Chapter 12
Between Relational Data and XML

Data integration between “Legacy Systems” and XML databases

- Note: “legacy” now means SQL ...

Mixing everything up ...

Access to data stored in relational databases by

- using an XML environment (e.g., saxon) and mapping relational data from a remote SQL database (e.g. Oracle) to XML, and working with it.
- using XML-world concepts and languages in an SQL environment, e.g. for exchanging data in XML format (again, e.g., Oracle).

(Note that IBM DB2/XML Extender and MS SQL Server provide similar functionality)

12.1 Publishing/Mapping Relational Data in XML

Several *generic* mappings are possible:

Consider country(name: “Germany”, code: “D”, population: 83536115, area: 356910)

- tables, rows, and subelements

```
<country>
 <row><name>Germany</name><code>D</code>
      <population>83536115</population><area>356910</area></row>
    :
</country>
```

- tuples with subelements

```
<country><name>Germany</name><code>D</code>
         <population>83536115</population><area>356910</area></country>
    :
```

- analogous with XML attributes
- advantage with subelements: SQL values can also be object types (mapped to XML) and XMLType contents!

Example: HTTP-based access to Oracle

The whole database is mapped (virtually) to XML:

```
<SCHMIDT>   -- user name as
  <COUNTRY>  -- all names are capitalized
    <ROW><NAME>Germany</NAME><CODE>D</CODE>
      <POPULATION>83536115</POPULATION><AREA>356910</AREA></ROW>
    :
  </COUNTRY>
  :
</SCHMIDT>
```

Access by extended URL notation:

- URL: *computer*:*port*/oradb/*user*/*tablename*/ROW[*condition*]
- capitalize user, table and attribute names
- URL must select a single element (whole table, or single row)

```
ap34.ifi...:8080/oradb/DUMMY/COUNTRY              %% show page source
ap34.ifi...:8080/oradb/DUMMY/COUNTRY/ROW[CODE='D']
ap34.ifi...:8080/oradb/DUMMY/COUNTRY/ROW[CODE='D']/NAME
ap34.ifi...:8080/oradb/DUMMY/COUNTRY/ROW[CODE='D']/NAME/text()
```


Generic Mappings (Cont'd)

Up to now: mapping of materialized base tables.
Problem: how to map the result of a query with computed columns?

```
SELECT Name, Population/Area FROM country
```

- tables, rows, and subelements:
  the DTD is independent from the relational schema
  ("JDBC-style" processing of result sets)

```
<table name="country">
  <row><column name="name">Germany</column>
       <column name="code">D</column>
       <column name="population">83536115</column>
       <column name="area">356910</column>
  </row>
   :
</table>
```

- another "most generic mapping" as (object, property, value) to be discussed later ...

Additionally: often, tools define their own access functionality ...

ACCESS TO SQL DATABASES WITH SAXON-XSLT

- uses JDBC technology for remote access (at least for Java XSL tools)
- defines namespace "sql"
- <sql:connect> with attributes "database" (JDBC url), "driver" (JDBC driver)
  returns a JDBC connection object as a value of type "external object" that can be bound to a variable, e.g. $connection.
  Note: there can be several connections at the same time
- <sql:query> with following attributes allows to state an SQL query whose result is generically mapped to XML:
  - connection
  - table: ... the "FROM" clause
  - column: ... the "SELECT" clause
  - where: optional condition
  - row-tag: tag to be used for rows (default: "row")
  - col-tag: tag to be used for columns (default: "col")

  result is a collection of <row> ... </row> elements that can e.g. be bound to a variable.

