Storing Semistructured Data with STORED

Alin Deutsch*
Univ. of Pennsylvania
adeutsch@gradient.cis.upenn.edu

Mary Fernandez
AT&T Labs
mff@research.att.com

Dan Suciu
AT&T Labs
suciu@research.att.com

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. The mapping is always lossless: parts of the semistructured data that do not fit into the relational schema are stored in an "overflow" store.

We present a new query language, called STORED, to specify these mappings. The mapping can be derived automatically for a given data instance or can be defined by the programmer. We have extended data mining techniques to generate storage mappings that are efficient for a semistructured data instance and, possibly, a given query mix. We describe an algorithm that given a mapping, rewrites queries and updates on the semistructured data model to the relational and overflow stores. We are interested in applying these techniques to XML data, which is as an instance of semistructured data, and we show how a document-type descriptor (DTD) of an XML document can be used to improve our techniques.

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 [?] and Tsimmis [?] store their data as graphs; the schema is stored as attributes labeling the graph's edges. Strudel [?] 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 encode 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, qurey and manage semistructured data. Semistructured data can alway 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

- to store and manage efficiently existing semistructured data sources, and

*Part of this work was done while the author visited AT&T Labs.

- to convert relational sources into a semistructured format, such as XML.

In the first application, the semistructured-data source exists, e.g. it might be a large XML file, and the relational schema and the STORED mapping are generated automatically from patterns discovered in the data source. Subsequently, queries and updates over the semistructured view are automatically rewritten into queries and updates over the relational store. If the 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.

<==[[[do we have space for the figure here ? dan]]]

Given a semistructured data source, we have to produce 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 by 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 [**?**] or data warehouse design [**?**]), our input is the data instance, not a set of queries[1]. 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. Our algorithm includes a data-mining step; for semistructured data, data mining has been described by Wang and Li [**?**], 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.

Once the relational mapping of the STORED query has been selected, the system automatically generates the overflow mapping necessary to ensure that any semistructured instance is stored losslessly. This is a fragment of the STORED mapping specifying which objects (or parts thereof) have to be stored in the overflow graph. The mapping has to be lossless for *any* data instance, not just on the given instance, because we support updates, which are propagated to the relational store if possible, or to the overflow graph otherwise. These overflow mappings tend to be unnecessarily conservative: we show that, in the case of XML data, a DTD can be used to simplify the overflow parts considerably.

Once the complete STORED mapping is generated (automatically or manually), 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 datalog queries is a well understood problem [**?**, **?**]. Since STORED is a new query language with novel features, query rewriting needs to be revisited. We show that arbitrary queries on semistructured data, with regular expressions and tree-like patterns, can be rewritten in terms of STORED mappings. A particular concern for us are updates. We show that insertions in the semistructured data can be automatically rewritten into insertions into the relational plus overflow store.

In summary, the 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 given semistructured instance and, possibly, a query mix.

- An algorithm for automatic generation of STORED overflow mappings for a given relational mapping, which also exploits a DTD, when available.

- A query rewriting and an update rewriting algorithm.

1.1 Illustration of Relational Storage for Semistructured Data

Our semistructured model is an ordered version of the OEM model [**?**]. 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

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

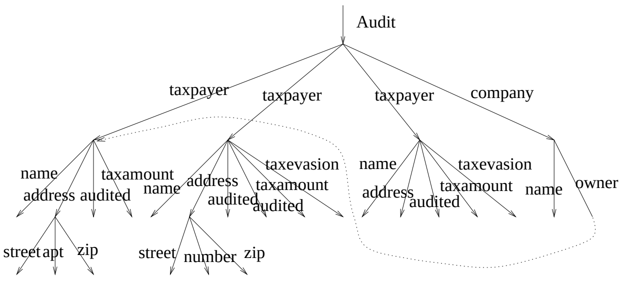

Figure 1: Illustration of an instance of semistructured data. Values and object identifiers have been dropped to reduce clutter.

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 `owner` attribute in Fig. 2 is the object `&o24`. Hence, throughout the paper we consider the data to be a tree. Object identifiers in the text representation are optional; for example, the last `taxpayer` could be written as:

```
taxpayer :
  {name : "Korolev", address : "Baikonur, Russia",
   audited : "10/12/86", taxamount : 0,
   taxevasion : "likely"}
```

If no object identifier is specified, an object is assigned a unique identifier automatically. All these conventions and 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 separated objects by their "types": `taxpayers` and `companies` are stored separately. Within `taxpayers`, we separated those with a complex `address` from those whose `address` is a string. 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`.

A data instance like in Fig. 3 can be managed directly by any RDBMS. Unlike semistructured data, the schema is not stored with the data. Of

```
Audit: &o1
  {taxpayer: &o24
     {name : &o41 "Gluschko",
      address : &o34 {street : &105 "Tyuratam",
                      appartment : &o623 "2C"
                      zip : &121 "07099"}
      audited : &o46 "10/12/63",
      taxamount : &o47 12332},
   taxpayer : &o21
     {name : &o132 "Kosberg",
      address : &o25 {street : &427 "Tyuratam",
                      number : &928 206,
                      zip : &121 "92443"}
      audited : &o46 "11/1/68",
      audited : &o46 "10/12/77",
      taxamount : &o283 0,
      taxevasion : &o632 "likely"}
   taxpayer : &o20
     {name : &o132 "Korolev",
      address : &o253 "Baikonur, Russia",
      audited : &o46 "10/12/86",
      taxamount : &o283 0,
      taxevasion : &o632 "likely"}
   company : &o26
      {name : &o623 "Rocket Propulsion Inc.",
       owner : &o24}
  }
```

Figure 2: Text representation of semistructured data

course, that choise is not unique, nor necessarily the best. Fig. 4 contains an alternative schema.

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. Our solution is to store that data in an *overflow* graph. Any semistructured data repository can be used to store the overflow graph: efficiency is no longer crucial, since one goal is to keep the overflow small. System issues arise from the integration of a relational storage with a semistructured overflow, but they are not addressed in this paper.

1.2 Related Work

Data clustering is the problem of grouping a large number of points in R^d into sets (clusters), where the distances between any two points in the same cluster is small. BIRCH [?] 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 [?] describe an algorithm which, when given a semistructured data, extracts a schema for

Taxpayer1

oid	name	street	no	apt	zip	audit1	audit2	taxamount	taxevation
o24	Gluschko	Tyuratam		2C	07099	10/12/63		12332	
o21	Kosberg	Tyuratam	206		92443	11/1/68	10/12/77	0	likely

Taxpayer2

oid	name	address	audited	taxamount	taxevasion
o20	Korolev	Baikonur	10/12/86	0	likely

Company

name	owner
Rocket Propulsion Inc.	o24

Figure 3: Relational storage

Taxpayer

oid	name	audit1	audit2	taxamount	taxevasion
o24	Gluschko	10/12/63		12332	
o21	Kosberg	11/1/68	10/12/77	0	likely
o20	Korolev	10/12/86		0	likely

Address1

taxpayeroid	street	no.	apt	zip
o24	Tyuratam		2C	07099
o21	Tyuratam	206		92443

Address2

taxpayeroid	address
o20	Baikonur

Figure 4: Alternative relational storage (`Company` is the same).

that data. This is essentially a clustering algorithm.

Theodoratos and Sellis [?] describe a warehouse-configuration algorithm that given a relational database instance and a set of queries, generates an optimal set of views which best support the given set of queries. This is related to the storage generation problem, were we search for relational views over semistructured data. What makes our problem different is that its input is a data instance, rather than a set of queries. Also, our "views" need to be lossless.

