In various application domains, the data are semi-structured; the database schema is loose-defined. 
Semi-structured data need specialized management methods. 
The most characteristic example is XML data where the elements (tags) define the semantics of the information.

XML (like HTML) is a subset of SGML 
In HTML the tags serve the primary purpose of describing how to display a data item. 
On the other hand, XML tags describe the data itself. Tags are called elements in XML. 
This means that a program receiving an XML document can interpret it in multiple ways, can filter the document upon its content, can restructure it for a different application, etc.

XML is becoming a standard for information exchange over the internet. 
Since actual data are stored in XML documents, it should be possible to query the data. 
Here comes the role of databases: How should we organize and query XML data?

Solution 1:
- Use specialized storage methods, query languages and query evaluation techniques for semi-structured data.

Solution 2:
- Represent XML data in relational tables, transform queries to SQL, and use the mature relational DB technology.
More on XML – Document Type Descriptors

- DTDs (Document Type Descriptors) define (and control) the schema of XML documents for a specific application.
- Thus, now the structure of these documents is not free, but should conform to the DTD.
- The DTD can help define a relational schema for the class of documents that conform to it.

Example of a DTD

```xml
<!ELEMENT book (booktitle, author)>  
<!ELEMENT article (title, author*, contactauthor)>  
<!ELEMENT contactauthor EMPTY>  
<!ATTLIST contactauthor authorID IDREF IMPLIED>  
<!ELEMENT monograph (title, author, editor)>  
<!ELEMENT editor monograph*>  
<!ATTLIST editor name CDATA #REQUIRED>  
<!ELEMENT author (name, address)>  
<!ATTLIST author id ID #REQUIRED>  
<!ELEMENT name (firstname?, lastname)>  
<!ELEMENT firstname (#PCDATA)>  
<!ELEMENT lastname (#PCDATA)>  
<!ELEMENT address ANY>  
```

More on XML – Queries

- Queries on XML data are described by the structural relationships of elements, attributes and values.
- Several XML Query Languages have been proposed.
- XML-QL, Lorel, UnQL, XQL, XPath, XQuery, ...

More on XML – Query Example

- Find the last name of the author of book “the selfish gene”
  - WHERE <book>
    <booktitle> The Selfish Gene </booktitle>
    <author> <lastname> $l </lastname> </>
  </>
  IN db.xml
  - CONSTRUCT <lastname> $l </lastname>

XML data represented as graphs

- An XML document can be represented as a node-labeled graph.
- The labels of the graph are element tags, attribute names and values.
- Most documents can be represented by trees. The edges that transform a tree to a graph come from ID references.

Example of a tree representation

```
book
  └── booktitle
      └── The Selfish Gene

author
  └── id
      └── dawkins

address
  └── city
      └── Timbuktu

name
  └── lastname
      └── Richard

address
  └── zip
      └── 99999
```
Example of a graph representation

XML Query types

- Queries with *absolute* path expressions. These queries retrieve paths where the first element is the ROOT of the document.
- Example: find all books written by author with lastname="Smith"
  - book/author/name/lastname/Smith

- Queries with *simple* path expressions. These queries retrieve paths where the first element can be any element of the document.
- Example: find all items written by author with lastname="Smith"
  - //author/name/lastname/Smith

- Queries with *regular* path expressions. These queries retrieve paths where not all elements on the path are specified.
- Example: find all documents with an "author" element with a descendant "Smith" in the graph
  - //author//Smith

- In general, other symbols may be used to denote the distance between the path elements.
- Example: find all documents with an "author" followed by one element, then one or none elements, and then by "Smith".
  - //author//Smith

Example of a graph representation (cont’d)
XML Query types

- Queries that match multiple regular path expressions. The paths are joined in a root element and the whole query is represented by a twig (small tree).
- Example: find the book with title “XML” written by an author with a descendant “Smith” in the graph.
  - book[/title/XML][//author//Smith]

Indexing and XML Query Processing

- Several storage schemes and indexes have been proposed for the queries discussed above.
- Some of them index the paths or subgraphs of the XML structures.
- Some decompose the information and flatten it into relational DB tables.

Path indexes for XML data

- If many documents exist, they are connected into a large graph by adding a common root.
- Then a structural summary of the XML graph is created.
- All the paths in the data graph are preserved into the summary graph.
- If we keep pointers to the original graph into the summary graph, then this becomes an index.

Path index example

A. a graph of documents

B. the 1-index

1. The 1-index maintains information about all paths in the original graph.

Path indexes for XML data

- The 1-index maintains information about all paths in the original graph.
- This makes the index very large (with size comparable to the data size).
- Therefore it is quite expensive to evaluate queries using this index.
- To address this problem an A(k)-index is proposed which indexes exactly only paths up to length k.
Bisimilarity

- Two nodes \( u, v \) are called **bisimilar** if:
  - They have the same label.
  - If \( u' \) is the parent of \( u \), then there is a parent \( v' \) of \( v \), such that \( u', v' \) are also bisimilar, and vice versa.

