city is not stored, while a city without a zip is stored.
illustrate the semantics on M13b. Assume that $\$X$ is bound to some object identifier $x$. After matching name and possibly address, $\$L$ is bound successively to all other attributes. For each binding of $\$L$ to some attribute name 1 with value $y$: (1) $x$ is stored in $\text{G.roots}$, (2) the edge $(x,1,y)$ is stored in $\text{G.edges}$, and (3) the subtree rooted at $y$ is stored in $\text{G.edges}$. Note that if some object $\$X$ has several name or address attributes, the first one is stored in $\text{R}$, and all others are stored in $\text{G1}$. Similarly, M13c stores in $\text{G2}$ all subobjects of address, except the first occurrence of street and city. An object can be reconstructed from $\text{R}, \text{G1}, \text{G2}$.

As a convenience, the value of the key variable is always stored in $\text{G.roots}$, even if it is not the actual root of the subgraph being stored. For example the following are stored by M13c: $\text{G2.roots}($$\$X$$)$ and $\text{G2.edges}($$\$X$$,$$\$X$$,$$\$K$$,$$\_$$)$ (where $\_\_$ is the actual value of the $$\$K$$ attribute).

Overflow queries cannot have nested patterns other than those containing the attribute variable.

In addition to storing relations in a relational repository, the overflow graphs are stored in a semistructured data object repository. Integrating the relational and overflow systems is necessary to preserve the flexibility of the original semistructured data. The performance requirements for the overflow system, however, are less demanding than for a stand-alone object repository, because the relational system handles most of the data.

3 Generating Storage Mappings

We now describe how to generate automatically the relational STORED mapping $\mathcal{M}$ given a semistructured data instance $\mathcal{D}$ (the overflow mapping is deferred to Sec. 4). Several competing goals determine $\mathcal{M}$’s effectiveness. First, we want to limit the number of tables. Although many RDBMs do not limit the number of tables, storing each object in a separate table is undesirable. Second, we want to bound the disk space. Although the size of the data instance $\mathcal{D}$ is fixed, its relational storage may be arbitrarily large, because an object may be stored in more than one relation. A related goal is minimizing the number of nulls. Some RDBMS store nulls efficiently, e.g., a null entry in a record requires only a byte, and nulls at the end of the record take no space. Nonetheless, we do not want to generate wide, sparse tables, since even one byte per null entry becomes expensive. A related restriction is that some RDBMS impose an upper limit on the number of attributes per table. Depending on the application, other goals may include reducing object splits and their redundant storage in multiple relations, or, on the contrary, increasing object redundancy to improve query evaluation.

These goals are best modeled as a cost-optimization problem. Given a data instance $\mathcal{D}$, generate a STORED mapping $\mathcal{M}$ that minimizes a storage-cost function $c(\mathcal{M})$, the cost of storing $\mathcal{M}(\mathcal{D})$. The generator must also accommodate hard constraints, e.g., the limit of attributes per table. A query mix, $Q = \{Q_1, \ldots, Q_k\}$, can also be considered. For a given STORED mapping $\mathcal{M}$, each query $Q_i$ can be rewritten as a relational query.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>Max tables</td>
</tr>
<tr>
<td>$\lambda_T$</td>
<td>Max attributes per table</td>
</tr>
<tr>
<td>$S$</td>
<td>Max disk space</td>
</tr>
<tr>
<td>$C$</td>
<td>Collection size threshold</td>
</tr>
<tr>
<td>Supp</td>
<td>Min Support</td>
</tr>
</tbody>
</table>

Table 1: Storage-generation parameters $Q_i$, on $\mathcal{M}(\mathcal{D})$ (Sec. 5). Given a weight $f_i$ for each query $Q_i$, and a query cost function $d(Q_i)$ that denotes the cost of evaluating $Q_i$ on the relational data, a second goal is to minimize the query-cost function, $d(M) = \sum_{i=1,k} f_i d(Q_i)$. The optimization problem can now be applied to the combined cost $c(\mathcal{M}) + d(\mathcal{M})$.

Unfortunately, the storage-cost cost optimization problem is NP-hard in the size of the input data (by reduction from the *rectilinear picture compression* problem [8]).

**Theorem 3.1** The problem of computing an optimal storage mapping $\mathcal{M}$ is NP-hard in the size of the semistructured data $\mathcal{D}$.

This is a daunting complexity. Typically, query optimization problems are NP-complete in the size of a query, but here the problem is NP-hard in the size of the data. Search algorithms like dynamic programming are unlikely to work, so we consider heuristics, starting from frequent patterns in the data discovered by data mining. Wang and Li [21] describe a data-mining algorithm for semistructured data. We review it briefly and refer to it as *WL’s algorithm*.