Wang and Li [?] recently extended data mining techniques to semistructured data. Their algorithm finds "interesting patterns", i.e. data trees with high support. We found data mining to be a good starting point for the STORED generation algorithm. But our goals are different from those in data mining: we search for more complex patterns, and attempt to cover most of the data. If applied directly, Wang and Li's patterns would generate a rather simple relational storage which covers only a small fragment of the data.

Tsatalos, Solomon, and Ioannidis pioneered the idea of achieving physical data independence by means of relational views in GMAPs [?]. STORED follows a similar philosophy in achieving independence from the underlying relational storage.

As a language, STORED is closely related to query languages for semistructured data such as Lorel [?, ?], UnQL [?], MSL [?], and StruQL [?, ?], all of which support path expressions for matching attribute paths in semistructured data. Due to its unique requirements, STORED is strictly weaker than these languages. XML can be modeled as semistructured data [?, ?], and a few XML query languages have been proposed (XQL from Microsoft, and XML-QL [?]).

The only other technique developed for storing SGML (and, in particular XML) documents without storing the schema explicitly uses an object-oriented database [?]. The technique requires a DTD, and uses it to derive an object-oriented schema. This method is commonly believed as the most appropriate for storing XML data when the DTD is known, and has renewed interest in object-oriented databases. In certain applications this relience on a DTD is not possible however. Moreover, the method results in fragmentation, since new classes and objects must be introduced to convert a DTD into a class hierarchy [?, ?]. We consider our technique to be a complementary one, in two ways: first it does not rely fundamentally on a DTD, and second it uses a RDBMS as opposed to an ODBMS.

The following is an outline of the paper. Sec. 2 introduces the STORED language. Sec. 3 describes an algorithm for automatically generating the relational STORED mappings from a data instance, while 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. We conclude in Sec. 7.

2 STORED

We describe here STORED (Semistructured TO Relational Data), a declarative query language that maps the semistructured data into a relational schema and overflow graphs. A *relational schema* is a collection of relation names $R_1, \ldots, R_m$ with their arities $n_1, \ldots, n_m$. An *overflow schema* is a collection of graph names, $G_1, \ldots, G_k$. Conceptually, each graph G_i consists of a unary relation $G_i.roots(X)$ and a ternary one $G_i.edges(X, L, Y)$ (X, Y object identifiers, L attribute name). A *mixed schema* consists of both a relational schema and an overflow schema.

A STORED query 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, nor regular path expresions. This is to ensure losslessness: *any* semistructued data instance must be fully reconstructed from the relational transformation.

2.1 Simple Storage Mappings

A STORED query consists of one or several `FROM WHERE STORE` mappings. The `FROM` clause consists of a single pattern, binding several variables. The `STORE` clause states how values bound to variables are stored.

Basic unnesting. In the following example nested `addr` objects are flattened into the 4-ary relation `Taxpr1`:

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

Each `FROM` clause defines a unique *key* variable. By default, this is the first variable in the pattern, e.g., `$X` above. Each binding of the key variable that matches the pattern results in exactly one tuple being stored by the `STORE` clause.

Patterns are sequences of attribute constants; regular path expressions are not allowed. This insures that we can uniquely recover the semistructured data from the relational storage. Joins are also prohibited. Finally, all variables in the `FROM` clause must be used in the `STORE` clause, but note that intermediate variables may be removed from the pattern, i.e., it's not necessary to write `$A` in `addr:$A` {`street:$S, zip:$Z`}.

Optional attributes. We can store optionally other attributes:

```
M2 = FROM Audit.taxpayer : $X
             {name : $N, addr : {street $S, zip $Z},
              OPTIONAL{ 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 are always non-null, but `audited` and `taxamount` are optional. When one of them is missing, the last two attributes in `Taxpr2` are `null`. Or, we can check for `audited` and `taxamount` independently:

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

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

Clustering objects. A STORED query may contain several mappings. The following illustrates how to cluster `taxpayers` into two relations, based on combinations of their attributes:

```
M4a = FROM Audit.taxpayer : $X
             {name : $N, addr : $P,
              OPTIONAL{audited : $A},
              OPTIONAL{taxamount : $T}
             }
      WHERE typeOf($P, "string")
      STORE Taxpr4a($X, $N, $P, $A, $T)

M4b = FROM Audit.taxpayer : $X
             {name : $N,
              addr :  {street $S,
                        OPTIONAL{apartment $Ap},
                        OPTIONAL{city $C,
                        OPTIONAL{zip $Z}}}
              OPTIONAL{audited : $A},
              OPTIONAL{taxamount : $T}
             }
      STORE Taxpr4b($X, $N, $S, $Ap, $C, $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 `apartment` and `city` are optional. When `city` is present, then a `zip` is optional [2].

Each `STORE` clause must refer to a distinct relation name (`Taxpr4a` and `Taxpr4b` in the exmaple): this is necessary to allow us to reconstruct the semistructured data.

In the above example, it is obvious that for a given object, only one of M4a or M4b will succeed,

[2]To be precise: a `zip` without a `city` is not stored, while a `city` without a `zip` *is* stored.

because the conditions on addr are mutually exclusive. In general, however, several mappings 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. On the other hand it increases the disc space requirement.

Object splitting. STORED can express three kinds of object splitting: horizontal splitting, vertical splitting, and object-subobject splitting. The latter consists in separating a large subobject from the containing object and storing it separately.

2.2 Multiple Attributes

One benefit of semistructured data is that it permits multiple values for an attribute; for example, it is permissible for an object to have two `phone` attributes and several `subordinate` attributes. We distinguish two classes of multiply occurring attributes: *small-set* and *collection*. A small-set attribute usually has low cardinality and most often is one. For example, most people only have one `phone`, but some may have two. A collection attribute denotes a collection of objects, usually with high cardinality. For example, bosses usually have several to many `subordinates`. For small-set attributes, we may increase the number of columns in the relations to accomodate 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.

Small-set attributes. This example shows how to store the `audited` attribute, which occasionally occurs twice:

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

M7 only 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`. It suffices to store a single permutation in `Taxpr7`. Here, STORED's semantics differs from the traditional one. Conceptually, STORED queries are rewritten such that each occurrence of an attribute name `a` is uniquely renamed as `a[1], a[2], a[3], ...`:

```
M8 = FROM Audit.taxpayer : $X
            {name[1] : $N,
             audited[1] : $A1,
             OPTIONAL{audited[2] : $A2}}
     STORE Taxpr8(N, A1, A2)
```

Only attributes below the key variable are enumerated: `Audit` and `taxpayers` are not. Now we use the fact that our semistructured data model is ordered. For each data object `$X`, we conceptually perform the same enumeration of its attributes: its `audited` attributes become `audited[1], audited[2], audited[3], ....` This guarantees the matching between the pattern and the data is unique.

To simplify both the query rewriting and overflow mapping generation algorithms we impose the following two restrictions:

- All occurrences of the same attribute in the same subpattern must be followed by identical subpatterns. Thus {`address : {street $S1, zip $X}, address : {street $S2, zip $Y}`} is OK, but {`address : {street $S1, zip $Z}, address : {street $S2, city $C}`} is not.

- If the same attribute occurs twice in some subpattern, then their surrounding `OPTIONAL` subpattern are either identical or the subpattern of the latter attributes is contained in the subpattern of the former. Thus {`name:$N1, name:$N2`} and {`name:$N1, OPTIONAL{name:$N2}`} are OK, but {`OPTIONAL{name:$N1}, name:$N2`} and {`OPTIONAL{name:$N1}, OPTIONAL{name:$N2}`} are not.