Administrative Parameters

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:sql="java:/net.sf.saxon.sql.SQLElementFactory"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon sql">

<!-- insert your database details here, or supply them in parameters -->
<xsl:param name="driver" select="'oracle.jdbc.driver.OracleDriver'"/>
<xsl:param name="database" select="'jdbc:oracle:thin:USER/PASSWD@IPADDRESS:1521:USER'"/>
<xsl:param name="user" select="'USER'"/>
<xsl:param name="password" select="'PASSWD'"/>

<xsl:variable name="connection"
     as="java:java.lang.Object" xmlns:java="http://saxon.sf.net/java-type">
  <sql:connect driver="{$driver}" database="{$database}"
               user="{$user}" password="{$password}">
    <xsl:fallback>
      <xsl:message terminate="yes">SQL extensions not installed</xsl:message>
    </xsl:fallback>
  </sql:connect>
</xsl:variable>
</xsl:stylesheet>
```

[Filename: XSLT/sql-administrativa-fake.xsl]

Example Access/Query

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:sql="java:/net.sf.saxon.sql.SQLElementFactory"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon sql">
<xsl:include href="sql-administrativa.xsl"/>

<xsl:template match="*">
  <xsl:call-template name="countries"/>
</xsl:template>

<xsl:template name="countries">
 <xsl:variable name="country-table">
   <sql:query connection="$connection" table="country" column="*"/>
 </xsl:variable>
 <xsl:copy-of select="$country-table"/>
 <sql:close connection="$connection"/>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/sql-query.xsl]

uses a primitive mapping that relies on the order of columns.

Example Access/Query

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:sql="java:/net.sf.saxon.sql.SQLElementFactory"
  xmlns:saxon="http://saxon.sf.net/"
  extension-element-prefixes="saxon sql">
<xsl:include href="sql-administrativa.xsl"/>

<xsl:template match="*">
 <xsl:call-template name="countries"/>
</xsl:template>

<xsl:template name="countries">
 <xsl:variable name="country-table">
   <sql:query connection="$connection" table="country,encompasses"
       where="country.code=encompasses.country"
       column="country.name,encompasses.continent,percentage"
       row-tag="country" column-tag="bla"/>
 </xsl:variable>
 <xsl:copy-of select="$country-table"/>
 <sql:close connection="$connection"/>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/sql-query2.xsl]

Further Commands

- <sql:insert> with attribute
  - attribute: table="..."
  - children: <column name="..." select="..."/>
    value can also be given as contents of the column element; currently always interpreted as a string.
- <sql:close> with a "connection" attribute

See also `http://www.saxonica.com/documentation/documentation.html`

Similar: http-based access to a relational database, e.g.
http://*url*/*username*/country/row[name='Germany']/population

12.2 The SQL/XML or SQLX standard

Coexistence between Relational Data and XML

- mapping relational data to XML (previous section)
- store large quantities of data relationally
  often, "the" XML presentation is just a view over relational data
  ("XML publishing")
- store XML data in relational systems
  - data-centric XML: efficient, several possibilities
  - document-centric XML: problematic

SQL/XML

- published by ISO in 2003: `www.sqlx.org`
- an SQL datatype “XMLType” for storing XML as “value” in databases:
  “SQL/XML includes an XML datatype for use inside SQL,” says Melton. “This allows the SQL engine to automatically ’shred’ data-oriented XML documents for storing some elements and contents in one place while keeping other elements in other places. Using indexes and joins, you can then bring this data back together very quickly.”
  - XML: shredding or storing as LOB (Large Object)
  - ... can then be exported to XML tools, e.g. by JDBC ...
  - ... or handled with special functions that are embedded in SQL.

Experiments with Oracle

- Oracle TechNet is often a valuable source for information:
  `http://otn.oracle.com/oramag/oracle/03-may/o33xml.html`
- 1/2 day Tutorial:
  `http://otn.oracle.com/tech/xml/xquery/pdf/allyourxml%20-%20xml%202003.pdf`
- Note: slight differences to SQL/XML Standard

SQL/XML

Making XML data a first-class citizen in relational databases,
seamless flow from relational data to XML and back

SQL to XML

- XML generation from SQL queries (i.e., in the SELECT clause)
  (e.g., as packets for data exchange)
- Define XML Type views over SQL data

SQL access and manipulation of XML inside the RDB

... use XPath expressions inside SQL (and rise the question what is actually the difference to XQuery)

- storing XML in RDB (e.g. if XML-data exchange packets came in)
- XPath-based extraction of XML content (SELECT clause)
- XPath-based query of XML content (WHERE clause)
- XPath-based update of XML content (in SQL)
- Define XPath-based relational views over XML content

"XML" as an SQL Datatype

XML/XMLType: an *SQL* datatype to hold XML data:

```
CREATE TABLE Mondial OF XMLType;    use as Row Object Value
CREATE TABLE CountryXML OF XMLType;    use as Row Object Values
CREATE TABLE cityXML (name VARCHAR2(32), population XMLType, coordinates XMLType);
   use as Column Object Values
```

- generation: INSERT INTO *table* VALUES (... , XMLType('*XML as ASCII*') ...)
- storage: chosen by the database
  - "shredding" and distributing over suitable tables (of object-relational object types) (queries are translated into SQL joins/dereferencing)
  - storing it as VARCHAR, CLOB (Character Large Object), or as separate file (the remainder of this section uses CLOB)
  - storing it "natively"
- query it by XPath
- export/exchange in ASCII

Importing XML Files

- graphical interface: copy XML document into database "directory"
- from files:
  - tell Oracle where it finds XML files (Admin):

    ```
    CREATE OR REPLACE DIRECTORY XMLDIR AS '/db' ;  -- on ap34 for global things
    CREATE OR REPLACE DIRECTORY XMLDIR AS '/home/bla/...' ;
        -- if you want to use your own one
    ```

    note: SQL directory object "XMLDIR" is now known.
    You can make it unknown by `DROP directory XMLDIR`.
  - copy XML file (e.g. m.xml) into XMLDIR
    * Note that the XML file must not contain a reference to a DTD!
    * note that the file must be publicly readable – `chmod` *filename* `644`
  - load the file into the database (use the xdb_utilities package)

    ```
    INSERT INTO mondial VALUES(xdb_utilities.getXMLfromfile('m.xml','XMLDIR'));
    set long 10000 ;   -- number of characters in the output
    SELECT * FROM mondial;
    ```

[Note: HTTP-based access in Oracle does currently not work inside XMLType]

SQL/XML: Generating XML by XML Publishing Functions

The SQL/XML Standard defines the following “XML publishing functions”:

- XMLElement: a function to transform a relational value into an XML element, in the form <elementName>value</elementName>
- XMLAttributes: a function used to place attributes in XML elements returned by SQL queries
- XMLForest: a function to generate a list, called a ”forest,” of XML elements from a list of relational values
- XMLConcat: a function to concatenate two or more XML values (more general than XMLForest)
- XMLAgg: a function to group, or aggregate, XML data in GROUP BY queries
- XMLNamespaces: a function to declare namespaces in an XML element
- XMLSerialize: a function to serialize an XML value as a character string

Constructing XML

Basic form: XMLElement

```
SELECT XMLElement("Country", XMLAttributes(code AS car_code),
          XMLElement("Name",name),
          XMLElement("Population",population),
          XMLElement("Area",area))
FROM country
WHERE area > 1000000
```

```
SELECT XMLElement("City", XMLAttributes(country AS country),
          XMLElement("Name",name),
          CASE WHEN longitude IS NULL THEN NULL
             ELSE XMLElement("Longitude",longitude) END,
          CASE WHEN latitude IS NULL THEN NULL
             ELSE XMLElement("Latitude",latitude) END)
FROM city
WHERE country='D';
```

- Note: a *functional* when!

CONSTRUCTING XML

XMLForest: short form for the same things

```
SELECT XMLElement("Country", XMLAttributes(code AS car_code),
          XMLForest(name AS "Name",
                    population AS "Population",
                    area AS "Area"))
FROM country
WHERE area > 1000000
```

XMLConcat – construct from a sequence of independently constructed XML:

```
SELECT XMLElement("Country", XMLAttributes(code AS car_code),
          XMLConcat(XMLElement("Name",name),
                    XMLElement("NoOfCities", (SELECT count(*)
                                              FROM City
                                              WHERE country=country.code))))
FROM country
WHERE area > 1000000
```


CONSTRUCTING XML

Grouping/Aggregation: Nested Lists

... and now we can have the number of cities in a country, together with a list of them:

```
SELECT XMLElement("Country", XMLAttributes(country AS car_code),
          XMLConcat(XMLElement("NoOfCities", count(*)),
                    XMLAgg(XMLElement("city",name)
                           ORDER by population)))
FROM city
GROUP BY country
```

Constructed XML can then be used for filling tables:

Filling a table ... with XML Row Values