**WL’s Data-Mining Algorithm.** WL’s data model is a large collection of semistructured objects, i.e., a graph with many roots. WL’s algorithms searches for *tree patterns*, which are trees consisting of attribute constants and the symbol "*"", which means any attribute. Attributes are indexed in WL, to allow for multiple attribute occurrences. An example of a WL pattern in our notation is:

\{name[1], phone[1], phone[2], address[1]: \{street[2], city[1], zip[1]\}\}

The support of this pattern is the number of root objects that contain at least name, two phones, and an address, with the latter containing at least street, city, zip. Given a particular semistructured instance and a minimum support, WL’s algorithm has two steps. First, it finds all paths with high support: the set of such paths is called $F_1$. The second step is an adaptation of the standard apriory algorithm [2] in which items are replaced by paths, and itemsets by tree patterns (in WL’s setting each ordered set of $k$ paths uniquely corresponds to a pattern tree with $k$ leaves). The algorithm generates successively the sets $F_2, F_3, \ldots, F_k, \ldots$ where each $F_k$ consists of sets of $k$ paths, whose associated pattern trees have high support.
Algorithm: Automatic Storage Generation

**INPUT:** X, A, S, C, Supp, and query mix Q

**OUTPUT:** Set of relational STORED mappings

**METHOD:**

1. Find all minimal prefixes with data support \( \geq \text{Supp} \)
2. Run the WL data mining algorithm with the changes in the text.
   - Let \( K' \) = number of maximally contained patterns found
3. Select \( K_0 \) (\(< K') patterns out of the \( K' \)
4. For each of the \( K_0 \) patterns, select the set of "required" attributes
5. For each of the \( K_0 \) patterns with required attributes, generate one or more STORED relational queries.

Figure 4: Automatic Storage Generation Algorithm

We explain each step of the algorithm, which appears in Fig. 4.

**Step 1:** Compute minimal path prefixes.
- First, we generate all prefixes \( 1_1, 1_2, \ldots, 1_k \) with support \( \geq \text{Supp} \). This requires a single pass through the data, during which we construct a trie structure in memory that encodes all prefixes in the data. Each trie node uniquely corresponds to a prefix, and has only those outgoing attributes \( a \) that were discovered in the data, plus \(-\) (the wildcard). We start with the empty trie and extend it as we traverse the data. Each trie node stores the support for that prefix. Once a trie node reaches minimal support \( \text{Supp} \), we delete the tree underneath, and do not further expand that node.
- Each prefix with high support identifies the collection of objects in the semistructured-data instance \( D \) that become the root objects for the data-mining algorithm in Step 2.

**Step 2:** Data mining.
- We use WL’s data mining algorithm with the following changes. First, we compute both the data support and the combined support (data plus query support). The algorithm is guided by the combined support, i.e., patterns are grown as long as their combined support is large. Second, we keep backpointers to subpatterns with high data support; this information is used in Step 3. Recall that WL’s algorithm generates a new set \( P \) of \( k \) paths in \( F_k \) by combining two sets of \( k - 1 \) paths in \( F_{k-1} \). We store one backpointer in \( P \) to whichever of its parent pattern has higher data support. This means some non-maximal patterns in \( F_{k-1} \) cannot be deleted at the end of phase \( k \), which increases memory usage. The increase, however, is not prohibitive, e.g., there will be \( k \) additional sets retained for each set in \( F_k \). Finally, we terminate the algorithm either when the combined support decreases below \( \text{Supp} \) or when \( k \) reaches \( A \).

**Step 3:** Select \( K_0 \) patterns. From Step 2, we have \( K' \) maximally contained patterns, each with support \( \geq \text{Supp} \). Recall that WL’s algorithm also retains \( F_1 \), the set of highly supported paths. Now we use a greedy approach to select \( K_0 \) patterns for further processing.
algorithm to select a subset of no more than K of the K' patterns that best cover the paths in F_1. We sort F_1 by the data support and start by picking any pattern P_1 that covers (i.e., contains) paths in F_1 with highest support. In general, we pick pattern P_k such as to (a) minimize the maximum overlap with P_1,...,P_{k-1}, and (b) in case of ties, cover the paths with highest support in F_1 which is still uncovered. We stop when we cover all paths in F_1 or when k reaches K (the maximum number of relations allowed).