Collection attributes. Collection attributes are stored as nested relations. For some relational database systems, this means storing them in a separate relation. In this example, we assume that each `irscenter` has many `hearing` attributes (defining a subcollection):

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

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

In essence, we store a many-to-one mapping from `Hearings` to `IrsCenter`. `M11b` has the key variable `$Y`, which is not the first variable, and thus must be declared explicitly. STORED requires that all other variables in the pattern tree are either below or above the `KEY` variable.

Some RDBMS allow relations to be nested. In such a system, we may store `Hearings` as an inner relation, rather than a separate one. In STORED, this can be done with a *nested query*: [[[we could delete this example. dan]]]

```
M12 = FROM Audit.irscenter : $X
            {centername : $N,
             centeraddress : $A}
      STORE IrsCenter($X, $N, $A,
               FROM $X.hearing : $Y
                   { hearingdate : $D,
                     taxpayername : $TN,
                     auditorname  : $AN,
                     decision : $D}
               STORE Hearings($D, $IN, $AN, $Z))
                     )
```

For every binding of $X, the outer STORE clause creates a 4-ary tuple whose fourth attribute is the nested relation created by the inner FROM-STORE clause. Note that the first pattern in nested query uses the binding of $X.

2.3 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:5551235},
       joe: {phone:5552345,fax:5552346},
       sue: {phone:5552525,fax:5556666}, ...}
```

STORED has *label variables* which are stored in relations as values. Label variables are required to occur before the key variable.

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

Since the data is a tree and $L appears before $X, there exists a unique binding of $L for each binding of $X.

2.4 Overflow Graphs

So far all STORED mappings targeted the relational storage. We describe here mappings into the overflow storage.

```
M13a = FROM Audit.taxpayer : $X
             {name : $N,
              OPTIONAL{address : {street $S, city $C}}}
       STORE R($X,$N,$S,$C)
```

Any attributes in the semistructured-data source that are not matched by this query, e.g., a phone or address.zip attribute, must be stored in the overflow graph.

Overflow STORED have a different syntax from relational mappings. The following two overflow mappings construct two graphs G1 and G2, accompany the mapping M13a:

```
M13b = FROM Audit.taxpayer : $X
              {name : $N,
               OPTIONAL{address : $A},
                        $L : _}
       OVERFLOW G1($L)
```

<==

```
M13c = FROM Audit.taxpayer : $X
              {name : $N,
               address : {street $S, city $C, $K : _}}
       OVERFLOW G2($K)
```

The patterns for overflow mappings have one attribute variable, which must occur last in the pattern (nested or not): other than that, they are like patterns in the relational mappings. Syntactically, the OVERFLOW statement always "stores" that attribute in the overflow graph.

The exact meaning is as follows. Consider M13b, and let $X be bound to some object identifier x. After matching name and, possibly address, $L is bound successively to all remaining attributes. For each binding of $L to some attribute name l with value y:

1. x is stored in G.roots,
2. the edge (x,l,y) is stored in G.edges, and
3. the entire subtree rooted at y is stored in G.edges.

This allows us to reconstruct the data from R1 and G1. Note that if some object $X has several name or address attributes, the first one is stored in R, while all others are stored in G1.

The key variable is always stored in G.roots, even if it is not the actual root of the subgraph being stored. For example the following are stored by M13c: G2.roots($X) and G2.edges($X, $K, _) (where _ is the actual value of the $K attribute). This is just a convenience: otherwise we would be forced to store the OID of address in the relation R.

Consider another example, which stores all objects different from taxpayers and company in G4:

```
M14 = FROM Audit.$L : _
      WHERE $L not in {taxpayer, company}
      OVERFLOW G3($L)
```

Overflow mappings cannot have nested patterns other than those containing the attribute variable. This condition ensures that overflow mappings are monotone and is required in the rewriting of updates (Sec. 5). For example, the following query is not permitted:

```
FROM Audit.taxpayer : $X
       { name: {firstname: $FN, lastname: $LN},
         $L : _}
OVERFLOW G($L)
```

Mixing relational and semistructured storage In addition to a relational system, we need a second system to support overflow graphs. Minimally, such a system must provide an object repository and a query engine and support updates: overflow graphs could be simply stored in text files. Integrating the two systems is necessary to preserve the flexibility of the original semistructured data. What is important is that the performance requirements for the overflow system are far less stringent than for a stand-alone semistructured data management system, because the relational system handles most of the processing.

3 Generating Storage Mappings

In this section, we describe how to generate automatically a STORED query `M` given a semistructured data instance `D`.

Several competing goals determine how "good" or effective a storm query is. First, we want to limit the number of tables: although many commercial RDBMS do not limit the maximum number of tables, the solution of storing each object in a separate table is undesirable. Second, we want to bound the disk space. Although size of the data instance `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 at all. Nonetheless, we do not want to generate very wide, sparse tables, because, one byte per null entry can become expensive. A related restriction is that some RDBMS impose an upper limit on the number of attributes per table[3]. Depending on the application, other goals may include reducing data fragmentation (i.e., avoiding horizontal and vertical splits), reducing redundant storage of objects in multiple relations, or increasing object redundancy to improve query evaluation.

Given these goals, we could model storage generation as a cost-optimization problem. Given a data instance `D`, generate a STORED query `M` that minimizes a *storage-cost function* `c(M)`, the cost of storing `M(D)`. In addition, the generator must accommodate hard constraints, e.g., the limit of attributes per table.

An additional complexity is the presence of a query mix, $\bar{\mathtt{Q}} = \{\mathtt{Q}_1, \ldots, \mathtt{Q}_k\}$, on the semistructured data instance `D`. For a given STORED query `M`, each query $\mathtt{Q_i}$ can be rewritten into a relational query, $\mathtt{Q_i^M}$, on `M(D)` (Sec. 5). Given a weight $\mathtt{f_i}$ for each query $\mathtt{Q_i}$ (the frequency or its relative importance), and a query cost function $\mathtt{d}(\mathtt{Q_i^M})$ that denotes the cost of evaluating $\mathtt{Q_i^M}$ over the relational dat a, a second goal is to minimize the *query-cost* function, $d(M) = \sum_{i=1,k} f_i d(Q_i^M)$ The optimization problem can now be applied to the *combined cost* $\mathtt{c(M)} + \mathtt{d(M)}$.

The storage-cost cost optimization problem is NP-hard in the size of the input data, even in simple contexts, such as no nested objects and when `c(M)` only counts the total number of nulls. We prove this by reduction from the *rectilinear picture compression* problem [?], which is NP-complete.

Theorem 3.1 *The problem of computing an optimal storage mapping* `M` *is NP-hard in the size of the semistructured data* `D`*, even when the storage-cost function is the number of nulls.*

We emphasize that this is a daunting complexity: query optimization problems are typically NP-complete in the size of a *query*, while here the NP-hardness is in the size of the *data*. Search algorithms like dynamic programming, are unlikely to work. We are forced to look at heuristics. We start with data mining techniques, and extend them to account for the query mix and to generate complex storage mappings. A data-mining algorithm for semistructured data has been recently described by Wang and Li [?]: we review it here briefly, and refer to it as *WL's algorithm.*

3.1 Review of the WL Semistructured Data Mining Algorithm

WL's semistructured-data model is a large collection of semistructured objects, i.e., it is 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[1], 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 [?] with items 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

[3] Oracle 8.0.4 imposes a limit of 1000.

Parameter Name	Meaning
K	Max number of tables
A	Max number of attributes per table
S	Max disk space
C	Collection size threshold
Supp	Min Support

Table 1: Parameters for the storage generation algorithm

sets of k paths, whose associated pattern trees have high support. A number of optimization techniques are described in [?] to reduce the size of the sets F_k.

3.2 Storage Generation Algorithm

Our storage generation algorithm has five parameters, listed in Table 1. It generates a relational storage with at most `K` tables, each having at most `A` attributes, and with total disk space at most `S`. We assume fixed length records. This results in an approximation of the disk space: more accurate measures are possible (e.g., to account for when nulls take far less space). `C` is a parameter distinguishing between "small collections" and "large collections" An attribute with less than `C` occurrences is considered a small collection, and the algorithm attempts to produce one column for each. Attributes with `C` or more occurrences are represented by nested relations. Finally, `Supp` is the minimum support, a parameter for the data-mining algorithm.

Storage patterns We define a type of pattern, called *storage patterns*, that are different from WL's patterns. A storage pattern has the form `F:B`, where `F` is the prefix and `B` the body. The prefix is a word $\mathtt{l}_1.\mathtt{l}_2\ldots\mathtt{l}_k$, where the labels $\mathtt{l}_1,\ldots,\mathtt{l}_k$ have one of the following forms:

- `a`: an attribute name
- `-`: a wildcard (similar to WL's "?")

The last label, $\mathtt{l}_k$, must be an attribute name. The body `B` has the form $\{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_p : \mathtt{B}_p\}$, where $\mathtt{B}_1, \ldots, \mathtt{B}_p$ are other bodies, and each label is of one of the following forms:

- `a[i]`: indexed attribute
- `a[*]`: collection attribute

An example of a pattern is:

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

Intuitively, this pattern is contained in all `taxpayer` objects with at least a `name`, two `phones`, and `C` `addresses`, the latter with streets and cities. Note that `phone[1]` is missing: only the highest index occurs in a storage pattern. These patterns have an obvious connection to STORED queries, which is formalized below. `a[i]` denotes `i` occurrences of the attribute name `i`, and `a[*]` denotes a nested collection.

Pattern Support Given a pattern `P = F:B` and a semistructured data instance `D`, the support of `P` is defined as follows. Let $\mathtt{o}_1, \ldots, \mathtt{o}_n$ be all objects in `D` reachable from the root by a path matching the prefix `F`. The support is defined to be the number of objects $\mathtt{o}_i$ which *contain* the body `B`.

The definition of containment is as follows. First, replace every occurrence of a label `a[*]` in `B` with `a[C]`. Assuming $\mathtt{B} = \{\mathtt{l}_1 : \mathtt{B}_1, \ldots, \mathtt{l}_p : \mathtt{B}_p\}$, an object `o` contains `B` iff for each label $\mathtt{l}_j$ of the form `a[i]` the object has at least `i` outgoing edges labeled `a`, to objects $\mathtt{o}_1, \ldots, \mathtt{o}_i$ and each $\mathtt{o}_i$ contains the pattern $\mathtt{B}_j$.

Modeling Query Support Queries on semistructured data have one or more data patterns, that specify paths to match in the input data, and conditions on the variables bound by the patterns (examples of queries are in Sec. 5). Our technique represents a query's patterns as data, and applies the data mining algorithms to this representation. The representation ignores any conditions other than patterns, and each pattern in a query is represented individually. Equivalently, we may assume that each query contains one pattern and no other conditions. Hence, a query is a tree, with edges labeled with attribute constants, attribute variables, or regular path expressions. Given the weight `f` of the query `Q`, we assume the data that "corresponds" to the pattern occurs `f` times. We make this precise next.

Given a storage pattern `P` and a query pattern `Q`, we define when `P` *occurs* in `Q`. To do this, we strip all indexes in `P`, i.e., conceptually replace `a[i]` with `a` and `a[*]` with `a`. Formally, `P` occurs in `Q` if and only if there exists a mapping from the nodes of `P` to those in `Q`. The mapping must map `P`'s root to `Q`'s root, and if some node `x` is mapped to a node `y`, and `y'` is `y`'s parent, then there exists an ancestor `x'` of `x` that is mapped to `y'`; in addition, the path from `x'` to `x` is either labeled with a word that belongs to the regular expression labeling the edge `(y', y)` or, when `x` is a leaf, then the word is a prefix of a word in that regular expression.

Given a query mix $\mathtt{Q}_1, \ldots, \mathtt{Q}_k$ with weights $\mathtt{f}_1, \ldots, \mathtt{f}_k$,

```
ALGORITHM: Automatic Storage Generation
INPUT:     K, A, S, C, Supp,  and query mix Q
OUTPUT:    Set of relational STORED mappings

METHOD:
   Step 1: Find all minimal prefixes
                   with data support >= Supp

   Step 2: - Run the WL data mining algorithm
                 with the changes in the text.

           - Let K' = number of maximally
                  contained  patterns found

   Step 3: Select K0 (<= K) patterns out of the K'


   Step 4: For each of the K0 patterns, select
            the set of ``required'' attributes

   Step 5: For each of the K0 patterns with
           required attributes, generate one or
           more STORED relational mappings.
```

Figure 5: Automatic Storage Generation Algorithm

we define the *query support* of a storage pattern P to be the sum of all $\mathtt{f_i}$ for which P is contained in $\mathtt{Q_i}$. The *mixed support* of P is the sum of the data support and the query support.

The algorithm Our algorithm is shown in Fig. 5. We explain each step next.

Step 1: Compute minimal path prefixes. First, we generate all prefixes $\mathtt{l_1.l_2 \ldots l_k}$ with support $\geq$ Supp. This is not a data-mining problem, because the support may increase as the prefix length increases. 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 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. Third, our patterns are simplified since, instead of keeping i distinct copies of the same attribute a[1], a[2], ..., a[i] we only keep one a[i]. This reduces both the number of patterns[4] and the amount of space for each pattern. Finally, we terminate the algorithm either when the combined support decreases below Supp or when k reaches A.

Step 3: Select K patterns. From Step 2, we have K' maximally contained patterns, each with support $\geq$ Supp. Recall that WL's algorithm also retains F_1, the set of highly supported paths. Now we use a greedy 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 $\mathtt{P_1}$ that covers (i.e., contains) paths in F_1 with highest support. In general, we pick pattern $\mathtt{P_k}$ such as to (a) minimize the maximum overlap with $\mathtt{P_1}, \ldots, \mathtt{P_{k-1}}$, 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 K patterns P selected in Step 3, we have a chain of backpointers to strictly smaller subpatterns, $\mathtt{P} = \mathtt{R}^1, \mathtt{R}^2, \ldots, \mathtt{R}^n$, with increasing data support (with $\mathtt{R}^n$ in F_1). In step 4 we choose some i for each pattern so that the subpattern $\mathtt{R}^\mathtt{i}$ 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 stored and therefore increase the total disk space. For a given i, the disk space requirement for P can be computed from the data

[4]For technical reasons, WL's algorithm needs to consider patterns of the form a[1], a[2], ..., a[i-1], a[i+1] too.

supports: for example, when P has no collections (i.e., no a[*] labels), then the disk space storage is the number of leaves in P times the data support of P_i. 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 query. Each storage pattern P, with its required subpattern R, is converted into one or more STORED mappings. The required subpattern R is mapped into the required part of the STORED mapping and all other attributes are mapped into the optional part. Collection labels in P (i.e., of the form a[*]) are treated as collections with a separate STORED query. We omit some details for lack of space. To illustrate, consider the following pattern/subpattern:

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

The associated STORED mappings are:

```
M1 = FROM Audit.taxpayer:$X
         { name:$N, phone:$P1,
           OPTIONAL{ phone:$P2} }
     STORE R1($X, $N, $P1, $P2)
M2 = FROM  Audit.taxpayer:$X.address:$Y
             { street $S, city $C }
     KEY $Y
     STORE R2($X, $Y, $S, $C)
```

4 Generating Overflow Mappings

In Sec. 3, we showed how to generate automatically relational mappings for a given data set and query mix. Next we address the *overflow-generation* problem: given relational mappings, generate explicit overflow mappings such that the combined STORED query is lossless. A related problem is *losslessness checking*: given a STORED query with relational and overflow mappings, decide whether it is lossless.

Without any knowledge of the data's structure, overflow mappings are always needed. Our goal is for the overflow and relational storages to have minimal overlap; typically this requires specifying many, complex mappings. In practice, we have found that specifying overflow mappings 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. We address both problems given a *semistructured schema*, S.

SS Schemas. Semistructured schemas are specified in a syntax similar to semistructured data, but we replace attribute lists by regular expressions and scalar constants by atomic types. For example, the schema:

```
S1 = SCHEMA {Audit :
         {(taxpayer: {name: String,
                      address: String,
                      (address: String)?,
                      (taxreturn: S2)*}
          )*}}
S2 = SCHEMA {year: String, amount: String,
            (extension:String)?}
```

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.

The regular-expression operators comma (concatenation), (_) | (_) (alternation), (_)? (optional), and (_)* (Kleene star) are recognized. All regular expressions are *unordered*, e.g., example above does not imply that name must appear before address. Unordered regular expressions are studied in [?]. Unordered, as opposed to ordered, databases is a departure from the XML framework, but is more in the spirit of database applications. For example, in our setting (name:String | address: String)* is equivalent to (name:String)*, (address:String)*.

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

```
S3 = SCHEMA {Audit :
      {(taxpayer : {name : String,
                    address : {street : string,
                               zip : int,
                               ($K : Any)*},
                   ($L : Any)*} )*}}
     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. It is similar to #PCDATA in DTD's. Label variables increase the flexibility of semistructured schemas, because they allow us to define Any.

Semistructured data exists independent of schema, therefore, a database instance can conform to multiple schemas. In principle, an XML source may conform to several DTD's, although a validating parser would only check the data against the DTD named in the source. Schemas form a partial order: $\mathtt{S} \subseteq \mathtt{S}'$, if any database conforming to S also conforms to S'. Note that Any is a top element in this partial order: $\mathtt{S} \subseteq \mathtt{Any}$, for all schemas S.

Arbitrary Schemas. Checking losslessness for a given query and schema is decidable, by reduction to the *schema-containment* problem in [?]. For a given STORED query Q, we construct a (very complex) schema $\mathtt{S_Q}$ describing precisely the instances on which Q is lossless. Then, given a schema S, it suffices to check whether $\mathtt{S_Q} \subseteq \mathtt{S}$. Beeri and Milo [?] have shown that this problem is decidable in double exponential space, which makes the solution impractical. Moreover, it is not clear how to use that result for the inference problem.

Restricted Schemas. The high complexity is due to the expressive power of the schemas and DTD's. For example, the schema [5] S4 = SCHEMA {name, (name, name, address, address, address)*} specifies that there are an odd number of names and the number of addresses is divisible by 3, and that these two numbers are of the form $2n+1$ and $3n$, for some $n \geq 0$. Such complex schemas are rarely needed in practice. Regular expressions in DTD's are typically used to express optional parts, required parts, and repetitions. We restrict our discussion to a simpler form of 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 $\mathtt{S_r}$ by performing the following transformations repeatedly (here p, p',... denote pairs (a:S), (a':S'), ...):

```
(p | p') -> p?, p'?
(p,p') ? -> p?, p'?
(p*) ?   -> p*
(p?) ?   -> p?
(p,p')*  -> p*,p'*
(p*)*    ->  p*
(p?)*    -> p*
```

One can easily verify that $\mathtt{S} \subseteq \mathtt{S_r}$.

For restricted schemas, we use a notation that indicates the range of each attribute's cardinality. In this notation, schema S1 becomes:

```
S1 = Audit[1] :
    { taxpayer[0,*] :
        { name[1],
          address[1,2],
            taxreturn[0,*] :
              {year[1], amount[1],
               extension[0,1]}}}
```

[5]We drop the trailing types, i.e. write name instead of name:String.

* denotes ∞. Each attribute a occurring in S has a range, denoted range(a), which is an interval [i,j], with i a number and j a number or *. When i=j we abbreviate range(a) by [i].

We describe the algorithm for overflow generation given a STORED query Q with relational mappings only and a restricted schema S. To simplify the presentation we assume that S is non-recursive and has no variables and that each attribute name occurs only once in the representation of S (as in S1 above). We address these restrictions later.

We start by indexing all attribute names in Q, as in Sec. 2.2. Recall that attribute names that occur above the key variable are not indexed, but in this section, we index them with *. The purpose of this minor change is explained below. For each attribute name a in Q, define high(a) to be its maximum index: when a has * as its only index, then high(a) = 0. We illustrate our algorithm on the following two STORED mappings:

```
M1 = FROM Audit[*].taxpayer[*]: $X
       {name[1]:$N, address[1]:$A,
        OPTIONAL taxreturn[1]:
            {year[1]: $Y, amount[1]: $A,
             extension[1]: $E}}
     STORE R($X, $N, $A, $Y, $A, $E)

M2 = FROM Audit[*].taxpayer[*]:$X.taxreturn[*]:$Z
             {year[1]: $Y, amount[1]: $A}
     KEY $Y
     STORE Q($X, $Z, $Y, $A)
```

We have:

```
high(Audit)=0, high(taxpayer)=0,
     high(name)=1, high(address)=1,
high(taxreturn)=1, high(year)=1,
     high(amount)=1, high(extension)=1
```

For each attribute name a that occurs in the schema, the algorithm checks whether a can be recovered without loss from the relational mappings in Q. To do this, we construct from S several canonical semistructured instances, denoted $\mathtt{D_a}$, and check whether Q stores all occurrences of a in these instances. Since S is non-recursive, its standard representation is a tree. $\mathtt{D_a}$ is derived from this tree by creating for each attribute name b several copies, labeled b[1], b[2], ..., b[k], and possibly b[*], as described next.

- When b is not a, and is not an ancestor of a in S, then we create i copies lableled b[1], b[2], ..., b[i], where i is the low value of range(b) (i.e. range(b) = [i,j]).

- When `b = a`, or `b` is an ancestor of `a` in the tree `S`, then we choose a certain `k` and create copies labeled `b[1], b[2], ..., b[k]`. Each value of `k` results in a different database $\mathtt{D_a}$, and the algorithm creates all of them. Namely let `range(b) = [i,j]` and `high(b) = m`. When `j=*` then `k` is choosen to be each of `i, i+1, ..., m, *`. When `j` is a number, then `k` is choosen to be each of `i, i+1, ..., min(j, m+1)`.