```
CREATE TABLE CountryXML OF XMLType;

INSERT INTO countryXML
(SELECT XMLElement("Country", XMLAttributes(code AS Code),
   XMLConcat(XMLElement("Name",name),
             (SELECT XMLElement("Capital",
                      XMLForest(name AS "Name",
                                population AS "Population"))
               FROM city
               WHERE country=country.code and city.name=capital)))
FROM country);
```


Filling a table: XML Column Values

```
CREATE TABLE cityXML (name VARCHAR2(32),
     population XMLType, coordinates XMLType);

INSERT INTO cityXML
(SELECT name,
  XMLElement("population", XMLAttributes(95 as year), population),
  CASE WHEN longitude IS NULL THEN NULL
   ELSE XMLElement("Coordinates",
                XMLConcat(XMLElement("Longitude", longitude),
                          XMLElement("Latitude", latitude))) END
 FROM city
 WHERE country='D');
```

HANDLING XML DATA FROM WITHIN SQL

- XMLType is defined as an abstract datatype:
- signature:

  extract: XMLType × XPath_Expression → XMLType/text/number
  extractValue: XMLType × XPath_Expression → text/number
  existsNode: XMLType × XPath_Expression → Boolean
- implementation based on user-defined object types
  - cf. object-relational extensions to SQL
  - internally using several storage variants:
    VARCHAR, CLOB, external files
- above operations also available as member methods

```
SELECT extract(value(m), '//city[name="Berlin"]') FROM mondial m;
SELECT m.extract('//city[name="Berlin"]') FROM mondial m;
SELECT m.extractValue('//country[@car_code="D"]/population')
FROM mondial m;   -- buggy!!!
```


SELECT: "Extract" Function

extract(XMLType_instance, XPath_string) XMLType_instance.extract(XPath_string)

First argument: selects an attribute with value of type "XMLType" in the current row (use value(.) function)
Second argument: applies XPath_string to it
Result: value of type XMLType or any other SQL type
(multi-valued results are concatenated)

XML Row Values

Value of the row is of type XMLType: apply methods directly

```
SELECT extract(value(c), '/Country/@CODE'),
       extract(value(c), '/Country/Capital/Name')
FROM countryXML c

SELECT c.extract('/Country/@CODE'),
       c.extract('/Country/Capital/Name')
FROM countryXML c
```

SELECT: "Extract" Function (Cont'd)

XML Column Values

Recall:

```
CREATE TABLE cityXML (name VARCHAR2(32),
    population XMLType, coordinates XMLType);
```

cityXML.population is an XMLType column object:

```
SELECT extract(population,'/') FROM cityXML;
SELECT c.population.extract('/') FROM cityXML c;
SELECT name, extractValue(population,'/population/@YEAR'),
       extractValue(population,'/population')
FROM cityXML;
SELECT name, c.population.extract('/population/@YEAR').getNumberVal(),
       c.population.extract('/population/text()').getNumberVal()
FROM cityXML c
ORDER BY 3;
```

- exact capitalization in XPath argument!
- extractValue currently not implemented as member method (bug)
- use getNumberVal() and getStringVal() functions

WHERE: "ExistsNode" Function

existsNode(XMLType_instance, XPath_string)

Checks if item is of XMLType_instance, and XPath_string has a nonempty result set:

```
SELECT name, extractValue(population,'/')
FROM cityXML
WHERE existsNode(population,'/population[text()>1000000]') = 1;

SELECT name, extractValue(population,'/')
FROM cityXML c
WHERE c.population.existsNode('/population[text()>1000000]') = 1;
```

Updating XML Contents

- the complete XMLType value is *changed*, not updated
- updateXML(...) as a (transformation) function!
  Note: the statement "SELECT updateXML(...) FROM ..." does not update the DB, but returns the value that would result from the update.

