CSE@UTA

Query Processing of XML Data

Leonidas Fegaras
University of Texas at Arlington

Traditional DB Applications

Characteristics:

- Typically business oriented
- Large amount of data
- Data is well-structured, normalized, with predefined schema
- Large number of concurrent users (transactions)
- Simple data, simple queries, and simple updates
- Typically update intensive
- Small transactions
- High performance, high availability, scalability
- Data integrity and security are of major importance
- Good administrative support, nice GUIs

CSE@UTA

Internet Applications

Internet applications

- use heterogeneous, complex, hierarchical, fast-evolving, unstructured/semistructured data
- access mostly read-only data
- require 100% availability
- manage millions of users world-wide
- have high-performance requirements
- are concerned with security (encryption)
- like to customize data in a personalized manner
- expect to gain user's trust for business-to-consumer transactions.

Internet users choose speed and availability over correctness

Examples of Applications

- Electronic Commerce:
  - Currently, mostly business-to-business (B2B) rather than business-to-consumer (B2C) interactions
  - Focus on selling and buying (order management, product catalog, etc)
- Web integration
  - Thousands of heterogeneous data sources and types
  - Dynamic data
  - Data warehouses
- Web publishing
  - Access different types of content from browsers (eg, email, PDF, HTML, XML)
  - Structured, dynamic, customized/personalized content
  - Integration with application
  - Accessible via major gateways and search engines.

XML

- XML (eXtensible Markup Language) is a textual language for representing and exchanging data on the web.
- It is based on SGML and was developed around 1996.
- It is a *metalanguage* (a language for describing other languages).
- It is extensible because it is not a fixed format like HTML.
- XML can be untyped (semistructured), but there are standards now for schema conformance (DTD and XML Schema).
- Without a schema, an XML document is well-formed if it satisfies simple syntactic constraints:
  - Tags come in pairs `<date>8/25/2001</date>` and must be properly nested:

```
<person> <name> ... </name> ... </person>    --- valid nesting
<person> <name> ... </person> ... </name>    --- invalid nesting
```

  - Text is bounded by tags (PCDATA: parsed character data)

```
<title> The Big Sleep </title>
<year> 1935 </ year>
```

XML Structure

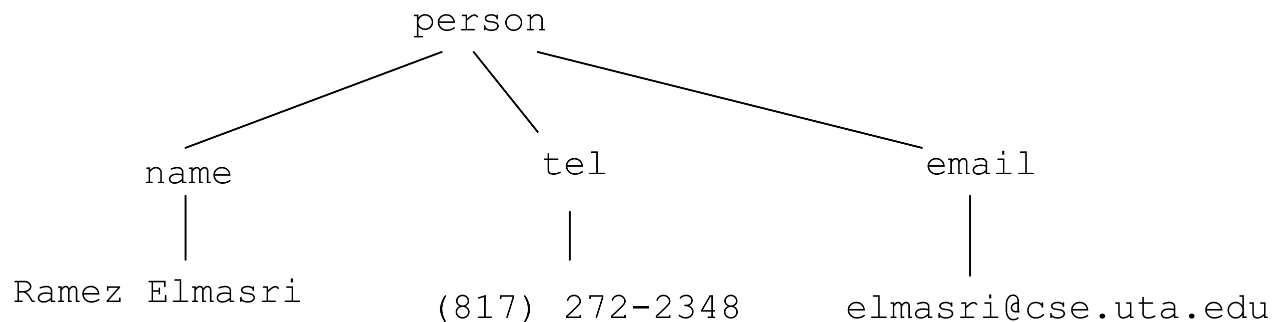
In XML:

```
<person>
        <name> Ramez Elmasri </name>
        <tel> (817) 272-2348 </tel>
        <email> elmasri@cse.uta.edu </email>
</person>
```

In Lisp:

```
(person (name "Ramez Elmasri")
        (tel "(817) 272-2348")
        (email "elmasri@cse.uta.edu"))
```

As a tree:



What XML has to do with Databases?

- Many XML standards have a database flavor:
  - Schema descriptions (DTD, XML-Schema)
  - Query languages (XPath, XQuery, XSL)
  - Programming interfaces (SAX, DOM)
- But, … XML is an exchange format, not a storage data model. It still needs:
  - efficient storage (eg, associative access of data)
  - high-performance query processing
  - concurrency control
  - data integrity
  - distribution/replication of data
  - security.

CSE@UTA

New Challenges

XML data:

- are document-centric rather than data-centric
- are hierarchical, semi-structured data
- have optional schema
- are stored in various forms:
  - native form (text document)
  - fixed-schema database (schema-less)
  - with application-specific schema (schema-based)
- are distributed on the web.

