On Storage Methods for Semistructured Data

Denilson Barbosa

dmb@cs.toronto.edu
University of Toronto
10 King's College Rd, Toronto
Ontario, M5S 3G4, Canada

1 Introduction

Semistructured data (SSD) [1, 17] has emerged as an important research topic in the past decade. It is primarily used in data integration [45, 12], where it is usually inadequate to define a single rigid schema that naturally encompasses all data sources involved. Other applications of SSD include electronic document management [4]; scientific databases, where a well known example is the ACeDB genome database [59]; data exchange [16]; and structural information retrieval [29, 44]. Given such a diverse spectrum of applications, one can expect large amounts of semistructured data being used in the near future.

Not surprisingly, semistructured data has attracted the attention of both academia and industry: a considerable amount of work has been done on defining data models and query languages for SSD, for instance. Providing efficient storage and query processing is one of the next challenges. Currently, there are several approaches for storing semistructured data, ranging from using files to full-fledged database management systems. For the latter, relational and object databases are the most common choice, although "native" Semistructured Database Management Systems (SSDBMSs) are starting to be developed.

In this paper, we survey the literature on this topic and classify the methods according to their ability to represent semistructured data and process typical queries. A brief introduction to the field is presented in Section 2, which also presents an example we use throughout the paper. Section 3 presents a taxonomy for the storage methods, which are discussed in the next three sections. A few conclusions are drawn in Section 7.

2 Preliminaries

In this section we establish the notation used throughout the paper. Due to space limitations, we restrict ourselves to bare bones; more can be found in [1, 3, 17].

We adopt a object-oriented terminology: data are represented as objects, which are either atomic or complex. Each atomic object represents a single literal of the usual base types (e.g., string, integer, etc.). A complex object is a set of attributes; each attribute is a $\langle$*label*: string, *value*: object$\rangle$ pair. The type of a complex object is a set of $\langle$*label*, *type*, *cardinality*$\rangle$ tuples, specifying the type and the maximum number of occurrences of attributes, for each label. Each object has a unique object identifier (*oid*); an object *reference* is an attribute whose value is the *oid* of an object. As a convention, assume that system generated *oid*s are provided whenever necessary.

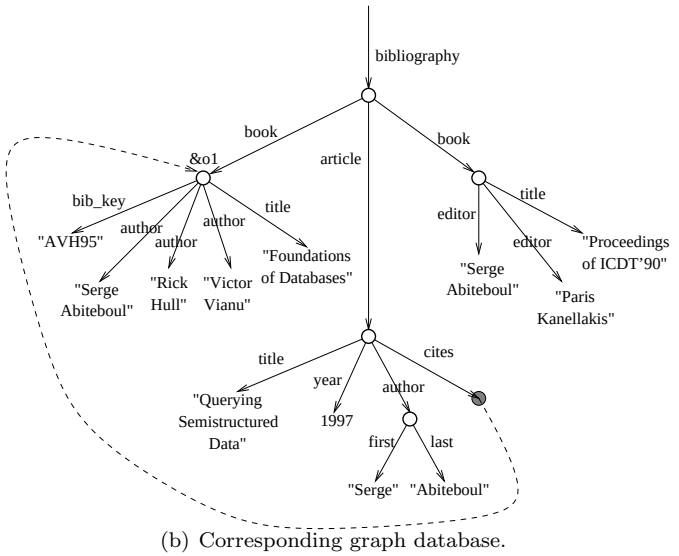
2.1 Semistructured databases

A *semistructured database* (SSDB) is a collection of objects in which attribute labels may be associated with values of different types, even within the same complex object. For this reason, it is usually difficult or impossible to define a schema for the data that has a complete, independent description of all objects in the database. Instead, typically each object has its type annotated to itself, i.e., a "local schema". As a consequence, the actual schema of the database, i.e., the union of all local schemas, is relatively large compared to the data alone, thus blurring the line that separates data from metadata.

Figure 1(a) depicts a textual description of a hypothetical semistructured database of bibliographic entries, which we use as an example throughout the paper. This example is not intended to cover all aspects of semistructured data, rather to give a flavor of the kinds of constructs we will be dealing with. In particular, note the irregular type for `book` objects and the reference from the `article` object to the first `book` object, via its *oid*

```
bibliography:{
 book:  &o1{
  bib_key:  "AHV95",
  title:  "Foundations of Databases",
  author:  "Serge Abiteboul",
  author:  "Rick Hull",
  author:  "Victor Vianu"},
 article:{
  title:  "Querying Semistructured Data",
  year:  1997,
  author:  {first:"Serge",
   last:"Abiteboul"},
  cites:  &o1},
 book:{
  title:"Proceedings of ICDT'90",
  editor:  "Serge Abiteboul",
  editor:  "Paris Kanellakis",
  year:  1990}
}
```

(a) OEM description.



(b) Corresponding graph database.

Figure 1: Example of a semistructured database.

(`&o1`).

Semistructured data models. Semistructured databases are often modelled by rooted directed graphs, in which vertices represent objects and edges represent relationships among objects: *attribute edges* represent the attributes of the objects, while *reference edges* represent references between objects (e.g., the `cites` attribute in our example). Reference edges are useful for avoiding duplicating the description of objects. Figure 1(b) shows a graph that models our example database.

2.2 XML

The Extensible Markup Language (XML) [15] is a standard meta-language developed by the World Wide Web Consortium (W3C) to provide authors with the flexibility of defining their own markup tags. XML is relevant to our discussion since it can also be used to represent semistructured data elegantly [3]. Roughly, a SSDB can be represented by an XML document as follows: objects are represented by *elements*; attributes are represented by nested elements or by *attributes*, which are limited to containing (lists of) literals only; object identifiers are represented by `ID` attributes; and, finally, references among objects are represented by `IDREF` or `IDREFS` attributes. In this way, XML enabled the joining of efforts from the database and the Web communities in what became a very active field.

Although XML can be used to represent semistructured data, the converse is not always true. Since it is a markup language, it imposes the restriction that the elements in a document be ordered. Note that there is no equivalent restriction over the attributes of an object. In order to represent XML content, some semistructured data models include ordering as part of their semantics (e.g., WebOQL Hypertrees [9]). Also, XML documents are usually modelled by trees: essentially the same graph model we defines except no reference edges are represented; instead, references (i.e., `IDREF`/`IDREFS` attributes) are treated as literals. Typical query languages for XML provide primitives for dereferencing `IDREF` attributes, which allow the data to be viewed as a graph. As a result, two interpretations for the same document are possible at query time. Therefore, it is safe to say that XML documents can be viewed as textual descriptions of semistructured databases.