```
UPDATE cityXML c
SET c.population   -- an XMLType element
    = updateXML(c.population,
                '/population/text()','1000000',
                '/population/@YEAR','2004')
WHERE name='Gottingen'

UPDATE countryXML c
SET value(c) = updateXML(value(c),
                '/Country/Name/text()','Fidschi')
WHERE existsNode(value(c),'/Country[Name="Fiji"]') = 1
```


12.3 Background: XMLType as Object Type

(cf. "Practical Training in SQL" course)

Since SQL3 Standard: Object(-relational) types

- user-definable: CREATE TYPE AS OBJECT ... / CREATE TYPE BODY
- stored as *row* or *column objects*
  CREATE TABLE cities OF CityObjectType;
- *member methods*
  - programmed in PL/SQL (or also in Java)
  - calls are embedded into SQL: SELECT *object.method(args)*
- reference attributes:
  CREATE TABLE COUNTRY (..., capital REF CityType, ...);
  SELECT c.capital ...;

⇒ now used for implementing XMLType

- as predefined internal classes/types
- can be used high-level from SQL, or low-level inside PL/SQL

XSLT IN ORACLE: "TRANSFORM" MEMBER METHOD

Member Method of XMLType: *XML-instance*.transform(*Stylesheet-as-XMLValue*)
as SQL function: SELECT XMLTransform(*XML-instance*,*Stylesheet-as-XMLValue*)

```
CREATE TABLE stylesheets
 (name VARCHAR2(100),
  stylesheet XMLTYPE);

INSERT INTO stylesheets
VALUES('mondial-simple.xsl',
        xdb_utilities.getXMLfromfile('mondial-simple.xsl','XMLDIR'));

SELECT value(m).transform(s.stylesheet)
FROM mondial m, stylesheets s
WHERE s.name = 'mondial-simple.xsl';

SELECT XMLTransform(value(m),s.stylesheet)
FROM mondial m, stylesheets s
WHERE s.name = 'mondial-simple.xsl';
```


Using built-in DOM, Parser, and XSL Engine

Tools from several packages can be explicitly used inside PL/SQL procedures:

- dbms_xmldom: implements DOM (usually, XML is transformed into DOM for processing it in detail)
  PL/SQL call: dbms_xmldom.*dosomething*(*object,args*)
- dbms_xmlparser: parses documents from CLOB or URL, parses DTD from CLOB or URL (and stores the result);
  access to the DOM instance/DTD in the parser then by "getdocument" or "getdoctype"
- dbms_xslprocessor: processxsl(*different arguments*);
  clob2file/file2clob allows for reading/writing;
  selectnodes/selectsinglenode/valueof: XPath queries

... for details: Oracle Documentation, google ...

```
CREATE OR REPLACE FUNCTION xslexample RETURN CLOB IS
  xmldoc       CLOB;
  xsldoc       CLOB;
  myParser     dbms_xmlparser.Parser;
  indomdoc     dbms_xmldom.domdocument;
  xsltdomdoc   dbms_xmldom.domdocument;
  xsl          dbms_xslprocessor.stylesheet;
  outdomdocf   dbms_xmldom.domdocumentfragment;
  outnode      dbms_xmldom.domnode;
  proc         dbms_xslprocessor.processor;
  html         CLOB;
BEGIN
  -- Get the XML document as CLOB
  SELECT value(m).getClobVal() INTO xmldoc FROM mondial m;
  -- Get the XSL Stylesheet as CLOB
  SELECT s.stylesheet.getClobVal() INTO xsldoc
  FROM stylesheets s WHERE name='mondial-simple.xsl';

  -- Get the new xml parser instance
  myParser := dbms_xmlparser.newParser;
  -- Parse the XML document and get its DOM
  dbms_xmlparser.parseClob(myParser, xmldoc);
  indomdoc := dbms_xmlparser.getDocument(myParser);

  -- Parse the XSL document and get its DOM
  dbms_xmlparser.parseClob(myParser, xsldoc);
  xsltdomdoc := dbms_xmlparser.getDocument(myParser);

  xsl  := dbms_xslprocessor.newstylesheet(xsltdomdoc, '');
  -- Get the new xsl processor instance
  proc := dbms_xslprocessor.newProcessor;

  -- Apply stylesheet to DOM document
  outdomdocf := dbms_xslprocessor.processxsl(proc, xsl, indomdoc);
  outnode    := dbms_xmldom.makenode(outdomdocf);