Step 4: Select required attributes. For each of the K0 patterns P selected in Step 3, we have a chain of backpointers to strictly smaller subpatterns, P = R_1, R_2,...,R_n, with increasing data support (with R_n in F_1). In Step 4, we choose some i for each pattern so that the subpattern R_i becomes the required part of P. We attempt to choose a small i, i.e., a pattern with the fewest attributes, because such a pattern will match more semistructured objects in the mapping associated to P.

Choosing a small i, however, will increase the number of nulls and the overlap with the set of objects stored in different relations, and therefore increase the total disk space. For a given i, the disk space requirement for P can be computed from the data supports. In Step 4, we initialize the i counter for each pattern to 1 and increment them in a round-robin fashion, until we are within the limit for the allowed disk space S. In addition, we stop incrementing the i's before two patterns end up containing each other's required part: we allow one to contain the required part of the other, but then we will stop reducing its required part.

Step 5: Generate STORED queries. Each storage pattern P, with its required subpattern R, is converted into one or more STORED queries. This is pretty straightforward. For example, for the following pattern/subpattern:

```latex
P = Audit.taxpayer: {name[1], phone[2]}
   address[1]: {street[1], city[1]}
```

we associate the following STORED queries:

```latex
M1 = FROM Audit.taxpayer: $X
   { name: $X, phone: $P1, 
     OPT { phone: $P2 }}

STORE R1($X, $X, $P1, $P2)
```

```latex
M2 = FROM Audit.taxpayer: $X.address: $Y
   { street: $S, city: $S}

STORE R2($X, $Y, $S, $SC)
```

A collection attribute like `address[*]` results either in a nested relation (if the RDBMS supports it), or in a separate relation: in the example we assumed the latter, hence the separate relation R2.

4 Generating Overflow Mappings

In Sec. 3, we showed how to generate automatically the relational mapping from a data instance. Next we show how to construct the accompanying overflow mapping, which ensures that the storage is lossless.

When nothing is known about the semistructured data overflow queries are always needed and tend to be complex. In practice, we found that specifying overflow queries is much simpler given information about the data's structure. Such information is sometimes available, for instance, the structure of XML data is specified by a DTD. Hence we discuss overflow mappings in the context of semistructured data schemas.

Consider the schema:

```latex
S1 = SCHEMA {Audit : 
   { (taxpayer: {name: String,
                     address: String,
                     (extension: String)?,
                     (taxreturn: S2)+ })
   })+
```

```latex
S2 = SCHEMA {year: String, amount: String,
              (extension: String)?}
```

It specifies that the data has arbitrarily many taxpayers, each with one name, one or two addresses, and arbitrarily many taxreturns. The latter are of "type" S2, meaning that they have a year, an amount, and zero or one extensions. Multiple types can be defined and recursion is permitted. As this example illustrates, schemas may contain regular expressions, with the usual operators (alternation "|", Kleene star "*", zero-or-one "?", concatenation " ", ), like in DTD's. However, unlike DTDs, these regular expressions are unordered [9].

Schemas may also contain attribute variables, and simple inequalities may be imposed on these variables. For example:

```latex
S3 = SCHEMA {Audit: 
   { (taxpayer: {name: String, address: 
                 { street: String, zip: int, (K: Any)+},
                 (office: String, office: String)?}),
     WHERE $K < zip, $L not in {name, address}
   } Any = SCHEMA {($P: Any)+}
```

Here taxpayers can have exactly one name and address, and any number of additional attributes different from name, address. Note that Any is a schema for _any_ semistructured data. This is of particular interest for us: when nothing is known about the data, we assume that the schema is Any.

Schemas like these presented here lead quickly to high complexity [3], although much of their power is not needed in practice. For instance, they allow us to define useless types like S4 = SCHEMA {name: String, (office: String, office: String)*}, which specifies that a person has an even number of offices. We consider here restricted schemas. A restricted schema is a schema in which every regular expression is a concatenation of expressions of the form (a:T), or (a:T)?, or (a:T)*. The schemas S1, S2, S3, Any, defined above, are restricted schemas, but S4 is not. Every schema S can be converted into a restricted schema S' by performing the following transformations repeatedly (here p, p',... denote pairs (a:S), (a'S:S'),...):

$$(p | p') \rightarrow p', p'$$  
$$(p,p')? \rightarrow p', p'$$  
$$(p') ? \rightarrow p$$  
$$(p') ? \rightarrow p$$
INPUT Relational mapping \( M \), schema \( S \)
OUTPUT Overflow mapping \( O \)

METHOD

Step 1: for each attribute \( a \) in \( S \)
construct set of databases \( D_a \)
Step 2: for each \( D \in D_a \)
compute \( M \) on \( D \)
if \( a \) is not stored by \( M \)
then generate overflow mapping

Figure 5: Generation of overflow mappings

In the remainder of this section, we use an abbreviated notation for restricted schemas, indicating the attributes’ ranges. For example, schema \( S_1 \) is denoted:

\[
S_1:\text{Audit}[1]: \{ \text{taxpayer}[0,\ast]:
\text{name}[1], \text{address}[1,2],
\text{taxreturn}[0,\ast]:\{\text{year}[1], \text{amount}[1], \text{extension}[0,1]\}\}
\]

Each range is \([i, j]\), with \( i \) a number and \( j \) a number or \(*\). When \( i=j \), the range is abbreviated by \([1]\).