2.3 Schemas and query languages

We defined a semistructured database as a collection of objects that need not conform to pre-specified types; i.e., data with no schema. Evidently, this offers little room for static optimizations for both storing and querying such databases. However, in many situations it is possible to capture the structure of, at least parts of, the database by context-free grammars that specify valid paths in the database graph. In XML terminology, such a grammar is called a Document Type Definition (DTD)[1] and is part

[1]Note that there can be more valid paths in an XML document than the ones described in its DTD, if elements of type `ANY` are

of the standard [15].

Note that context-free grammars can define fairly irregular constructs, allowing infinitely many valid paths in the database. We say such grammars define *loose* schemas for databases. Therefore, we can rephrase our definition of SSDB to a collection of objects that conforms to a loose schema, if one is provided. In the XML jargon, we say a document is *well-formed* if it is syntactically correct, and *valid* if it is both well formed and conforms to a DTD. There are other schema formalisms for XML; we refer the reader to [43] for more on this subject.

Query languages for semistructured data already abound and come in different flavors, with varying expressive power. Typically, languages originating from the database community (e.g., Lorel [6]) provide full data manipulation capabilities while languages originating from the document management community (e.g., XSL [7]) offer a more limited set of operations. Defining a unified query language for XML that serves both purposes efficiently is the task being faced by the W3C Query working group [21].

Although differing in many aspects, the query languages for semistructured data share many features [14]. Of particular interest are the the use of regular path expressions for "navigating" the database (i.e, locating objects in the database); the specification of the structure of the result of the query; and the definition of *structural queries* (i.e., queries that range over both data and schema). A through analysis of these languages is outside the scope of this work; we refer the interested reader to [14, 28, 33].

For the sake of consistency, we say a query language for semistructured data is a *semistructured query language* (SSQL).

3 Storage methods for semistructured data

As more applications for semistructured data arise, it becomes more important to provide generic and efficient mechanisms for storing and querying this kind of data. Similarly to traditional databases, desirable features for these systems are persistent data storage, data independence, support for a declarative query language and the ability to enforce the integrity of the data. As this technology evolves, other standard features of traditional DBMSs (e.g., concurrent access) will become available.

3.1 A taxonomy for the storage methods

There are already many storage methods for semistructured data, using different mechanisms for providing persistence and query processing. We classify them according to the following categories:

used.

1. Files: a textual description of the database (e.g., Figure 1(a)) is stored in one or many files. Two persistence mechanisms are distinguished: the file system or a DBMS.
2. DBMS mappings: a mapping from the semistructured data model to a database schema is defined for storing and querying the data. We further classify these methods according to these (orthogonal) criteria:
   - The target schema: relational, object-oriented or hybrid;
   - The need for a schema: some methods rely on loose schemas for defining mappings, while others do not require or exploit one, when available.
3. Native Methods: the database is stored and queried by special purpose engines.

4 Files

The most obvious storage approach is to keep a description of the database in (possibly many) text files, which are loaded into suitable memory-resident structures whenever processing is necessary. The key idea for this method is that the data can be access via high-level abstractions, defined over the files (e.g., *structuring schemas* [5]). In this way, one can define indices for improving query performance, for instance.

4.1 File system approach

The methods in this category rely on the machine's file system to provide persistence for the files comprising the database. Hence, they focus solely on processing queries over these files.

Kweelt. Kweelt [52] is an implementation of the Quilt [22] query language for XML documents. In Kweelt, the documents are parsed and loaded into memory-resident Document Object Model (DOM) [63] structures, which is part of the W3C family of XML related standards. In this model, documents are represented by trees (a.k.a. DOM trees). The DOM model defines also a sophisticated API for traversing the document tree, which is powerful enough for Kweelt's needs. Query results in this method are XML documents as well.

4.2 RDBMS approach

The main problem with the previous method is that the paraphernalia used for query processing (the DOM trees) are kept in main memory and have to be rebuilt every time a query is issued. One way of overcoming this limitation is to use a DBMS for storing the files comprising the database, and define indices over these files for

```
<XColumn>
   <table name="book_side_tab">
     <column name="title" type="string"
       path="/bibliography/book/title"
       multi_occurrence="NO"/>
     <column name="bib_key" type="string"
       path="/bibliography/book/@bib_key"
       multi_occurrence="NO"/>
     <column name="year" type="integer"
       path="/bibliography/book/year"
       multi_occurrence="YES"/>
   </table>
   ...
</XColumn>
```

Figure 2: Definition of side tables in DB2.

processing queries. Current DBMSs allow the storage of text files either internally, as character large objects (CLOBs), or externally, in the file system.

XColumns. IBM's DB2 XML Extender [39] offers two mechanisms for storing XML documents: XCollections (discussed in Section 5.1.2) and XColumns, in which the documents are kept intact. The XColumns approach is intended primarily for small documents that do not change often and whose elements are not queried very frequently. Data heavily used for querying can be copied into *side tables*, which are indexed, as any other table in the system. Figure 2 shows a the specification of a side table for `book` elements in our example; note that `title` and `bib_key` are candidate keys. Side tables are updated by DB2 every time a document is modified, added to or removed from the system.

The query language for this method is an extension to SQL for allowing regular path expressions in `WHERE` clauses. Access to external files is provided by DB2's Text Extender, which relies on inverted files [38] for accessing the elements. Structural queries are processed in a similar way. Query results are relations.

4.3 Discussion of methods based on files

This approach is potentially inefficient since the data has to be parsed, loaded into memory, processed and, for update operations, dumped back to disk. This situation is ameliorated by the use of a DBMS for storing indices to speedup the access to the data. Nevertheless, at the time of writing, we are not aware of any DBMS that has such functionality built in at its core. Therefore, the query processors only accesses the data via calls to external modules, which can be expensive. Also, little or no improvement might be achieved if a query refers to elements not indexed by the system.

Edge

source	ord	name	flag	target
1	1	book	ref	2
1	2	article	ref	3
1	3	book	ref	4
2	1	bib_key	string	v_1
2	2	title	string	v_2
2	3	author	string	v_3
...				
3	1	cites	string	v_6
3	2	title	string	v_7
3	3	year	integer	v_8
3	4	author	ref	5
...				
5	1	first	string	v_{12}
...				

V_{String}

v_{id}	value
v_1	AHV95
v_2	Foundations of...
v_3	Serge Abiteboul
...	
v_6	AHV95
...	
v_{12}	Serge
...	