  -- Write the transformed output to the CLOB
  dbms_xmldom.writetoCLOB(outnode, html);
  -- Return the transformed output
  return(html);
END;
/
```

[Filename: OraXML/xslexample.sql]

12.4 Storing XML Data in Database Systems

- "shredding" and distributing over suitable tables (of object-relational object types) (queries are translated into SQL joins/dereferencing)
  - Schema-based
  - Generic mapping of the graph structure
- storing it as VARCHAR, CLOB (Character Large Object), or as separate file with special functionality
- storing it "natively"/model-based: internal object model

Literature

Klettke/Meyer "XML & Datenbanken" (dpunkt-Verlag), Ch. 8
Schöning: "XML und Datenbanken" (Hanser), Ch. 7,8
Chaudhri/Rashid/Zicari: XML Data Management

12.4.1 Mapping XML $\rightarrow$ Relational Model

two basic approaches:

- Schema-based: one or more "customized" tables for each element type
  ($\rightarrow$ similar to relational normalization theory)
  - (possibly) many null values
  - efficient access on data that belongs together
- one generic large table based on the graph structure:
  (element-id, name of the property, value/id of the property)
  - no null values
  - although memory-consuming (keys/names that are stored once in (1) are now stored for each occurrence)
  - data that belongs together is split over several tupels

$\Rightarrow$ in both cases, theory and efficiency of relational database systems can be exploited.

SCHEMA-BASED STORAGE

necessary: DTD or XML Schema of the instance.

1. For each element type that has children or attributes, define a table that contains
   - a column that holds the primary key of the parent,
   - a primary key column if the element type has a member that satisfies (1) or (2),
   - for each scalar attribute and child element type with text-only contents that appears at most once, a column that holds the text contents.
2. for each multi-valued attribute or text-only subelement type that occurs more than once for some element type, a separate table is created with the following columns:
   - key of the parent node,
   - the (attribute or text) value

   (similar to 1:n relationships in the relational model).

- for mixed content: possible solutions depend on the specific structure
- special treatment for infrequent properties (to avoid nulls): handling in a separate XMLType column that holds all these properties together.

Schema-Based Storage: Example

For Mondial countries, provinces and cities, the following relations are created:

- country: key(mondial), key, name, code, population, area, ...
- border: ref(country), ref(other country), length
- language: ref(country), language, percentage
- province: ref(country), key, name, population, area
- city: ref(country), ref(province), key, name, longitude, latitude
- city-population: ref(city), year, value

Exercise

- give an excerpt of the instance
- translate some XPath/XQuery queries to SQL
- extended exercise: generate and populate the schema in SQL

Supported: Oracle (with augmented XML Schema), IBM DB 2 (with DAD – Data Access Definition), MS SQL Server (extended Data-Reduced XML)

OBJECT-RELATIONAL APPROACH: INTERNAL OBJECT TYPES

- “shredded storage” of XML data is in general not implemented by plain relational tables, but using object-relational technology:
  object types, collections, varrays etc ...
- collections/varrays: with value- and path indexes
- XPath expressions are rewritten into these structures

Integration of “legacy” object types

- application-dependent object types (as used in SQL3 in pre-XML times): standard mapping to XML (e.g. for data exchange)

Oracle & XML Schema

... register XMLSchema (must be typed in one single line!)

```
EXEC dbms_xmlschema.registerURI('http://mondial.de/m.xsd',
         'http://dbis.informatik.uni-goettingen.de/Mondial/mondial.xsd')
```

can be deleted with

```
EXEC dbms_xmlschema.deleteSchema('http://mondial.de/m.xsd',
      dbms_xmlschema.DELETE_CASCADE_FORCE)
```

- ... now, it knows http://mondial.de/m.xsd and created object tables for all root element types:
  SELECT * from ALL_XML_TABLES;

```
CREATE TABLE mondial2 OF XMLType
 XMLTYPE STORE AS OBJECT RELATIONAL
 XMLSCHEMA "http://mondial.de/m.xsd"
 ELEMENT "mondial"
```