Bisimilarity Example

- Nodes 4 and 10 are bisimilar
- Nodes 10 and 15 are not bisimilar

Bisimilarity defines the 1-index

- Bisimilar nodes are stored in the same node in the summary index.

Bisimilarity and the A(\( k \)) index

- In the A(\( k \)) index, the notion of k-bisimilarity is used:
  - Two nodes \( u, v \) are 0-bisimilar, if they have the same label.
  - Two nodes \( u, v \) are k-bisimilar, if they have the same label and for every parent \( u' \) of \( u \), there is a parent \( v' \) of \( v \), such that \( u' \) and \( v' \) are (k-1)-bisimilar, and vice versa.

k-bisimilarity Example

- Nodes 5 and 16 are 1-bisimilar
- Nodes 6 and 15 are not 1-bisimilar

Bisimilarity and the A(\( k \)) index

- The A(\( k \))-index stores exactly all paths of length \( k \), or else: all k-bisimilar nodes in the data graph are stored in the same node in the index graph.
- This means that all incoming paths up to length \( k \) are encoded in the index.
A(k)-index example

**A(3) and A(2)-index**

```
  1 alldocuments (root)
     2,8 book
       3,9 title
         4,6,10 author
       name 5,7,11 address
     13 article
       14 title
         15,18 author
       name 16,19 address
```

**A(1)-index**

```
  1 alldocuments (root)
     2,8 book
       3,9 title
         4,6,10 author
       name 5,7,11,16,19 address
     13 article
       14 title
         15,18 author
       name 16,19 address
```

**A(0)-index**

```
  1 alldocuments (root)
     2,8 book
       3,9 title
         4,6,10 author
       name 5,7,11,16,19 address
     13 article
       14 title
         15,18 author
       name 16,19 address
```

Using the A(k) index to search

- A Label Map is constructed together with the index, where each label points to its positions in the index.

Evaluation of path queries

- Assume that a path query q of length \( \leq k \) is applied.
- The A(k)-index can answer the query as follows.
- First the last label of q is found and the label map is used to find its positions in the index.
- Then the index is traversed backwards to complete the answer.

Evaluation of path queries (example)

- Query book/title

Evaluation of path queries

- If the path of the query is longer than k, we may need to access the actual data.
- Thus, A(k) index alone cannot be used to answer the query in this case.
- This is because if we traverse the index backwards we may find false positive paths that actually do not exist in the graph.
- Paths that share information are grouped to decrease the potential cost.
Evaluation of path queries (2nd example)

- Query book/author/name
- path has length 2>1

A(1) index

<table>
<thead>
<tr>
<th>book</th>
<th>article</th>
<th>title</th>
<th>author</th>
<th>name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3,9</td>
<td>10,13</td>
<td>14</td>
<td>12,17</td>
</tr>
<tr>
<td></td>
<td>Alldocuments (root)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problems of path indexes

- They are appropriate only for simple path queries up to a certain length.
- Therefore if a query has branches or regular path expressions the index cannot provide exact answers, but the actual data have to be accessed.
- Also these indexes have high storage and update cost.

Storing and indexing XML data in relational databases

- We can decompose the structural information into tables and use them to answer queries.
- This reduces the volume of data that need to be accessed for a single query and we can use off-the-shelf query processing and optimization techniques.
- On the other hand, we may need expensive joins during query processing.

A decomposition model for XML data

- The storage model indexes the elements and text of the documents by their position in the graph.
- If the structures are trees, this representation can help to answer queries fast.
- On the other hand, for graphs the positions of the elements many times cannot help fast query evaluation because of recursion and other problems the incur.

Encoding elements, attributes and values based on their positions.

- The position of each element/attribute occurrence is represented as a 3-tuple (Document-id, StartPos:EndPos, LevelNum).
- Values (text) is encoded using (Document-id, StartPos, LevelNum):
  - Document-id is the id of the document that contains the element
  - StartPos is the number of words from the beginning of the document until the start of the element
  - EndPos is the number of words from the beginning of the document until the end of the element
  - LevelNum is the nesting depth of the element

Encoding example

```xml
<book>
  <booktitle>The Selfish Gene</booktitle>
  <author id="dawkings">
    <name>
      <firstname>Richard</firstname>
      <lastname>Dawkins</lastname>
    </name>
    <address>
      <city>Timbuktu</city>
      <zip>99999</zip>
    </address>
  </author>
</book>
```
Using the encoding to determine a structural relationship

- We can use the encoding to find fast the relationship between two elements (or between an element and a value).
- Element $e_1$ is an ancestor of element $e_2$ in the same document iff:
  - $e_1.$DocumentId = $e_2.$DocumentId
  - $e_1.$StartPos > $e_2.$StartPos && $e_1.$EndPos < $e_2.$EndPos (interval coverage)
- If the above hold and, in addition, $e_1.$LevelNum+1 = $e_2.$LevelNum, then $e_1$ is the parent of $e_2$.