V_{Integer}

v_{id}	value
v_8	1997
v_{14}	1990

Figure 3: Edge Table mapping.

5 DBMS mappings

Most of the current methods lie in this category. Nevertheless, they differ greatly in many aspects. For presentation convenience, we show the relational mappings first, then move to hybrid relational storage methods, and finally discuss the use of an object database for storing semistructured data.

5.1 Relational mappings

The goal for the methods in this section is to exploit currently available relational DBMSs as storage mechanisms for semistructured data. The approach here is to define a mapping from the graph model into relations; such mapping is then used for translating queries over the original database into equivalent SQL statements over the relational database. Interestingly, the methods discussed here in many ways resemble the ones proposed for mapping semantic data models [47].

We further classify the methods in this section according to the way query results are presented to the user (*relational* or *semistructured*).

5.1.1 Structure-independent mappings

The approach for the methods in this section is to define a generic mapping, independent of the structure of the

book

id	label	ordinal	target
2	bib_key	1	v_1
2	title	2	v_2
2	author	3	v_3
...			
4	title	1	v_9
4	editor	2	v_{10}
...			

Figure 4: Result of evaluating the path expression `/book` over our example database.

databases being mapped. Note that these methods do not require a schema for the database being mapped.

Edge Tables. The method in [35] was one of the first proposals for using relational databases for storing XML data. Relations are used to store the edges (the *Edge table*) and the leaves of the tree (the *value tables*), for which string and integer literals are distinguished[2]. The *Edge* table stores, for each edge, its source and destination vertices, label, order within the parent element (the source vertex), and a flag to indicate the type of the destination vertex (atomic or complex object). See Figure 3 for the mapping of our example database.

In this method, user queries are written directly in SQL. Path expressions are evaluated by repeatedly joining the edge table with itself, and, eventually, with the value tables. For example, consider evaluating the following query over the mapping of Figure 3: "retrieve all books in the database", which corresponds to the path expression `/book`. The following SQL:1999 [30] statement retrieves all subtrees in the graph whose root matches the path expression:

```
WITH RECURSIVE book(id, label, ordinal, target) AS
   ((SELECT source, name, ordinal, target FROM Edge E2
     WHERE EXISTS (SELECT * FROM EDGE E1 WHERE
                   E1.name='book' AND E1.target=E2.source
                   AND E1.source=1))
     UNION ALL
    (SELECT E.source, E.name, E.ordinal, E.target FROM
     Edge E, book B WHERE b.target = E.source))
```

The result of this expression is shown in Figure 4. Note, however, that to obtain the result of the original query we still need to join the *book* relation with all value tables, which becomes more expensive as more data types are allowed.

Two variations of the method are proposed. The first, called binary relations, consists of partitioning the *Edge* table according to the labels of the edges. This means creating one table per label in the graph, with the same attributes in the *Edge* table except label, which names the relation itself (a similar approach is taken by [61]). In the second variation, a universal relation is used for storing the edges; such relation can be viewed as a full outer join of all binary relations. The paper also considers *inlining* some leaf nodes in the graph, i.e., storing them directly in the edge tables. An inlined relation is the equivalent of a left outer join of an edge table with all value tables. Figure 8 shows the inlined version of the database in Figure 3.

Both variations aim at reducing the number of joins required for querying the database but come at a certain cost. First, the relational schema depends on the labels in the semistructured database. Thus, when new edge labels are found, the schema of the mapped database has to be modified (either by adding new binary edge tables, or by adding columns to the universal relation). On the other hand, the user may decide whether to remove the corresponding relations or columns when an edge label is no longer present in the database. Inlining also reduces the number of joins for query processing by the expense of representing many *null* values. All three mapping schemes provide relational results.

Node Tables. While in the previous approach the edges were stored in relations, in [57], nodes receive the same treatment. This method is also intended for storing XML documents. In this method, each element corresponds to a *region* [23], defined as follows. Each word is assigned a number corresponding to its order of occurrence in the document (i.e., `"AHV95"` $\rightarrow 0$, `"Foundations"` $\rightarrow 1$, etc.); tag names are ignored. A region is denoted by a pair of integers, corresponding to the number associated with the first and the last words in the element. For instance, the first `book` element comprises the region $\langle 0, 9\rangle$ (i.e., from `"AHV95"` to `"Vianu"`), and its second `author` corresponds to the region $\langle 6, 7\rangle$ (i.e., from `"Rick"` to `"Hull"`).

The mapping works as follows[3]. All paths in the document are uniquely identified and stored in the *Path* relation. Each inner node of the graph is stored in the *Element* relation, together with the delimiters of its region and its corresponding path identifier. The leaves of the tree are stored in the *Text* or *Attribute* relations, containing their path identifiers, regions and actual literal values.

User queries are expressed in XQL [51], and automatically translated into SQL. Processing path expressions in the mapped database consists of first selecting all the paths that match the expression and, for each match, verifying which inner nodes to retrieve. Once these elements are known, it is easy to reconstruct the subtrees rooted at them (i.e., extract their attributes and text nodes), by region containment. This method provides

[2]The XML standard defines only the string datatype, but other schema languages offer more datatypes [32].

[3]Non-essential details of the mapping are omitted here.

relational results.

Monet Mapping. The method described in [53] defines mappings from XML documents into databases using the Monet data model [13], which is based on binary relations. The mapping is based on *associations*, defined as pairs $(o, \cdot) \in \mathsf{oid} \times (\mathsf{oid} \cup \mathsf{int} \cup \mathsf{string})$[4], which correspond to the edges in the graph data model as follows. Associations of type $\mathsf{oid} \times \mathsf{oid}$ represent edges between elements (denoted by $\xrightarrow{e}$); associations of any other type represent edges between elements and attributes (denoted by $\xrightarrow{a}$).

Note that each path in the document defines many associations. For example, the path *bibliography* $\xrightarrow{e}$ *book* defines associations of the form $\langle o_1, o_i \rangle$, where o_1 is the *oid* of the root element in the graph and o_i is an *oid* of an element that can be reached from the root, following an edge labeled `book`. Similarly, the path *bibliography* $\xrightarrow{e}$ *book* $\xrightarrow{a}$ *bib_key* defines associations between the *oids* of all elements reachable from the root by traversing a `book` edge with string literals, corresponding to the values of the `bib_key` attributes for the respective `book` objects. Each association is stored in a separate binary relation, named by its corresponding path. Textual elements are represented as `string` attributes of `cdata` elements, for simplicity.