We describe next the algorithm for overflow generation, for a given relational STORED mapping \( M \), and a restricted schema \( S \). For presentation purposes we assume \( S \) is non-recursive, has no variables, and each attribute occurs at most once in \( S \). Thus \( S \) is equivalent to a tree, with each attribute occurring exactly once, labeled with a range denoted \( \text{range}(a) \). We will relax these assumptions later.

We start by indexing all attribute names in \( M \), as in Sec. 2. For each attribute name \( a \) in \( M \), define \( \text{high}(a) \) to be its maximum index: when \( a \) is not indexed (occurs before the key variable), then \( \text{high}(a) = 0 \). We illustrate our algorithm on the following STORED mapping:

\[
M = \text{FROM Audit}.\text{taxpayer}: \$X
\{ \text{name}[1]: \$N, \text{address}[1,2]: \$A, \\
\text{OPT taxreturn}[1]: \text{year}[1]: \$Y, \text{amount}[1]: \$A, \text{extension}[1]: \$E\}
\]

\[
M' = \text{FROM Audit}.\text{taxpayer}: \$X, \text{taxreturn}: \$Z
\{ \text{year}[1]: \$Y, \text{amount}[1]: \$A\}
\]

\[
\text{STORE } Q(\$X, \$Z, \$Y, \$A)
\]

We have:

\[
\text{high(Audit)} = 0, \text{high(taxpayer)} = 0, \text{high(name)} = 1, \\
\text{high(address)} = 1, \text{high(year)} = 1, \\
\text{high(amount)} = 1, \text{high(extension)} = 1
\]

The algorithm is shown in Fig. 5 and is illustrated in Fig. 6 on the schema \( S_1 \) and the relational STORED queries \( M, M' \). Step 1 constructs a set \( D_a \) of canonical databases, for each attribute \( a \) in \( S \). Intuitively these databases “challenge” \( M \) to prove that it can store the attribute \( a \). \( D_a \) is obtained by recursively traversing the tree associated to \( S \), and creating for each attribute \( b \) several copies, labeled \( b[1], b[2], \ldots, b[k] \), and possibly \( b[*] \), as follows. Let \( \text{range}(b) = [i, j]\):

- When \( b \neq a \) and \( b \) is not an ancestor of \( a \) in \( S \), then we create \( i \) copies \( b[1], b[2], \ldots, b[i] \).
- When \( b = a \) or \( b \) is an ancestor of \( a \) in \( S \), then we choose a certain \( k \) and create copies labeled \( b[1], b[2], \ldots, b[k] \). Each value of \( k \) results in a different database in \( D_a \). Let \( \text{high}(b) = m \). When \( j = \ast \) then \( k \) is chosen to be each of \( i, i+1, \ldots, m, * \). When \( j \) is a number, then \( k \) is chosen to be each of \( i, i+1, \ldots, \min(j, m+1) \).

Fig. 6 depicts the construction of \( D_a \) for most attributes \( a \) (the other attributes, Audit, taxpayer, taxreturn, are similar).

In Step 2, the algorithm evaluates \( M \) on each canonical database \( D \in D_a \), and checks whether \( M \) stores the value of every attribute \( a \): if not, then an overflow query is generated. In Fig. 6, this happens in two cases: \( \text{address}[2] \) in \( D_{\text{address}} \) and \( \text{extension}[1] \) in \( D_{\text{extension}} \) (both edges are emboldened). The corresponding overflow queries are generated from \( D \) in an obvious fashion. In our example the two queries are:

\[
01 = \text{FROM Audit}.\text{taxpayer}: \$X
\{ \text{name}: \$W, \text{address}: \$A, \text{taxreturn}: \$T\}
\text{WHERE } \$L = \text{address}
\text{OVERFLOW G1(\$L)}
\]

\[
02 = \text{FROM Audit}.\text{taxpayer}: \$X
\{ \text{name}: \$W, \text{address}: \$A, \text{taxreturn}: \$T, \\
\text{taxreturn}: \{ \text{year}: \$Y, \text{amount}: \$A, \text{taxreturn}: \$T\}
\text{WHERE } \$L = \text{extension}
\text{OVERFLOW G2(\$L)}
\]