Rest of the Talk

- Adding XML support to an OODB
- Indexing web-accessible XML data
- An XML algebra
- A framework for processing XML streams

Outline

- Adding XML support to an OODB

  I will present:
  - an extension to ODMG ODL, called **XML-ODL**;
  - a mapping from XML-ODL to ODL;
  - a translation scheme from XQuery into efficient OQL code.
- Indexing web-accessible XML data
- An XML algebra
- A framework for processing XML streams

CSE@UTA

Design Goals

We wanted to:

- provide full XML functionality (data model and XQuery support) to an existing DBMS (λ-DB);
- provide uniform access of:
  - database data,
  - database-resident XML data (both schema-based & schema-less), and
  - web-accessible XML data (native form),

  in the same query language (XQuery);
- facilitate effective data storage and efficient query evaluation based on schema information (when available);
- provide clear, compositional semantics;
- avoid data translation.

Why Object-Oriented Databases?

- It is easier and more natural to map nested XML elements to nested collections than to flat tables;
- The translation of XQuery into an existing database query language may create many levels of nested queries. But SQL supports very limited forms of query nesting, group-by, sorting, etc.
  - e.g. it is difficult to translate an XML query that constructs XML elements on the fly into SQL.
- OQL can capture all XQuery features with minimal effort. OQL already provides:
  - sorting,
  - arbitrary nesting of queries,
  - grouping & aggregation,
  - universal & existential quantification,
  - random access of list sub-elements.

CSE@UTA

Related Work

- Many XML query languages (XQL, Quilt, XML-QL, Lorel, Ozone, POQL, WebOQL, X-OQL,…)
- XQuery has already been given typing rules and formal semantics (a mapping from XQuery to Core XQuery).
- Some XML projects use OODB technology: Lore, YAT/Xyleme, eXcelon, …

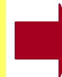
What is New Here?

- We provide complete, compositional semantics, which is also used as an effective translation scheme.
- In our semantics:
  - schema-less, schema-based, and web-accessible XML data, as well as OODB data, can be handled together in the same query;
  - schema-less queries do not have to change when a schema is given (static errors supersede run-time errors);
  - schema information, when provided, is utilized for effective storage and efficient query processing.

An XQuery Example

```
<result>
     for $b in document("bibliography.xml")/bib//book
   where $b/year/data() > 1995
     and count($b/author) > 2
     and $b/title contains "Emacs"
  return <book> <author>{ $b/author/lastname/text() }</author>,
                $b/title,
                <related>{ for $r in $b/@related_to return $r/title }</related>
         </book>
</result>
```

```
<bib>
 <vendor id="id0_1">
   <name>Amazon</name>
   <email>webmaster@amazon.com</email>
   <book  ISBN="0-8053-1755-4"   related_to="0-7482-6284-4 07365-6522-7">
      <title>Learning GNU Emacs</title>
      <publisher>O'Reilly</publisher>
      <year>1996</year>
      <price>40.33</price>
      <author> <firstname>Debra</firstname> <lastname>Cameron</lastname></author>
      <author> <firstname>Bill</firstname> <lastname>Rosenblatt</lastname></author>
      <author> <firstname>Eric</firstname> <lastname>Raymond</lastname> </author>
   </book>
 </vendor>
</bib>
```



```
<result>
 <book>
   <author>"Cameron", "Rosenblatt", "Raymond"</author>
    <title>"Learning GNU Emacs"</title>
   <related>
      <title>"GNU Emacs and XEmacs"</title>
      <title>"GNU Emacs Manual"</title>
   </related>
 </book>
</result>
```

CSE@UTA

Schema-Less (Generic) Mapping

A fixed ODL schema for storing schema-less XML data:

```
class XML_element ( extent Elements )
{  attribute          element_type          element;
};
union element_type switch ( element_kind )
{  case TAG:          node_type    tag;
   case PCDATA:       string       data;
};
struct node_type
{  string                          name;
   list< attribute_binding >       attributes;
   list< XML_element >             content;
};
```

Translation of XQuery Paths

CSE@UTA

For example, e/A is translated into:

```
select y
from x in e,
     y in ( case x.element of
              PCDATA: list( ),
              TAG: if x.element.tag.name = "A"
                      then x.element.tag.content
                      else list( )
            end )
```

Wildcard projection, e//A, requires a transitive closure (a recursive OQL function).

XML-ODL

XML-ODL incorporates Xduce-style XML types into ODL:

()	identity
A[t]	tagged type
{A1:s1,…,An:sn} t	type with attributes (s1,…,sn are simple types)
t1, t2	concatenation
t1 | t2	alternation
t*	repetition
t?	optionality
any	schema-less XML
integer	
string	

XML[t] may appear anywhere an ODL type is expected.