No specific query language has been defined for this method. Instead, the authors illustrate how OQL-like queries could be evaluated in the Monet system, which incorporates an algebra and extensions to the SQL language, due to its fragmented nature [13]. It is easy to see that evaluating path expression implies identifying and repeatedly joining binary relations. Note that this method suffers from the same problem of the binary edge table, namely: the relational schema depends on the paths in the database, and changes in the the data might require updates in the relational schema. We assume query results are relations in this method.

5.1.2 Structure-dependent mappings

The methods discussed in the previous section can be viewed as mechanisms for storing generic graphs in relational databases. A different approach is taken for the methods in this category: exploit the structural properties of the database to generate the relational schema.

A possible DTD for the XML document corresponding to our example database and used for the following discussion could be:

```
<!DOCTYPE bibliography [
   <!ELEMENT book (title, (author*|editor*), year?)>
   <!ATTLIST book bib_key ID #IMPLIED>
   <!ELEMENT article (title, year, author*)>
   <!ATTLIST article cites IDREF #IMPLIED>
   <!ELEMENT author (ANY)>
]>
```

[4]This method also distinguishes strings and integer literal types.

```
article (articleID: integer, article.title: string, article.year: string,
  article.cites: string)
book (bookID: integer, book.title: string, book.year: string,
  book.bib_key: string )
author (authorID: integer, authorParentID: integer,
  author.isRoot: boolean, author: string)
editor (editorID: integer, editorParentID: integer, editor: string)
```

(a) Relational schema.

```
article = {(5, "Querying Semistructured Data", "1997", "AHV95")}
book = {(1, "Foundations of Databases", null, "AHV95"),
        (7, "Proceedings of ICDT", "1990", null)}
author = {(2, 1, false, "Serge Abiteboul"),
          (3, 1, false, "Rick Hull"),
          (4, 1, false, "Victor Vianu"),
          (6, 5, false, "<first>Serge</first><last>Abiteboul</last>")}
editor = {(8, 7, "Serge Abiteboul"),
          (9, 7, "Paris Kanellakis")}
```

(b) Database.

Figure 5: DTD Mapping for our example database.

Note that `author` elements in this DTD are declared to be of type `ANY`, meaning that no structural restrictions apply over them.

DTD Mapping. The method described in [56] aims at deriving relational database schemas from DTDs. Since a DTD can define fairly complex structures, a set of "simplifying transformations" are used to derive a less restrictive DTD, which is then used for defining the mapping. As a result, any valid document according to the original DTD can be stored in the database, but the converse is not true. In our example, the `book` element is simplified to `<!ELEMENT book (title, author*, editor*, year?)>`, allowing the storage of books with both authors and editors, which was not permitted originally.

In our example, the mapping is derived by the following heuristics: create relations for elements with in-degree 0 (`book` and `article`); create relations for elements that can occur multiple times (`author` and `editor`); finally, inline the remaining elements as columns in the relations corresponding to their parents. The nesting of the other elements is preserved by `ID` attributes (See Figure 5). Since our DTD allows `author` elements to occur independently of `book` or `article` elements, the isRoot attribute is used to distinguish these cases. Note that elements of type `ANY` are stored as unmapped strings[5]. This method has another mapping rule for dealing with recursion, not required in our example.

XML-QL [26] is the query language for this method: queries are translated into SQL and the results are formatted as XML (i.e., the query results are semistruc-

[5]This makes this method a hybrid one (discussed in Section 5.2). We present it here since it is the only that aims at inferring a relational schema solely from a grammar.

```
<DAD>
   ...
   <element_node name="bibliography">
     <element_node name="book">
       <element_node name="title">
         <text_node>
           <RDB_node>
             <table name = "book_tab"/>
             <column name = "book_title"/>
           </RDB_node>
         </text_node>
       </element_node>
       ...
     </element_node>
     ...
   </element_node>
</DAD>
```

Figure 6: DAD file mapping for `book` elements.

tured). Note that processing structural queries may require access to both the relational database schema (e.g., to determine that `book` has a `title` sub-element) and instance (e.g., to determine that `book` has an `author` sub-element).

Not all productions in a DTD are covered by the this method. One simple example is the rule `<!ELEMENT bibliography ((book|article)*)>`, which is not only perfectly valid but also common in practice. Another example is `<!ELEMENT author(#PCDATA|(first,last))>`, which defines an element of mixed content and is also valid in our database. We believe the mapping generation algorithm could be extended to deal with rules such as the ones presented above. For instance, a mapping for the first rule could be to define a new relation (bibliography), containing bibliographyID as attribute, and include corresponding parentID attributes in both the book and article relations. For the second rule, one could inline all three elements. However, this strategy might yield the excessive representation of *null* values in the database; we return to this problem in Section 5.4.

XCollections. In the XCollections [39] approach, literals in the document are mapped into columns of relations. The mapping is specified by a Document Access Definition (DAD) file. A fragment of one mapping for our example database is shown in Figure 6. Note that the mapping mimics the nesting of the elements in the document: elements matching the path `bibliography.book.title` are stored in the `book_title` column of the `book_tab` relation. The `<text_node>` tag specifies that a matching element should contain only text. Attributes are mapped in a similar way.

This method allows not only the integration of XML and relational data but also the dynamic generation of XML content from the database. The DAD file can also be used to specify a mapping in the opposite direction, consisting of an SQL query and the "skeleton" for the resulting XML document. Thus, query results in this method are semistructured, although only for queries defined *a priori*.

Other methods in commercial DBMSs. The storage of XML content has attracted the attention of most DBMS vendors. Besides the XColumn and XCollection methods of IBM's DB2, Oracle, Microsoft and Sybase products are compared in [25]. In summary, all these products offer similar functionalities, differing essentially in the way the mappings are specified and executed. IBM and Microsoft products support *declarative* mappings, i.e., the data is loaded by their systems, as specified by the mapping. In both Oracle XML-SQL Utility [11] and Sybase, the user has to write an application for loading the data; system libraries are provided for facilitating the coding. We discuss the generation of XML documents from relations in these systems below.

5.1.3 Discussion of relational mappings

The high popularity of RDBMSs is definitively a strong motivation for exploiting this technology for storing semistructured data. However, as one might expect, this approach suffers from some problems inherent to the hierarchical nature of the data. We next discuss the major challenges.