We address now the restrictions in our algorithm. First, recursive types in \( S \) are handled by unfolding. Although, in general, this results in an infinite tree, only a finite portion needs to be considered, because the queries in \( M \) are non-recursive (have no regular-path expressions). Second, label variables \( \$L \) in \( S \) will occur as constants in the canonical databases. Conceptually, \( \$L \) can be one of the attribute constants occurring in the
Algorithm: Query rewriting

Input: Stored mapping M, user query Q
Output: Set of rewritten queries Q1, ..., Qk over the mixed schema

Method:

Step 1 (preprocessing)
- From M, construct inversion rules I
- From I, construct canonical data instance D

Step 2 Compute Q on D, ignoring the WHERE conditions

Step 3 For each answer, compute all minimal covers of the answer with inversion rules

Step 4 For each cover, generate a query Qi by adding WHERE conditions from Q, from gluing the inversion rules, and from I

Figure 7: Rewriting algorithm

query and schema, or anything else, and the algorithm takes this into account when evaluating M. Finally, repeated attributes in the schema are handled by replacing them with the concatenation of all attributes above them.

5 Query and Update Rewriting

Given a STORED mapping, the system accepts queries and updates over the semistructured data and rewrites them into queries and updates over the relational store, and the overflow graphs. This section describes the rewriting algorithm. We start first with query rewriting, then address update rewriting.

The algorithm appears in Fig. 7. We explain it below and illustrate it on the STORED mapping M:

Ma = FROM Audit.taxpayer: $x
{ name[1] : $fn,
  lastname[1] : $ln,
  addr[1] : {street[1] : $s,
    city[1] : $c},
  OPT(taxamount[1] : $t)}
STORE Taxpayer($x, $fn, $ln, $s, $c, $t)

Mb = FROM Audit.taxpayer: $x
{ addr[1] : {street[1] : $s,
    city[1] : $c, $l : _}},
OVERFLOW G1($l)

Mc = FROM Audit.taxpayer: $x
{ name[1] : $fn,
  taxamount[1] : $t},
SL : _,
OVERFLOW G2($l)

Our example query Q returns the names of taxpayers whose taxamount is less than 10% of their income on some W4 form and whose address contains "Philadelphia":

Q = SELECT $x
FROM Audit.taxpayer: $x
{ name : $fn, taxamount : $t, W4form.income : $i, address : $a}
WHERE $t < 0.1 * $i, $a = "Philadelphia"

Note that the query returns a set of name oids, each with firstname and lastname.

The queries we consider have patterns with regular expressions (in the FROM clause), and arbitrary conditions in the WHERE clause (i.e., joins are allowed).

These are common features of all query languages for semistructured data [17, 4, 7, 6].

Step 1 is a preprocessing step and starts by constructing inversion rules for the queries in M. This step is an adaptation from [14], where inversion rules were first used in query rewriting. In our context, the rules are generated internally by the system and are not intended for the user. We describe them in a concrete syntax for presentation purposes only. Each rule consists of a FROM, WHERE, and CONSTRUCT clause. The first two are as before; the CONSTRUCT clause contains a tree constructor, in which node oids are generated by Skolem Functions which, in turn, correspond to attribute names (including indexes). For our example, the inversion rules are shown in Fig. 8, and the Skolem functions are SAudit, S_taxpayer, S_name1, etc. For each relational query in M, one inversion rule is created for the required subpattern, and one inversion rule for each OPT subpattern. For each overflow query, a single inversion rule is created for its required subpattern (and OPT subpatterns are ignored). Thus, Ma0 and Ma1 are created for Ma, and Tb and IC are created from Mb and Mc respectively. As in [14], inversion rules have the property that they reconstruct the entire semistructured data.

Step 1 continues by constructing a single canonical data instance D from the inversion rules. This is achieved by “fusing” all symbolic objects in the CONSTRUCT clauses of all inversion rules (Fig. 9). Its nodes are labeled with Skolem function names or overflow graph names, and edges are either unlabeled (when the target is an overflow graph) or labeled with an attribute name. In addition, edges are annotated with the name of the inversion rule (e.g. [Ma0], [Ma1], [Tb], [Tc]).