```
INSERT INTO mondial2 VALUES(xdb_utilities.getXMLfromfile('m.xml','XMLDIR'));
SELECT XMLisValid(value(m)) FROM mondial2 m;
```


GRAPH-STRUCTURE-BASED STORAGE

Without any schema knowledge, the graph structure can be represented in a single large table:

NodeNumber	ParentNode	[SiblingNo if ordered]	Name	Value

(see next page)

Alternatives

- separate table for elements and attributes (without node number and sibling number)
- separate between no-value, string value and numeric values for storing adequate types.
- previous-sibling and following-sibling columns instead of sibling-no (DOM style)

Querying

- requires recursive queries (PL/SQL; CONNECT BY)
- large joins (using the same large table several times)
- not implemented in any commercial system [according to Schöning 2003]

NodeNumber	ParentNode	SiblingNo	Name	Value
1	doc	1	mondial	
2	1	1	country	
3	2		@code	D
4	2		@membership	ref(eu)
:	:		:	:
41	2		@membership	ref(un)
42	2		@area	356910
43	2		@capital	ref(92)
44	2	1	name	Germany
45	2	2	population	83536115
:	:		:	:
90	2	47	province	
91	90	1	name	Berlin
92	90	2	city	
93	92		@country	ref(2)
94	92	1	name	Berlin
95	92	2	population	
96	95		@year	1995
97	95		text()	3472009
:	:		:	:

12.4.2 “Opaque” Storage

XML documents are stored as a whole as special datatype that can be used as row type or column data type (most commercial DBS; as described above for SQL/XML)

- approaches with text-based storage (CLOBs, files)
- specialized functionality for this datatype
  (cf. object-relational DBs: member functions)
  - XPath querying, XSLT support
  - validation
  - text search functions
- syntax embedded into SQL
- supported by indexes
  - full text indexes
  - path indexes/ “functional” indexes (user-defined, e.g. over //city/@country)
  - application and refinement of classical algorithms
- optimization of queries below the relational level!

12.4.3 “Native” Storage

Using “original” concepts of the database for storing XML (internal XML or object model) instead of mapping it or “simply” representing it as ASCII.

- often based on existing object-oriented DB-systems with application of concepts from hierarchical and network-DBs
- no document transformation to another data model
- data model/classes based on the notions of “tree”, “element”, “attribute”, “document order”
- navigation
- XPath/XQuery/XSQL APIs

“Native” Storage: Systems and Products

Many early implementations came from the object-oriented area:

- XML-QL, based on the Strudel system
- LoPiX, based on F-Logic
- Lorel-XML, based on Lorel/OEM
- Tamino (Software AG, Darmstadt, founded 1969 (Adabas, hierarchical DB), Tamino 1999, with XQL, first native XML DBMS),
- Excelon (until 1998: ObjectDesign with “ObjectStore”; since 12.2002: acquired by Progress Software Corp.)
- POET (Hamburg, “Persistent Objects and Extended Database Technology”, product 1990, spin-off from BKS 1992, OQL interfaces, SGML Document Repository 1997, Content Management Suite since 1998, merger with Versant 2004)
- Infonyte (GMD IPSI XQL 1998, based on a “Persistent DOM”, 12.2000: spinoff TU Darmstadt/Fraunhofer-IPSI)

GENERIC DATABASE BEHAVIOR FOR XML DATABASES

Everything that has been developed and dicussed for relational databases is also relevant for XML:

- physical storage + storage management
- optimization, evaluation algorithms
- multiuser operation, transactions (ACID), safety, access control
- ECA-rules, triggers

The algorithms and theoretical foundations are very similar.

Often, relational (or hierarchical) DB technology is actually used inside.

COEXISTENCE OF XML AND RELATIONAL DATA

- generating XML (views, data exchange packets, ...) from stored relational data
- relational (and object-relational) techniques used for efficiently storing data-centric XML
- storing text-oriented data in RDB with specialized "native" datatypes
- XPath is also accepted by SQL/XML
- additional XML processing functionality by packages and object types
- XQuery is still not the "winner" for data-oriented applications!
- is it the winner for document-oriented applications?
  very new: 9.7.2004: `http://www.w3.org/TR/xquery-full-text/`