Query processing. We identify three major problems related to (semistructured) query processing for relational storage methods. First, as mentioned in Section 2.3, SSQLs allow the definition of the structure of the query result, unlike SQL whose result sets are "flat" relations. In general, query processing in these systems can be viewed as a three-step process: translating the user query to SQL statements, processing them, and formatting the result set according to the rules specified in the original user query. The latter step can be generalized as publishing relational data as semistructured data, which we discuss below.

Processing structural queries is another problem, since it typically requires querying both the relational database schema and instance, and the SQL standard does not provide explicit mechanisms for the maintenance of catalogs [24]. As a result, the translation of such queries may vary depending on the DBMS chosen, compromising the portability of the methods.

The third problem concerns the expressive power of typical SSQLs. Reachability and connectivity queries, which are trivial to express using path expressions, are known not to be expressible in SQL-92, which is the SQL standard currently supported by most vendors. Object-oriented extensions to the relational technology, such as persistent stored modules [24], aim at providing more

computing power to SQL, and can be used for computing such queries. However, this leaves the user with the responsibility of defining algorithms for such computations, which might greatly impact query processing performance in the system. Another alternative is to use SQL:1999, as in our example query in Section 5.1.1. However, the current syntax for defining recursive queries requires the user to know whether there are cycles in the database graph, and to deal with them accordingly [20], thus complicating the process of writing queries, be it automatic or manual.

Publishing Relational Data as SSD. The problem of publishing relational data as semistructured data has received considerable attention from both academia and industry [18, 25, 34, 55]. Among the methods covered here, [56, 39, 11] offer semistructured output, although the latter two do so only for predefined queries.

Most RDBMS vendors currently support the generation of XML content from relations. The Oracle and Sybase products define their own XML syntax for representing relational databases (i.e., tags for representing relations, tuples, etc.). IBM DB2 and Microsoft SQL Server, on the other hand, offer the user the possibility of defining arbitrary tag names in the resulting document, by providing explicit mappings from columns into elements [25].

Validity checking. Checking whether the data is valid is a very important and common operation, for any kind of database. The most simple notion of validity for SSDBs is conformance with the grammar specified in their schemas. Note that checking the structural validity of a SSDB consists of verifying two things: first, all paths in the database are defined by the grammar; and second, that certain paths do exist in the database. For instance, consider the DTD used in our example: whenever we have an object that is reachable by the path `bibliography.article.title`, we need exactly one object matching `bibliography.article.year` and one or more objects matching `bibliography.article.author`. Moreover, all these objects must descend from the have the same `article` object. Note that verifying such properties requires querying the database, which can be an expensive operation.

The relational technology provides a rich set of integrity constraints used to define validity for database instances. Among these constraints, the referential integrity is one of the most commonly used and efficiently enforced. Note that referential integrity alone can capture most of the structure of a relational database. Unfortunately, it might be the case that more sophisticated constraints are needed for enforcing the validity of mapped databases. As an example, note that it is possible to define different cardinality constraints for the attributes in the SSDB, which cannot be enforced simply by referential integrity. As another example (using the DTD mapping), note that one cannot enforce that all authorParentID values in the author relation refer to either a book or an article tuple using referential integrity alone. For such a simple case we need to define an *assertion* [24], which typically has a much higher cost of evaluation when compared with referential integrity constraints.

```
M1a = FROM bibliography.article:$X
      {title:  $T, year $:Y, OPT {cites:  $C}}
      STORE Article($X,$T,$Y,$C)
M1b = FROM bibliography.article:$X.author:$A
      KEY $X,$A
      STORE Author_Article($A,$X)
M1c = FROM bibliography.article:$X.author:$A
      {$O:_}
      OVERFLOW G($O)
```

Figure 7: STORED mapping.

Note that more sophisticated integrity constraints can be defined by more powerful schema languages; enforcing such constraints over mapped relational databases might be too expensive. None of the methods discussed in this section address this problem. Finally, one should note that the validity of the mapped database has to be enforced also after updates are processed. However, as a matter of fact, updating semistructured data is a subject that has not received the attention it deserves; a recent proposal for an update language for XML is presented in [60].

5.2 Hybrid mappings

As the name suggests, hybrid methods store both mapped and unmapped content (i.e., only part of the data is mapped). Typically, the portions of the data with regular structure are mapped while the remaining "outlier" objects are stored separately.

STORED. The underlying data model for STORED [27] is an ordered version of the Object Exchange Model (OEM) [48] (see Section 6), which can be used to represent XML content as well. In this method, a semistructured database is mapped into a *mixed schema*, consisting of a relational schema for the mapped data, and a *graph schema*, used to store the unmapped portions of the data. The mappings are generated by a data mining algorithm, described in [62], which takes as inputs the expected query mix and properties for the desired relational database, and outputs the mixed schema plus a collection of queries in the STORED (Semistructured TO RElational Data) query language. This method does not require a schema, but one is exploited if provided for

deriving more efficient graph schemas.

STORED queries can be modified by the user, if necessary. They resemble XSL stylesheets, in the sense that both are applied on a pattern-matching basis. Each query has the form `FROM WHERE STORE`, and is defined as follows. The `FROM` clause specifies a pattern that binds variables when it matches some objects in the semistructured database; the `STORE` clause specifies how these objects are represented in the database and the `WHERE` clause specifies restrictions over them.

Recall Figure 1(a) that shows a textual OEM description of our example. Figure 7 shows three rules for mapping `article` objects. Rule `M1a` specifies that the `title`, `year` and `citation` attributes are stored in the `Article` relation (note that variable `$X` matches the *oid* of the `article`). Rule `M1b` specifies that atomic `author` attributes are stored in a separate relation; the (composite) key of that relation is also specified. Finally, rule `M1c` states that complex author objects have to be stored in an *overflow* graph (i.e., are not mapped). Overflow graphs can be implemented in different ways. For instance, one could use the Edge Table mapping method.

No specific query language is defined for this method. Instead, the authors use an hypothetical language possessing most of the common features found in current languages. User queries are rewritten into SQL; update operations are also translated. Query results are relations for this method.

5.3 Object databases

The methods in this category are based on the assumption that SSDBs are better captured by object-oriented data models than by the relational model. Note that all the methods discussed earlier could also be implemented using an object database as storage engine.

Ozone. Ozone [42] extends the ODMG [19] data model and query languages for dealing with semistructured data. Following our taxonomy, it is the object-oriented counterpart of the hybrid relational storage methods (discussed in Section 5.2), since it is implemented on top of O_2 [10] and allows the storage of OEM objects by the addition of a new base type to the system. Thus, the user can map part of the data into ODMG objects and store the rest as OEM objects. Ozone allows the storage of XML data as well, via ordered collections of OEM objects. Ozone's query language (OQL^S) was largely influenced by Lorel [6] (see Section 6).