Fig. 6 depicts the construction of $\mathtt{D_a}$. Only instances corresponding to leaf attributes are shown. The other three: ($\mathtt{D_{Audit}}, \mathtt{D_{taxpayer}}, \mathtt{D_{taxreturn}}$), are similar. $\mathtt{D_{name}}$ has as few instances of each attribute as possible. For $\mathtt{D_{address}}$, we create two instances, because `range(address) = [1,2]`. Both $\mathtt{D_{year}}$ and $\mathtt{D_{amount}}$ have two instances (one for $\mathtt{D_{taxreturn[1]}}$ and one for $\mathtt{D_{taxreturn[*]}}$). Only the larger instance is shown; the the other has only the `taxreturn[1]` attribute, but not `taxreturn[*]`. The construction of $\mathtt{D_{extension}}$: is similar.

In the last step, the algorithm "evaluates" `Q` on each canonical database $\mathtt{D_a}$. The evaluation semantics is modified from the standard definition as follows:

1. An attribute `b[i]` in the query, with `i` a number, matches only the attribute `b[i]` in the database (standard semantics).

2. An attribute `b[*]` in the query matches any attribute `b[j]`, for `j` either a number or `*` (new semantics).

If any of the edges in $\mathtt{D_a}$ are unmatched, then the query is lossy, and we need to generate overflow mappings: one for each unmatched edge. In our example, there are two unmatched edges: `address[2]` in $\mathtt{D_{address}}$ and `extension[1]` in $\mathtt{D_{extension}}$. The corresponding overflow mappings are generated directly:

```
O1 = FROM Audit.taxpayer:$X {
               name:$N, address:$A,
               $L:_
     WHERE $L = address
     OVERFLOW G1($L)

O2 = FROM Audit.taxpayer:$X {
               name:$N, address:$A, taxreturn:$T,
               taxreturn:{year:$Y, amount:$A, $L:_}}
     WHERE $L = extension
     OVERFLOW G2($L)
```

The overflow-generation algorithm can be adapted easily to check losslessness of a query `Q` with both relational and overflow mappings. We proceed as above, but evaluate both relational and overflow mappings on the databases $\mathtt{D_a}$. The details are omitted.

We address now the restrictions in our algorithm:

Recursive Types We unfold recursive types. If `d` is the maximum nesting depth in the query `Q` (recall that `Q` does not have regular path expressions), then it suffices to unfold `S` up to depth `d+1`. When we generate the canonical databases, we stop at that depth, because anything below is unreachable by `Q`. In practice, we may unfold `S` and construct canonical databases on-the-fly, as we evaluate the queries: this may save us from unfolding portions of `S` that the queries do not reach anyway.

Label variables Each label variable `$L` will occur as constants in the canonical databases $\mathtt{D_a}$; in addition we have now databases $\mathtt{D_{\$L}}$. Conceptually, `$L` can be one of the attribute constants occurring in the query and schema, or anything else. This has to be taken into account both during the evaluation of `Q` and when checking whether an edge is covered.

Repeated attribute names We rename attribute names occurring in different places in the schema. For that, it suffices to replace `a` with the concatenation of all attribute names on the path from the root to `a`. The same has to be done in the query.

Finally, we discuss the complexity of our algorithm. In the worst case, its running time is exponential. This happens when there exists a chain of nested attributes with large ranges in `S` leading to some attribute `a`; then there are exponentially many canonical databases $\mathtt{D_a}$. The following modification ensures that the algorithm will run in polynomial time, at the expense of less focused overflow mappings. Namely in item (2) above we consider multiple databases only when `b = a`. When `b` is `a`'s ancestor it suffices to pick a single database (with attribute `b[*]`, when `range(b) = [i,*]`, or with attributes `b[1], ..., b[j]`, when `range(b) = [i,j]`). For example, in Fig. 6, the databases $\mathtt{D_{year}}, \mathtt{D_{account}}$ and $\mathtt{D_{extension}}$ will have only the `taxreturn[*]` attribute. Now the overflow mapping `Q2` becomes:

```
O2 = FROM Audit.taxpayer:$X {
               name:$N, address:$A,
               taxreturn:{year:$Y, amount:$A, $L:_}}
     WHERE $L = extension
     OVERFLOW G2($L)
```

Theorem 4.1 *The above algorithm correctly computes the overflow mappings for a STORED query* `Q` *and a restricted semistructured schema* `S`*. The algorithm produces overflow mappings only when*

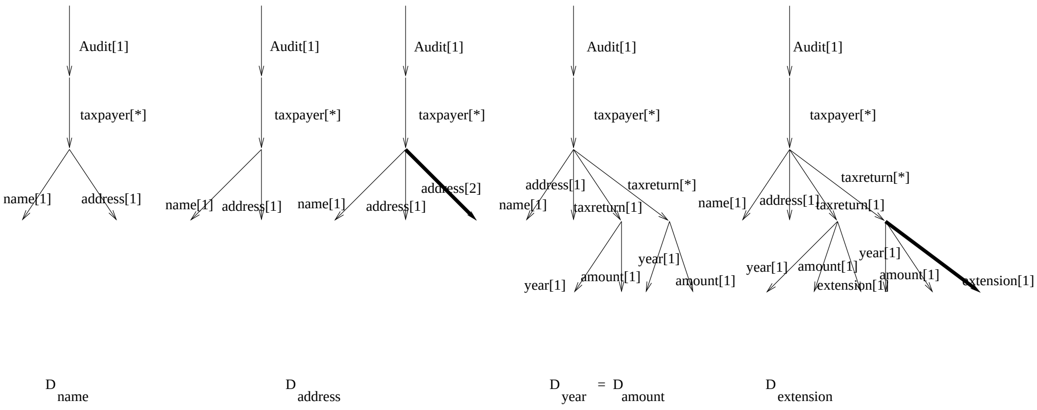

Figure 6: Canonical Databases for Schema S1

needed. The algorithm can also be applied to an arbitrary schema S*, by converting* S *first to a restricted schema.*

5 Query and Update Rewriting

Given a STORED query, 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. Query rewriting is related to *query rewriting using views* [?, ?]; update rewriting seems specific to our setting.

Reconstructing the Semistructured Data We start by describing the technique by which a semistructured-data instance can reconstructed from its relational and overflow storage. This technique underlies query and update rewriting: in practice, a system would never physically reconstruct the semistructured data.

Reconstruction is based on *inversion rules*, a concept introduced by Duschka and Genesereth [?] as a tool for rewriting relational queries. In their framework, inversion rules are like datalog rules with Skolem functions in the rule's heads. In our setting, the head of an inversion rule is a *tree constructor* which is similar to the tree constructors found in MSL [?] and XML-QL [?].

Inversion rules are generated internally by the system, and not intended for the user. We describe them in a concrete syntax for presentation purposes only. Given a STORED query, inversion rules are created as follows: for each relational mapping, one inversion rule is created for the required subpattern, and one inversion rule for each `OPTIONAL` subpattern; for each overflow mapping, a single inversion rule is created for its required subpattern (and `OPTIONAL` subpatterns are ignored).

We illustrate on the STORED example in Fig. 7, for which we already enumerated the attributes (Sec. 2.2): it has four inversion rules, shown in

```
Ma  = FROM Audit.taxpayer: $X
       { name[1] : {firstname[1] : $FN,
                    lastname[1] : $LN},
         addr[1] : {street[1] : $S,
                    city[1] : $C},
         OPTIONAL{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] : $N,
              OPTIONAL{taxamount[1] : $T},
              $L : _}
      OVERFLOW G2($L)
```

Figure 7: Example of STORED query

Fig. 8, which are constructed in a straightforward way. The Skolem functions' names, e.g., `S_Audit()`, are generated from the attribute names, including the index. Some care is needed when the same attribute name occurs under different paths: we must choose distinct Skolem function names for the different occurrences.