Steps 2 and 3 do the actual rewriting (this is related to query decomposition and algebraic optimization [15] for the Mediator Specification Language, MSL). Step 2 is a simple computation of the query on D, during which the conditions in the WHERE clause are not checked, and
unlabeled edges in $D$ may match any attribute in the query. For our example, the result of the evaluation is:

<table>
<thead>
<tr>
<th>$S_X$</th>
<th>$S_N$</th>
<th>$S_T$</th>
<th>$S_L$</th>
<th>$S_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{taxpayer}$</td>
<td>$S_{last_name}$</td>
<td>$S_{taxamount}$</td>
<td>$G_2$</td>
<td>$S_{street}$</td>
</tr>
<tr>
<td>$S_{taxpayer}$</td>
<td>$S_{first_name}$</td>
<td>$S_{taxamount}$</td>
<td>$G_2$</td>
<td>$S_{city}$</td>
</tr>
<tr>
<td>$S_{taxpayer}$</td>
<td>$S_{last_name}$</td>
<td>$S_{taxamount}$</td>
<td>$G_2$</td>
<td>$G_1$</td>
</tr>
<tr>
<td>$S_{taxpayer}$</td>
<td>$S_{first_name}$</td>
<td>$S_{taxamount}$</td>
<td>$G_2$</td>
<td>$G_1$</td>
</tr>
<tr>
<td>$S_{taxpayer}$</td>
<td>$G_2$</td>
<td>$S_{taxamount}$</td>
<td>$G_2$</td>
<td>$S_{street}$</td>
</tr>
<tr>
<td>$S_{taxpayer}$</td>
<td>$G_2$</td>
<td>$S_{taxamount}$</td>
<td>$G_2$</td>
<td>$S_{city}$</td>
</tr>
<tr>
<td>$S_{taxpayer}$</td>
<td>$G_2$</td>
<td>$G_2$</td>
<td>$G_2$</td>
<td>$G_1$</td>
</tr>
<tr>
<td>$S_{taxpayer}$</td>
<td>$G_2$</td>
<td>$G_2$</td>
<td>$G_2$</td>
<td>$G_1$</td>
</tr>
</tbody>
</table>

Step 3 considers each row in the answer relation, and finds all minimal covers of the corresponding subgraph of $D$ by the inversion rules. Consider the first row above, the edges of the corresponding subgraph of $D$ are highlighted in Fig. 9, and the unique minimal cover is \{ $I_a0$, $I_a1$, $I_c$ \}

Finally, for each minimal cover, Step 4 constructs a query over the mixed schema. The query is essentially obtained by joining all inversion rules in the cover (which results in new WHERE conditions), and adding conditions from the original query $Q$. For our example, the resulting query is:

```sql
SELECT $S_{name}$(X)
{ firstname : $S_{first_name}$(X), $S_{fn}$, // reconstruct last_name : $S_{last_name}$(X), $S_{ln}$ } // name
FROM $Taxpayer$(X, $S_{fn}$, $S_{ln}$, $S_S$, $S_C$, $S_T$). // $I_{a0}$, $I_{a1}
G2.roots($X$), $X$.w4form.income $S_I$ // $I_c$
WHERE $S_I$ != null. // from $I_{a1}$
$S_X$ = $X'$. // glue $I_{a1}$ with $I_c$ at $S_{taxpayer}$
$S_T$ < 0.1 + $I_1$, $S_S$ = "Philadelphia" // from $Q$
```

This query checks "Philadelphia" in the street position; another query would check for it in the city position. The query reconstructs the name using Skolem functions. Its FROM clause contains mixed relational goals (like $Taxpayer(...)$) with semistructured data patterns on the overflow graphs. The WHERE clause may have join conditions, like $S_X = S_X'$.

**Updates.** We consider a strict subset of the Lory's [1] update statements:

- `UPDATE <object-selector> <- <object-expression>`
- `UPDATE <object-selector> := <value-expression>`

In both statements `<object-selector>` is a query whose result must be a single object identifier $o$, and `<object-expression>` is a constant expression denoting a complex-value object $o'$ (syntax like in Fig. 2). The first statement adds (attribute, value) pairs of $o'$ to $o$ (to the end of the order). The second statement replaces the value of the atomic object $o$ with `value-expression`. We describe the rewriting of the first update statement; the second is easier and handled similarly.

Update rewriting has two steps. First, `<object-selector>` is evaluated on the canonical database $D$. We illustrate with the STORED mapping $M$ above and the following update:

```sql
UPDATE (SELECT $X$
FROM $Audit$.taxpayer:$X$ { name.lastname:$N$ } WHERE $S_N$ = "Smith") += {taxamount : "$100000"}.
```

Referring to Fig. 9, there are two results in Step 1: ($X = S_{taxpayer}$, $S_N = S_{last_name}$), and ($X = S_{taxpayer}$, $S_N = S_{taxamount}$). We illustrate in the sequel with the first result only.