Note that although Ozone is as a hybrid method, it allows structured data to be viewed as OEM objects, and vice-versa. From the user's perspective, this means that the database can be treated as a semantically homogeneous entity, manipulated by an "unified" query language (contrasting with other hybrid methods).

Other methods. There are already some commercial products for storing XML data based on object databases: POET [49] and eXcelon [31] are examples. Typically, these products offer persistent storage of documents and APIs for accessing them, such as DOM, with bindings for common object-oriented programming languages, such as Java and C++.

5.4 Discussion of DBMS mappings

The methods presented in this section have the same goal: build on current DBMS technology to store semistructured data. However, they differ in many aspects. Next we make some general observations about the methods discussed in this section.

Manual versus automatic mappings. In the XCollections and Ozone methods, the user is responsible for defining the mappings, which have to cover all objects in the database and all relationships among them. One possible way of doing this could be specifying mappings for every path from the root of the database graph. Note that the user has also to decide how to deal with reference edges.

Manually defining a mapping for a complex semistructured database might be a tedious and error-prone task. Moreover, all valid ways of structuring the data have to be known by the user; if a schema is not provided, inspecting the entire database might be required. For these reasons, manual mappings are suited only for databases with simple and regular structure. The mappings in all other commercial DBMSs fall in this category as well.

The need for a schema. Structure-sensitive methods require and exploit a schema when defining the mapping of a given database. Structure-insensitive methods, on the other hand, independ on schemas and, thus, are better suited for applications when the structure of the data is too complex, unknown or changes frequently. Note that although a schema can always be inferred [58], the cost of modifying the relational database schema whenever new structural constructs are added to the database might render the structure-sensitive methods inadequate for databases with changing structure. Note that the same applies for some structure-insensitive methods (the binary version of Edge Tables and the Monet mapping).

In summary, the choice of the most appropriate relational mapping depends on many characteristics of the database in hand. Depending of how irregular is the database structure and how frequently it changes, hybrid or structure-insensitive mappings such as the "standard" Edge Tables should prove more adequate.

Relational versus object databases. There are two main

Edge

source	ord	name	flag	target	V_{string}	V_{integer}
1	1	book	ref	2	*null*	*null*
1	2	article	ref	3	*null*	*null*
1	3	book	ref	4	*null*	*null*
2	1	bib_key	string	*null*	AHV95	*null*
2	2	title	string	*null*	Foundations of...	*null*
2	3	author	string	*null*	Serge Abiteboul	*null*
...						
3	1	cites	string	*null*	AHV95	*null*
3	3	year	integer	*null*	*null*	1997
3	4	author	ref	5	*null*	*null*
...						
5	1	first	string	*null*	Serge	*null*
...						

Figure 8: Inlining in an *Edge* table.

challenges when using relational technology for representing a semistructured database: capturing its nested structure, and coping with irregularities in this structure. With respect to the first problem, object-oriented data databases do better than their relational counterparts since they more naturally represent complex, deeply nested data. However, irregular structures are an issue for both technologies. Also, there are still many problems related to storage, indexing and query processing in object databases [50].

The choice between one of these platforms might be guided by non-technical reasons, however. The amount of relational data currently being used and the affordability of current RDBMSs is definitely a very strong motivation for exploiting this technology for storing semistructured data.

Hybrid versus non-hybrid methods. A hybrid method might offer an attractive compromise solution for the cases in which parts of the data present somewhat regular structures and are often used for query processing. This leaves us with the task of determining which parts of the database to map and which to leave untouched. Unfortunately, identifying the most common structures in a semistructured database, given the properties for the intended relational database and a set of querying costs (i.e., a query mix), is NP-hard on the size of the database [27], which is discouraging. Defining efficient heuristic mining algorithms becomes an important problem. Another hybrid method, which uses a nested relational model [2] and a similar mapping algorithm, is presented in [41].

Note that a drawback for using a hybrid method is that two different schemas have to be integrated by the applications. The differences may go beyond the way in which objects are represented and manipulated: performance discrepancies are to be expected as well.

Representation of null values. We conclude with a discussion on the need for representing of *null* values in the relational database, due to irregularities in the structure of the data being mapped. Except for the inlined and universal Edge Tables in [35], structure-insensitive methods do not *require* the representation of *null* values, which is not the case for structure-sensitive methods.

Figure 8 shows the inlined version of our Edge Table mapping (Figure 3). Note that in each tuple, exactly one of the last three columns is non-*null*, which might require the fine tuning of physical storage and indexing structures. This problem grows worse as more datatypes (thus more *Value* tables) are allowed. One interesting analysis in [27] is the percentage of *null* values in the database as a function of the parameters for the mapping algorithm, which could be part of a benchmarking suite for SSDB storage methods.

6 Native methods

Native storage methods are based on the assumption that current database technology cannot provide adequate performance for storing SSDBs. Their most common argument is the inherent *fragmentation* of the data when using traditional DBMSs. For example, consider storing books formatted in XML in a database consisting of collections (relations or classes) of chapters, sections, paragraphs, etc. Retrieving say a chapter of a book, implies accessing many collections (possibly physically distributed), thus yielding poor performance. The goal of the methods in this section is to implement such operations efficiently while providing traditional DBMS functionality, such as indexing, declarative query processing, query optimization, etc.

Lore. Lore (Lightweight Object Repository) [46] was one of the first management systems for semistructured data; this system was recently modified to handle XML data as well [36]. Lore is based on the OEM model, a simple and self-describing nested object data model. Lore's architecture is rather conventional, as are the main operations implemented by its query processor and physical data manager. There are, however, some novelties in the indexing of the data which may require the development of new query optimization techniques [46].

As already mentioned, OEM represents semistructured data naturally (Figure 1(a) is a textual description of an OEM representation of our example database). Lorel [6] is Lore's query language: an extension of OQL to handle regular path expressions [3]. It provides both the customary query operations and also updates, which makes it one of the most powerful query languages for semistructured data [14]. Lorel queries are translated

into an OQL-like form at evaluation time, thus enabling the use of traditional optimization techniques.

One of the innovative aspects of Lore is the use of DataGuides [37] as structural summaries of the database, representing its structure and storing statistics about its objects, used for query optimization. Every path in the database has to be represented in the DataGuide, which is updated every time the database is. For this reason, one can think of a DataGuide as a "dynamic schema". Lore's user interface allows the visualization and browsing of DataGuides, which then serve as "user guides" to the database it represents.