Query Rewriting We consider queries on semistructured data featuring tree pattern matching and regular path expressions. These are now common features in all query languages for semistructured data [?, ?, ?, ?]. As our running example we consider the query:

```
Q = SELECT $N
    FROM Audit.taxpayer:$X
          { name : $N, taxamount:$T,
            w4form.income:$I, address.*:$A }
    WHERE $T < 0.1 * $I, $A = "Philadelphia"
```

which we rewrite using the STORED mapping

```
Ia0 = FROM Taxpayer($X, $FN, $LN, $S, $C, $T)
      CONSTRUCT Audit : S_Audit()
        {taxpayer: S_taxpayer($X)
          { name : S_name_1($X)
               {firstname : S_firstname_1($X), $FN,
                lastname : S_lastname_1($X) $LN},
            addr : S_addr_1($X)
               {street : S_street_1($X) $S,
                city :  S_city_1($X) $C}
          }
        }

Ia1 = FROM Taxpayer($X, $FN, $LN, $S, $C, $T)
      WHERE $T != null
      CONSTRUCT Audit : S_Audit()
        {taxpayer: S_taxpayer($X)
              { taxamount : S_taxamount_1($X) $T
              }
        }

Ib  = FROM  G1.roots($X), G1.edges($X,$L,$Y)
      CONSTRUCT Audit : S_Audit()
        {taxpayer: S_taxpayer($X)
           {addr : S_addr_1($X) {$L : $Y}}}

Ic  = FROM G2.roots($X), G2.edges($X, $K, $Z)
      WHERE $K != taxamount
      CONSTRUCT Audit : S_Audit()
         {taxpayer: S_taxpayer($X) {$K : $Z}}
```

Figure 8: Inversion rules for query in Fig. 7

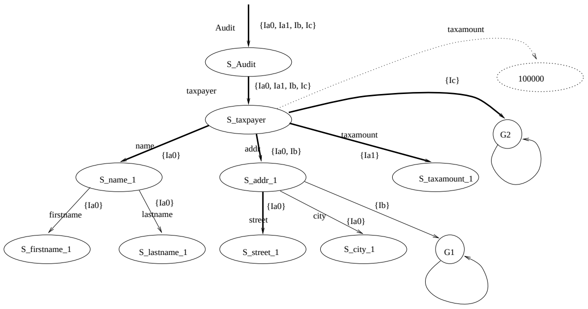

Figure 9: Canonical Object for STORED query in Fig. 7. The dotted edge is an extension, used for updates.

in `M` in Fig. 7. The pattern is in the `FROM` clause. The pattern's syntax is like that of semistructured data (Fig. 2) with two changes: attribute names are replaced with regular path expressions (e.g `address.*`), and oid's are replaced with variables. In general, more than one pattern can be specified. The `WHERE` clause specifies conditions on the variables. In our example, `Q` returns the names of taxpayers whose `taxamount` is less than `10%` of their income on some W4 form, and having `"Philadelphia"` somewhere under `address`. Note that the query returns a set of `name` oids, each with `firstname` and `lastname`.

Given a query `Q` on the semistructured data, and a STORED mapping `M`, we rewrite `Q` into a mixed query on both the relational store and the overflow graphs. We assume all inversion rules `I` to have been precomputed. To rewrite `Q`, it suffices to compose `Q` with the inversion rules `I`. We don't want to physically compute the intermediate result, which is the entire semistructured data instance, so we first perform algebraic simplifications on the composed query. This process is related to *query decomposition and algebraic optimization* [?] for the Mediator Specification Language (MSL). We describe it next.

We use the inversion rules to precompute a canonical, symbolic data object `O`. The object is created by "fusing" all symbolic objects in all inversion rules (those in the `CONSTRUCT` clause). Its nodes are labeled with Skolem function names or overflow graph names, and its edges are labeled with attribute names or are unlabeled, if the target node represents an overflow graph. Each edge is also annotated with the names of the inversion rules that generated that edge.

Query rewriting has two steps:

1. Evaluate `Q` on the symbolic object `O`.

2. For each result, compute all "covers" of the corresponding subobject in `O`.

We illustrate both steps on the example in Fig. 7 and Fig. 8. The canonical object is depicted in Fig. 9. In this example `name`, `taxamount`, and `address` are attributes mentioned in the STORED query, but `w4form` and `income` are not. Obviously, they must be retrieved from some overflow graph.

In step 1, we "evaluate" `Q` on `O`. This process is similar to evaluating `Q` on any semistructured-data source, with two exceptions: the conditions in the `WHERE` clause are not checked, and unlabeled edges in `O` match any attribute in the query. The result of the evaluation is summarized in the relation:

$N	$T	$I	$A
S_name_1	S_taxamount_1	G2	S_street_1
S_name_1	S_taxamount_1	G2	S_city_1
S_name_1	S_taxamount_1	G2	G1
S_name_1	G2	G2	S_street_1
S_name_1	G2	G2	S_city_1
S_name_1	G2	G2	G1
G2	S_taxamount_1	G2	S_street_1
G2	S_taxamount_1	G2	S_city_1
G2	S_taxamount_1	G2	G1
G2	G2	G2	S_street_1
G2	G2	G2	S_city_1
G2	G2	G2	G1

`$X` is bound to `S.taxpayer` in each row and is not shown. Consider the first row. It corresponds to a subgraph of `O`, whose edges are highlighted in Fig. 9.

In step 2, for each subgraph defined by a row in the result relation, we compute all minimal subsets of inversion rules that cover the subgraph edges. From each cover, we construct a corresponding mixed query and take the union of all such queries. The process is straightforward, but omitted due to limited space.

In general, there may be exponentially many covers of a subgraph, but in practice often there are far less. For example, for the subgraph in Fig. 9, there exists a single minimal cover, { `Ia0`, `Ia1`, `Ic`}, which results in the query:

```
SELECT S_name($X)
  { firstname : S_firstname($X) $FN, // reconstruct
    lastname : S_lastname($X) $LN}   // name

FROM Taxpayer($X, $FN, $LN, $S, $C, $T),
           // for Ia0 and Ia1
     G2.roots($X'), $X'.w4form.income $I'
           // for Ic

WHERE $T != null,  // from Ia1
      $X = $X'     // glue Ia1 with Ic at S_taxpayer
```

The join condition `$X = $X'` is generated by "gluing" `Ia1` with `Ic` at the `S-taxpayer` node. Note that the query reconstructs the `name` OIDs using the Skolem functions `S_name`.

In the mixed query, the `FROM` clause contains both relational goals (like `Taxpayer(...)`) and semistructured data patterns on the overflow graphs. The `WHERE` clause may have join conditions, like `$X = $X'`.

The other eleven rows in the table are treated similarly, and result in additional queries. The complete rewritten query is their union. An interesting observation is that for certain data sources not all rows in the result table are meaningful. For example, when each `taxpayer` has at most one `name`, then no `name` attributes will ever be stored in `G2`. Therefore, we never bind `$N` to `G2`, and can discard half of the rows in the result relation. Other rows can be discarded if, for example, each taxpayer has at most one `taxamount`, etc. In general, one could use a semistructured schema to optimize the query rewriting, but this goes beyond the scope of this paper.