In the second step, for each row in the result, we extend $D$ with `<object-expression>`: this is the dotted edge in Fig. 9. Now we execute every STORED query in $M$ on the extended object, but only consider results that use at least some edge of the extension (the “dotted” edges). For each such result, we generate one update instruction on the mixed schema. We illustrate using queries $Ma$ and $Mc$. One of the bindings of $Ma$'s variables is:

- $S_X$ = $S_{taxpayer}$, $S_{fn}$ = $S_{first_name}$
- $S_{ln}$ = $S_{last_name}$, $S_S$ = $S_{street}$, $S_C$ = $S_{city}$

This corresponds to the following update:

```sql
UPDATE $Taxpayer$(X, $S_{fn}$, $S_{ln}$, $S_S$, $S_C$, $S_T$)
SET $S_T$ = 100000
WHERE $S_{ln}$ = "Smith"
```

When evaluating $Mc$, the order of the two `taxamount` attributes in Fig. 9 matters. There are two bindings for $Mc$: we consider the one that binds $S_L$ to the new `taxamount`, which means `taxamount[1]` (in $Mc$) must have been bound to the first edge. This translates into the update:

```sql
UPDATE $G2.roots($X$), $G2.edges($X$.taxamount,100000)
WHERE $Taxpayer$(X, $S_{fn}$, $S_{ln}$, $S_S$, $S_C$, $S_T$).
$S_{ln}$ = "Smith", $S_T$ != null
```

6 Preliminary Experiments

We ran some preliminary experiments with our automatic STORED generation algorithm. Our data set is DBLP, the popular database bibliography Web site\(^4\), a collection of XML-like files. It does not have an explicit structure. Each XML file corresponds to one publication: a proceedings paper, a journal article, a book, a PhD thesis, etc. The directory structure captures a lot of information. For example, the top level contains books, conf, journals, ms, persons, and phd directories. Under conf, there is one subdirectory

\(\text{http://www.informatik.uni-trier.de/ley/db/about/instr.html}\)
FROM Bib.inproceedings $X$
{ author: $A1$, OPT{author: $A2$, OPT{author: $A3$}},
  OPT{title: $T$}, OPT{pages: $PP$}, OPT{year: $Y$},
  OPT{booktitle: $B$}, OPT{url: $U$}}
STORE R1($A1$, $A2$, $A3$, $T$, $PP$, $Y$, $B$, $U$)

FROM Bib.article $X$
{ author: $A1$, OPT{author: $A2$, OPT{author: $A3$}},
  OPT{title: $T$}, OPT{pages: $PP$}, OPT{year: $Y$},
  OPT{volume: $V$}, OPT{journal: $J$}, OPT{number: $N$},
  OPT{title: $T$}, OPT{year: $Y$}, OPT{volume: $V$},
  OPT{url: $U$}}
STORE R2($A1$, $A2$, $A3$, $T$, $PP$, $Y$, $V$, $J$, $N$, $U$)

FROM Bib.article $X$
{ author: $A1$, OPT{author: $A2$, OPT{author: $A3$}},
  OPT{pages: $PP$}, OPT{year: $Y$},
  OPT{number: $N$}}
STORE R3($A1$, $A2$, $A3$, $PP$, $Y$, $N$)

Figure 10: STORED query for $A=8$

for each conference (255 in total), etc. The directory information, however, can be fully recovered from the publications themselves. A typical entry is:

```
<inproceedings key="Abiteboul97">
  <author>George Abiteboul</author>,
  <title>Querying Semi-Structured Data</title>,
  <pages>1-18</pages>,
  <year>1997</year>,
  <url>db/conf/icdt/icut97.htm#Abiteboul97</url>
</inproceedings>
```

The publication data is irregular: some entries have multiple author’s, optional url’s, or many citation attributes; a few have unfamiliar attributes. Most attributes have scalar values, but some have structure. There are about 92,000 publications that, when represented as a semistructured data, have 861,000 edges. The total disk space is 95M.

We decided to ignore the directory structure and moved all files in one directory. We chose a minimum support of 8500, which is 8.6%. Only articles and inproceedings had minimum support (books are only 307, phd thesis 67). In a separate experiment (not reported for lack of space), we also considered a query mix, in which one query with high weight referred to books. In that experiment, books did have minimum support, and the system generated a relation for storing book objects. We found no collection attributes; citation was a good candidate, but it did not have high enough support. We also found no nested attributes with high enough support.