Future work for this project includes providing other typical DBMS features, such as transaction management, views and triggers. Also, extending OEM towards a temporal data model and developing more efficient query optimization techniques, especially tailored to Lore's internal storage mechanisms.

NATIX. NATIX [40] is a physical storage engine for tree-structured data, especially XML documents. Its underlying data model is ordered trees. The core of NATIX is a classical record manager, responsible for disk and memory management, as well as buffering. Other components of the system include an index manager, a schema manager and a document manager. The latter provides access to the data at the document and element levels of granularity; its typical operations are the decomposition (and subsequent recomposition) of documents into physical records.

NATIX memory space is divided into equally sized (up to 32KB) pages, which can hold a variable number of records, each of them containing exactly one subtree of the database. Three kinds of nodes are defined: literal nodes, containing uninterpreted sequences of bytes (the leaves of the tree); aggregate nodes, representing *inner* nodes of the tree; and proxy nodes, used to link subtrees in different records. The method consists of efficiently encoding subtrees within records and maintaining this "physical tree" as the document is updated. The maintenance policy can be customized to specify the average expected occupancy of records and, for each pair of elements, whether or not to store them in the same record.

In its current implementation, NATIX supports XPath as its query language. Future work for the project include studying indexing mechanisms and the performance of different maintenance policies.

Other native methods. There are commercially available native storage methods as well. Tamino [54], developed by Software AG, allows the native storage of XML data and integration with relational sources as well. XQL was chosen as the query language.

6.1 Discussion of native methods

As mentioned earlier, the motivation for the methods in this section is the assumption that efficient storage of semistructured data cannot be achieved by traditional database engines. It remains to be seen, however, whether better performance can be delivered. Many techniques remain unproven technology and open problems abound. For instance, there are many DataGuides for each SSDB, with different creation and maintenance costs. Although preliminary experimental results indicate acceptable performance, the worst case complexity for building a DataGuide is exponential space and time on the size of the database [37], clearly not satisfactory.

Applications built on top native storage methods are starting to flourish. One notable example is Xyleme [8], which aims at being a repository for all XML data on the Web, and uses NATIX as underlying storage method. Besides storing and querying (in OQL fashion) XML documents, Xyleme deals also with collecting and classifying documents on the Web; refreshing its repository; processing (potentially millions) of user queries; and delivering XML content to users through a query subscription system.

7 Conclusions

In this paper we justified the need for efficient storage and query processing of semistructured data and surveyed the literature on alternative methods. Current approaches range from the use of files and native storage mechanisms to the use of traditional (both relational and object-oriented) database management systems. A summary of the methods is presented in Table 1. The second column in that table shows the data models perceived by the user, as opposed to the ones used by the actual storage engine. Given such diverse universe of possibilities, it becomes evident that a benchmark suite tailored for this domain is required for objectively comparing the performance of these methods.

The methods discussed here show that storing semistructured data does in fact require the development of new technologies. Query processing and validity checking, for instance, are particularly hard to perform using current DBMSs. Also, one cannot forget typical document management operations, such as browsing, which might prove inadequate for current technology as well. A benchmark suite should take such operations into account. An interesting question is then whether extensions to current technology alone will suffice for typical applications.

Also, one cannot overlook the amount of structured data currently available. Most businesses use relational technology and, apparently, this scenario is not changing in the foreseeable future. Therefore, integration with tra-

Method	Data Model	Query Language	Persistence Mechanism	Semistructured Output?	Nulls?
Kweelt	XML	Quilt	File System	Yes	N/A
XColumns	XML	SQL[a]	File System or BLOB	No	N/A
XCollections	Relational	SQL	Relations	Yes	Yes
Edge Tables	Relational	SQL	Relations	No	No
Node Tables	XML	XQL	Relations	No	No
Monet Mapping	Monet	N/A	Binary Relations	No	No
DTD Mapping	XML	XML-QL	Relations	Yes	Yes
STORED	Ordered OEM	N/A[b]	Relations / Native	No	Yes
Ozone	ODMG / OEM	OQL^S	O_2	Yes	Yes
Lore	OEM	Lorel	Native	Yes	N/A
NATIX	Ordered Trees	XPath	Native	Yes	N/A

[a] Modified to allow path expressions.

[b] The authors use a generic query language for semistructured data for presenting their work.

Table 1: Summary of the storage alternatives.

ditional database applications seems to be a requirement for any semistructured data storage method. Evidently, this problem is aggravated for file-based, object-oriented and native storage methods.

Finally, the advent of more powerful schema formalisms for semistructured data (e.g., XML Schema allows the specification of user-defined datatypes) may offer new opportunities and challenges. However, one has to keep in mind that semistructured databases might come without schemas. Therefore generic storage methods should not rely exclusively on their existence.

Acknowledgments

The author would like to thank Leonid Libkin, Renée Miller and John Mylopoulos for their comments on a earlier draft of this paper. Special thanks go to Alberto Mendelzon, for his many invaluable comments and careful reviews, as this work evolved into its final form.

References

[1] S. Abiteboul. Querying semi-structured data. In *Proceedings of the 6th International Conference on Database Theory (ICDT'97)*, Delphi, Greece, January 8-10 1997.

[2] S. Abiteboul and C. Beeri. The power of languages for the manipulation of complex values. *VLDB Journal*, 4(4):727–794, October 1995.

[3] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kauffman Publishers, Inc., 1999.

[4] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1):5–19, April 1997.

[5] S. Abiteboul, S. Cluet, and T. Milo. A logical view over structured files. *VLDB Journal*, 7(2):96–114, 1998.

[6] S. Abiteboul, D. Quass, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–99, April 1997.

[7] S. Adler, A. Berglund, J. Caruso, S. Deach, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, and S. Zilles. Extensible stylesheet language (XSL) version 1.0 - W3C working draft. Available at `http://www.w3.org/TR/xsl/`, October, 18 2000.

[8] V. Aguilera, S. Cluet, P. Veltri, D. Vodislav, and F. Wattez. Querying XML documents in Xyleme. In *Proceedings of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, July 28 2000.

[9] G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *Proceedings of the Fourteenth International Conference*

on Data Engineering (ICDE'98), pages 24–33, Orlando, Florida, USA, February 23-27 1998.

[10] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an object-oriented database system: The story of O_2*. Morgan Kauffman Publishers, Inc., San Francisco, California, 1992.

[11] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - the XML enabled data management system. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*, pages 561 –568, San Diego, California, 28 February - 3 March 2000.