Answering queries using the encoding

- Assume that all documents have been flattened to tables and the encoding is used to index the positions of each element and value in the documents.
- We store all information in a table:
  (ElementId, Document-id, StartPos:EndPos, LevelNum)
- The table is clustered by ElementId and sorted by (Document-id, StartPos).

Answering queries using the encoding (cont’d)

- The query is broken into binary parent-child or ancestor descendant relationships.
- Example:
  - book/title/XML[//author//Smith]
  - Broken to:
    - book/title
    - title/XML
    - book/author
    - author//Smith

Answering queries using the encoding (cont’d)

- Each binary query is executed as a join, and their results are “stitched” together to formulate the results of the whole query.
- Example: book/author/address
  - book/author: (2,4),(2,6) (8,10)
  - author/address: (10,12),(15,17)
  - book/author/title: (8,10,12)
How to process the binary joins

Thus the "heart" of XML query processing is the algorithm that joins the elements table to retrieve the results for each individual query component.

One method to process the binary join is to apply a merge join algorithm, since the table is already sorted by Element,DocId,StartPos.

Assume that the query is an A//D, where A is the ancestor element and D is the descendant element.

Worst case for the tree-merge join algorithm

The tree-merge join algorithm

The tree-merge join algorithm may perform many passes to the "inner" DList table, one for each element in AList that matches the elements there.

In order to avoid this a stack-tree join algorithm is proposed.

OBSERVATION: We can get all the join results by a depth-first traversal of the XML tree.

The stack-tree join algorithm

The lists are merged together as before, but a stack is maintained to keep nested AList elements which are in the same path as the current element from DList.

When a qualifying element in DList is found, all elements of AList in the stack are output.

Stack-tree join example

Dr. N. Mamoulis Advanced Database Technologies 53
Comments on the stack-tree join algorithm

- The algorithm has better worst case complexity than the tree merge join algorithm.
- Both of them have two versions; one that outputs results sorted on AList elements and one that outputs results sorted on DList elements.

Limitation of the binary join algorithms

- They are used only for binary joins. If a query is complex and contains many binary relationships, many intermediate results have to be merged.

Example:

- book[/title/XML][//author//Smith]
- Broken to:
  - book/title
  - title/XML
  - book//author
  - author//Smith

An extension of the stack-tree join algorithm

- The path join and twig join algorithms extend the basic stack join algorithm for complex queries.
- The idea is the same, but multiple stacks are used to avoid merging the intermediate results.
- Path join is appropriate for path queries only (e.g., book/author/name)
- Twig join is appropriate for branching (tree) expressions.

Example of Path-Join

```
Query a/b/c

...at point c3
```

```
StackA
  a1 b1 c1
  a2 b2 c2

StackB
  a1 b3 c3
  a3 b4 c4
```

```
Output
  c3,b4,a3
  c3,b4,a1
  c3,b3,a1
  c1,b2,a1
  c2,b2,a1
```

```
c3,b3,a3 is not an answer because b3 points to a1 in the next stack!
```

Path-join has optimal asymptotic cost for single-path queries, but ...

... if a query is a twig of multiple paths may produce many partial results which have then to be joined.

The twig-join, joins these results at production time.

Example:

- query a[[b/c]][[d/e]]
- two paths a/b/c
d- two paths a/d/e
- only one twig a[b/c][d/e]

Twig Join

- The twig join applies path join at multiple paths at the same time.
- When at some node there are potential solutions for each path of the query, the algorithm waits for these results and waits for them to be computed.
- Then the results from each path are joined.

partial result: a/b/c
merge result: a[b/c][d/e]
Limitations of the twig-join and stack-based methods

- It is useful for simple twigs only, but it is not trivial to extend it for arbitrary trees.
- The encoding can be used for tree-structured XML data only. However, in many cases XML data are graphs. In this case the encoding (and also the stack-based algorithms) are not applicable.

Summary

- XML data are everywhere today, and efficient management and querying systems are needed.
- This is why today XML data management is one of the hottest research topics in DB.
- There are two streams for XML data management:
  - Store XML data into native systems and use special indexing and querying methods.
  - Transform XML data into relational tables and use/adapt relational query algorithms.

References