Figure 10 contains an example of one generated STORED mapping with $A=8$. There were only 8 attributes of high support for inproceedings, and all 8 in combination still had high support: hence a single STORED query maps inproceedings. There were more than 8 attributes of high support for article, therefore these objects are split into two relations: R2, tries to cover most objects, by requiring only the author attribute, while R3, requires two authors, giving it the best chance to store objects not stored in R2.

We ran two sets of experiments: one that varied the maximum number of attributes per relation, $A$, and one that varied total disk space allocated to the relations, S. The results are shown in Table 2.

We varied $A$ from 3 to 9 (since there were 10 attributes of high support). We assessed the quality by the algorithm by measuring the number of queries, the data coverage, and the number of nulls. Queries is the total number of relational STORED queries generated (hence, relations). Coverage is the percentage of the 861k edges that are stored in the relations. Space is the estimated disk space required by the relational storage (assuming fixed-length records). Nulls represent the amount of space occupied by nulls.

The first table shows that data fragmentation directly depends on the maximum relation arity. When $A$ is small, objects need to be split across many relations (then, joined at query time). On the other hand, space is better utilized for small $A$’s, because the number of null entries is smaller. The coverage (total number of edges stored in the relational part) is consistent, at approximately 90%. The actual overflow graphs would be larger than 10% of the data, because some overlap would exist between the overflow graphs and the relational store.

The second set of results show a clear degradation of the coverage when disk space for the relations is limited. When $S$ is small, the algorithm generates more required attributes (Sec. 3): this decrease the number of nulls and to improve utilization of the relations, but covers fewer objects.

In summary, under reasonable assumptions, the generated STORED queries can cover a large percentage of the data, and they do this by exploiting the regularities found in the DBPL data instance.

<table>
<thead>
<tr>
<th>A</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queries</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Coverage (%)</td>
<td>94</td>
<td>94</td>
<td>94</td>
<td>90</td>
<td>90</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>Space (M)</td>
<td>1.25</td>
<td>1.38</td>
<td>1.45</td>
<td>1</td>
<td>1</td>
<td>0.9</td>
<td>1.2</td>
</tr>
<tr>
<td>Nulls (%)</td>
<td>25</td>
<td>16</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2: Effect of varying maximum number of attributes per relation and maximum disk space.

7 Related Work and Discussion

Data clustering is the problem of grouping a large number of points in $\mathbb{R}^d$ into sets (clusters), where the distances between any two points in the same cluster is small. BIRCH [22] produces good clusters in just a few passes over a large data set. We found the storage-generation problem hard to model as a data clustering problem, because objects with widely different structures may be well stored together. Nestorov, Abiteboul, and Motwani [13] describe a clustering-based algorithm which, when given a semistructured data, extracts a schema for that data.

Theodoratos and Sellis [18] describe a warehouse-configuration algorithm that when given a relational
database instance and a set of queries, generates an optimal set of views that best support the query set. This is related to the storage-generation problem, where we search for relational views over semistructured data. In our problem, however, the input is a data instance instead of a query set, and our “views” must be lossless.

Tsalatas et al. [19] pioneered the idea of achieving physical data independence by means of relational views. STORED follows a similar philosophy in achieving independence from the underlying relational storage.

Linguistically, STORED is closely related to query languages for semistructured data, such as Lorel [17, 1], UnQL [4], MSL [15], and StruQL [7, 6], all of which provide path expressions for matching attribute paths in semistructured data. Due to its unique requirements, STORED is strictly weaker than these languages.

Object-oriented databases [5] can store SGML and XML documents without explicitly storing their schema, but they require a DTD to derive an object-oriented schema. This can be an effective storage mechanism for XML data when the DTD is known, but in some applications, a DTD may not exist. Also, the technique can increase data fragmentation, because new classes and objects must be created to convert a DTD into a class hierarchy [10, 12]. Our technique is complementary: it does not require a DTD, and it uses a RDBMS instead of an ODBMS.

Wang and Li [21] extended data-mining techniques to semistructured data. Their algorithm finds “interesting patterns”, i.e., subtrees with high support. Data mining is a good foundation for the STORED generation algorithm, although we must search for more complex patterns and attempt to cover most of the data. If applied directly, Wang and Li’s patterns would generate a simple relational storage that covers only a small fragment of the data.

Storing semistructured data in relations is an ambitious goal, because the two models are apparently incompatible. Our hypothesis is that many semistructured data sources have a regular structure, with few outliers: this structure should be exploited when storing the data. Our preliminary experiments using the DBLP bibliography database support this hypothesis: in particular, DBLP data is quite regular, even if some outliers escape normal classification.

References