[12] C. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-based information mediation with MIX. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of Data (SIGMOD'99)*, pages 597–599, Philadelphia, Pennsylvania, USA, June 1-3 1999. ACM Press.

[13] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal*, 8(2):101–119, 1999.

[14] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, March 2000.

[15] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (second edition) - W3C recommendation. Available at `http://www.w3.org/TR/2000/REC-xml-20001006`, October 6 2000.

[16] M. Bryan. The SGML centre - guidelines for using XML for electronic data interchange. Available at `http://www.xmledi-group.org/xmledigroup/guide.htm`, 1998. Personal Communication.

[17] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (SIGMOD'97)*, pages 117–121, 1997.

[18] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB'2000)*, pages 105–110, Dallas, Texas, USA, May 18-19 2000.

[19] R. G. G. Cattell. *The Object Database Standard: ODMG-93*. Morgan Kauffman Publishers, Inc., 1994.

[20] D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kauffman Publishers, Inc., 1998.

[21] D. Chamberlin, P. Fankhauser, M. Marchiori, and J. Robie. XML query requirements - W3C working draft. Available at `http://www.w3.org/TR/xmlquery-req`, August 15 2000.

[22] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB'2000)*, pages 53–62, Dallas, Texas, USA, May 18-19 2000.

[23] M. P. Consens and T. Milo. Algebras for querying text regions: Expressive power and optimization. *Journal of Computer and System Sciences*, 57(3):272–288, December 1998.

[24] C. J. Date and H. Darwen. *The SQL standard*. Addison Wesley, 4th edition, 1997.

[25] I. Dayen. Storing XML documents in relational databases. Available at `http://www.xml.com/pub/a/2001/06/20/databases.html` .

[26] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of The Eighth International World Wide Web Conference (WWW'8)*, Toronto, Canada, May 11-14 1999.

[27] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of Data (SIGMOD'99)*, Philadelphia, Pennsylvania, May 31 - June 3 1999.

[28] A. Deutsh, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *IEEE Data Engineering Bulletin*, 22(3), September 1999.

[29] D. Egnor and R. Lord. Structured information retrieval using XML. In *Proceedings of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, July 28 2000.

[30] A. Eisenberg and J. Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Record*, 28(1):131–138, 1999.

[31] Object design inc. an XML data server for publishing entreprise Web applications. White paper.

[32] D. C. Fallside. XML schema part 0: Primer - W3C candidate recommendation. Available at `http://www.w3.org/TR/xmlschema-0/`, October 24 2000.

[33] M. Fernadez, J. Siméon, and P. Wadler. XML query languages : Experiences and exemplars. Available at `http://www.research.att.com/~mff/files/exemplars.ps`, 2000.

[34] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the 9th International World Wide Web Conference (WWW'9)*, Amsterdam, The Netherlands, May 15 - 19 2000.

[35] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), September 1999.

[36] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *ACM SIGMOD Workshop on The Web and Databases (WebDB'99)*, pages 25–30, Philadelphia, Pennsylvania, USA, June 3-4 1999.

[37] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, Athens, Greece, August 25-29 1997.

[38] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. Inverted files. In W. B. Frakes and R. A. Baeza-Yates, editors, *Information Retrieval - Data Structures and Algorithms*, chapter 3, pages 28–43. Prentice Hall, 1992.

[39] IBM DB2 Universal Database XML Extender - administration and programming. Available at `http://www-4.ibm.com/software/data/db2/extenders/xmlext/`, 1999.

[40] C. Kanne and G. Moerkotte. Efficient storage of XML data. Unpublished manuscript, 1999.

[41] M. Klettke and H. Meyer. XML and object-relational database systems - enhancing structural mappings based on statistics. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 18-19 2000.

[42] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. In *Proceedings of the Eighth International Workshop on Database Programming Languages (DBPL'99)*, Kinloch Rannoch, Scotland, September 1999.

[43] D. Lee and W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, September 2000.

[44] R. Luk, A. Chan, T. Dillon, and H.V. Leong. A survey of search engines for XML documents. In *Proceedings of the ACM SIGIR 2000 Workshop on XML and Information Retrieval*, Athens, Greece, July 28 2000.

[45] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with XML and relational. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB'2000)*, pages 623–626, Cairo, Egypt, September 10-14 2000.

[46] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[47] B. Nixon, L. Chung, D. Lauzon, J. Mylopoulos, A. Borgida, and M. Stanley. Implementation of a compiler for a semantic data model: Experiences with Taxis. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*, pages 118–131, San Francisco, CA USA, May 27 - 29 1987.

[48] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 6-10 1995. IEEE Computer Society Press.

[49] POET software corporation - the POET XML repository, November 1998. White paper.

[50] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw - Hill, second edition, 2000.

[51] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). In *Proceedings of The W3C Query Languages Workshop (QL'98)*, Boston, Massachussets, USA, December 3-4 1998.

[52] A. Sahuguet. Kweelt, the making-of: Mistakes made and lessons learned. Technical Report MS-CIS-00-23, University of Pennsylvania, 2000.

[53] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 18-19 2000.

[54] H. Schöning and J. Wäsch. Tamino - an internet database system. In *Proceedings of the 7th International Conference on Extending Database Technology (EDBT'00)*, Konstanz, Germany, March 2000.

[55] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB'2000)*, pages 65–76, Cairo, Egypt, September 10-14 2000.

[56] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99)*, Edinburgh - Scotland, September 7-10 1999.

[57] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and retrieval of XML documents using object-relational databases. In *Proceedings of the 10 th International Conference on Database and Expert Systems Applications (DEXA'99)*, Florence, August 30 - September 3 1999.

[58] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[59] L.D. Stein and J. Thierry-Mieg. ACeDB: a genome database management system. *Computing in Science & Engineering*, 1(3):44 –52, May-June 1999.

[60] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In Timos Sellis and Sharad Mehrotra, editors, *Proceedings of the 2001 ACM SIGMOD International Conferenceo n Management of Data*, pages 413–424, Santa Barbara, CA, USA, May 21-24 2001.

[61] R. van Zwol, P. M.G. Apers, and A. N. Wilschut. Modelling and querying semistructured data with MOA. In *Proceedings of the Workshop on Semi Structured Data and Non-Standard Data Formats*, Jerualem, Israel, January 1999.

[62] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'98)*, Melbourne, Austrailia, August 24-28 1998.

[63] L. Wood, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, and C. Wilson. Document object model (DOM) level 1 specification (second edition) - version 1.0 - W3C working draft. Available at `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`, September 29 2000.