Summarizing:

Theorem 5.1 *Assuming that the STORED mapping is lossless, the rewriting algorithm translates any query on semistructured data into an equivalent one over the relational store and overflow graphs.*

Updates We consider two update statements:

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

The first statement inserts a new subgraph in a complex object, the second updates the atomic value of an atomic object. In both, `<object-selector>` is a query whose result must be a single object identifier `o`. Here `<object-expression>` is a constant expression denoting a complex-value object `o'` (syntax like in Fig. 2). The meaning of the first statement is that all `(attribute, value)` pairs of `o'` are added to `o`: the order in both sets is preserved, and the new pairs are added at the end of `o`'s value. The meaning of the second statement is that the value of the atomic object `o` is replaced with `<value-expression>`. This simple update language is a strict subset of the Lorel's [?].

Given a STORED query `M`, update rewriting consists of two steps. The first evaluates the `<object-selector>` on the canonical object `O`. For each result, in the second step we extend the canonical object, then evaluate each mapping in `M` on the extended object. We illustrate insertions only: updates are easier, and omitted. Our running example is:

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

together with the STORED mapping in Fig. 7. The first step evaluates the query on the object in Fig. 9. There are two bindings: ($X = S_taxpayer, $N = S_lastname), and ($X = S_taxpayer, $N = G2). We illustrate step two only on the first binding. In step two, we construct an *extended canonical* object, by gluing O with the object being inserted: in this case it is taxamount: "10000". The resulting object is illustrated in Fig. 9 by the dottent line. Now we execute the STORED query M on the extended object, but only consider those bindings which use at lease some edge of the extension. In our example M has three mappings: we illustrate for Ma and Mc. One of the bindings of Ma's variables is:

```
$X / S_taxpayer, $FN / S_firstname,
$LN / S_lasname_1, $S / S_street_1, $C, S.city_1
```

This corresponds to the following update:

```
UPDATE Taxpayer($X, $FN, $LN, $S, $C, $T)
SET    $T := 100000
WHERE  $LN="Smith"
```

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

```
UPDATE G2.roots($X), G2.edges($X,taxamount,100000)
WHERE Taxpayer($X, $FN, $LN, $S, $C, $T),
      $LN="Smith", $T != null
```

The system will rewrite one update statement on the semistructured data into one or several update statements on the relational store and the overflow graphs. Summarizing:

Theorem 5.2 *Assuming that the STORED query is lossless, the method oulined above correctly translates an updated over the semistructured data model into a set of updates on the relational store and the overflow graphs.*

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[6] The DBLP is 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 alot of information. For example, the top level contains books, conf, journals, ms, persons, and phd directories. Under conf, there is one subdirectory for each conference (255 in total), and each conference directory contains all publication files, for all years. The directory information, however, can be fully recovered from the publications themselves. A typical entry is:

```
<inproceedings key="Abiteboul97">
<author>Serge Abiteboul</author>,
<title>Querying Semi-Structured Data.</title>,
<pages>1-18</pages>,
<year>1997</year>,
<booktitle>ICDT</booktitle>,
<url>db/conf/icdt/icdt97.html#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, such as the following structured title field:

```
<inproceedings key="GorePSV98">
<author>Rajeev Gor&eacute;</author>,
<author>Joachim Posegga</author>,
<author>Andrew Slater</author>,
<author>Harald Vogt</author>,
<title>System Description: <I>card</I>
    <I>T</I><SUP><I>A</I></SUP><I>P</I>:
    The  First Theorem Prover on a Smart Card.
</title>,
<pages>47-50</pages>,
<year>1998</year>,
<booktitle>CADE</booktitle>,
<url>db/conf/cade/cade98.html#GorePSV98</url>
<ee>http://link.springer.de/link/service/
      series/0558/bibs/1421/14210047.htm</ee>
</inproceedings>
```

There are about 92,000 publication that, when represented as a semistructured data, have 861,000 edges. The total disk space is 95M.

We flattened the DBLP data, by moving all files into a single directory, thus we did not mine the directory structure, but just the data. We chose a minimum support of 8500, which is 8.6%. Publications of type article and inproceedings had minimum support. All others had much less than our minimum suppport; for example, there were only 307 books and 67 phd's. 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.

Fig. 10 contains an example of one generated STORED query with A=8. There were only 8

[6] http://www.informatik.uni-trier.de/~ley/db/about/instr.html

```
FROM Bib.inproceedings $X
        { author: $A1, OPTIONAL{author: $A2, OPTIONAL{author: $A3}},
          OPTIONAL{title: $T}, OPTIONAL{pages: $PP}, OPTIONAL{year: $Y},
          OPTIONAL{booktitle: $B}, OPTIONAL{url: $U}}
STORE R1($A1, $A2, $A3, $T, $PP, $Y, $B, $U)


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

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

Figure 10: STORED query for `A=8`

attributes of high support for `inproceedings`, and all 8 in combination still had high support, therefore a single STORED mapping was generated for `inproceedings`. There were more than 8 attributes of high support for `article`, therefore these `article` objects where split into two relations. The first, `R2`, tries to cover most `article` objects, by requiring only the `author` attribute. The second, `R3`, requires two `authors`, which gives it best chances to store those objects *not* stored in `R2`.

We ran two sets of experiments: one that varied the maximum number of attributes per relation, denoted `A` in Sec. 3), and one that varied total disk space allocated to the relations. The results are shown in Table 2.

Because the data was highly regular, few attributes had high support, so we started with artificially low values for `A` (from 3 to 9). We assessed the quality by the algorithm by measuring the *number of mappings* generated, the *data coverage*, and the number of `nulls`. *No. mappings* is the total number of STORED mappings generated (e.g., 3 mappings for `A=8`, shown in Fig. 10). *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. Recall that our algorithm simplifies the space cost by assuming fixed-length records. *Nulls* represent the amount of space occupied by nulls.

The first table that data fragmentation directly depends on the maximum relation arity. For small `A`'s, many relations are generated, and large objects are split vertically. At query time, split objects must be joined. 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 is limited. Recall from Sec. 3 that a bound on the space is obtained by imposing more required attributes. The effect of requiring more attributes is to decrease the number of nulls and to improve utilization of the relations, but at the expense of covering 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.

7 Conclusions

We have proposed a comprehensive framework for storing and managing semistructured data in a relational database management system. The goal itself is ambitious, because the two models are apparently incompatible. We have argued, however, that many semistructured data instances have a fairly regular structure which, when exploited aggressively, allows the data to be stored in a relational format. Our preliminary experiments using the DBLP bibliography database support this hypothesis and show that an efficient relational mapping can be derived automatically from the semistructured-data instance. Additional experiments using other semistructured sources are necessary.

Our method relies on a declarative mapping from the semistructured data model into the relational model, expressed in the new query language STORED. We have discussed many aspects of this mapping: automatic generation of the relational mappings from a semistructured-data instance, automatic generation of the overflow mappings from a semistructured schema, and query and update rewritings.

We have described two applications: storing and managing existing semistructured data sources and converting relational sources into a semistructured format, such as XML. Some of the most difficult topics addressed in this paper (automatic generation of the relational and overflow mappings mappings) apply only to the former application. We see the latter application as equally, if not more, important than the former, especially as information providers begin to export their relational data in

A	3	4	5	6	7	8	9
No. mappings	9	9	5	4	4	3	3
Coverage	91%	94%	94%	90%	92%	90%	90%
Space	1.19M	1.59M	1.15M	1M	1M	0.9M	1.2M
Nulls	23k	116k	112k	123k	91k	106k	201k

Space	0.5M	0.78M	0.93M	1.0M
Nulls	2.5k	40k	106k	106k
Coverage	59%	77%	90%	90%

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

XML.

The most difficult part of our approach is the automatic generation of the relational mappings. We have proposed one solution, based on data mining, but more work on this problem is needed.