XML-ODL Example

```
<!ELEMENT bib (vendor*)>
<!ELEMENT vendor (name, email, book*)>
<!ATTLIST vendor id ID #REQUIRED>
<!ELEMENT book (title, publisher?, year?, price, author+)>
<!ATTLIST book ISBN ID #REQUIRED>
<!ATTLIST book related_to IDrefs>
<!ELEMENT author (firstname?, lastname)>
```

```
bib[ vendor[ { id: ID }
             ( name[string],
               email[string],
               book[ { ISBN: ID,
                       related_to: bib.vendor.book.ISBN* }
                     ( title[string],
                       publisher[string]?,
                       year[integer],
                       price[integer],
                       author[ firstname[string]?,
                               lastname[string] ]+ )
                   ]* )
           ]* ]
```

CSE@UTA

XML-ODL to ODL Mapping

Some mapping rules:

[A[t]]	→	[t]
[t1, t2]	→	**struct** { [t1] fst; [t2] snd; }
[t1 | t2]	→	**union** (utag) { **case** LEFT: [t1] left;
		case RIGHT: [t2] right; }
[t*]	→	list< [t] >

If it has an ID attribute, [{A1:s1,…,An:sn} t] is mapped to a class; otherwise, it is mapped to a struct.

XQuery Paths to OQL Mapping

$[\ t\ ]^{x}_{e/A}$ maps the XML path e/A into OQL code,
given that the type of e is t and the mapping of e is x.

Some mapping rules:

$[\ A[t]\ ]^{x}_{e/A} \rightarrow x$

$[\ B[t]\ ]^{x}_{e/A} \rightarrow \text{empty}$

$$[\ t_1, t_2\ ]^{x}_{e/A} \rightarrow \begin{cases} [\ t_1\ ]^{x.fst}_{e/A} & \text{if } [\ t_2\ ]^{x.snd}_{e/A} \text{ is empty} \\ [\ t_2\ ]^{x.snd}_{e/A} & \text{if } [\ t_1\ ]^{x.fst}_{e/A} \text{ is empty} \\ \textbf{struct}\ \{\ \text{fst: } [\ t_1\ ]^{x.fst}_{e/A};\ \text{snd: } [\ t_2\ ]^{x.snd}_{e/A};\ \} & \end{cases}$$

$$[\ t^*\ ]^{x}_{e/A} \rightarrow \begin{cases} \text{empty} & \text{if } [\ t\ ]^{x}_{e/A} \text{ is empty} \\ \textbf{select}\ [\ t\ ]^{v}_{e/A}\ \textbf{from}\ v\ \textbf{in}\ x & \end{cases}$$

No searching (transitive closure) is needed for e//A.

Outline

- Adding XML support to an OODB
- Indexing web-accessible XML data
- An XML algebra
- A framework for processing XML streams

Indexing Web-Accessible XML Data

Need to index both structure and content:

```
for $b in document("*")//book
where $b//author//lastname="Smith"
return $b//title
```

Web-accessible queries may contain many wildcard projections.

Users

- may be unaware of the detailed structure of the requested XML documents
- may want to find multiple documents with incompatible structures using just one query
- may want to accommodate a future evolution of structure without changing the query.

Need to search the web for XML documents that

- match all the paths appearing in the query, and
- satisfy the query content restrictions.

The XML Inverse Indexes

XML inverse indexes can be coded in ODL:

```
struct word_spec { doc, level, location };

struct tag_spec
{ doc, level, ordinal, beginloc, endloc };

class XML_word ( key word extent word_index )
{  attribute string word;
   attribute set< word_spec > occurs;
};

class XML_tag ( key tag extent tag_index )
{  attribute string tag;
   attribute set< tag_spec > occurs;
};
```

Translating Web XML Queries into OQL

XML-OQL path expressions over web-accessible XML data can now be translated into OQL code over these indexes.

The path expression e/A is mapped to:

```
select y.doc, y.level, y.begin_loc, y.end_loc
from x in e
        a in tag_index,
        y in a.occurs
where a.tag="A"
   and x.doc=y.doc
   and x.level+1=y.level
   and x.begin_loc<y.begin_loc
   and x.end_loc>y.end_loc
```

A typical query optimizer will use the primary index of tag_index (a B^+-tree) to find the elements with tag "A".

But …

- Each projection in a web-accessing query, such as e/A, generates one large OQL query. What about:

  /books/book/author/lastname

  It will generate a 4-level nested query!
- Basic query unnesting, though, can make this query flat:

```
select b4
from a1 in tag_index, b1 in a1.occurs,
        a2 in tag_index, b2 in a2.occurs,
        a3 in tag_index, b3 in a3.occurs,
        a4 in tag_index, b4 in a1.occurs
where a1.tag="books" and a2.tag="book" and a3.tag="author"
   and a4.tag="lastname" and b1.doc=b2.doc=b3.doc=b4.doc
   and b1.level+1=b2.level and b2.level+1=b3.level and b3.level+1=b4.level
   and b1.begin_loc<b2.begin_loc and b1.end_loc>b2.end_loc
   and …
```

CSE@UTA

Outline

- Adding XML support to an OODB
- Indexing web-accessible XML data
- An XML algebra
- A framework for processing XML streams

Need for a New XML Algebra

- Translating XQuery to OQL makes sense if data are already stored in an OODB.
- If we want access XML data in their native form (from web-accessible files), we need a new algebra well-suited for handling tree-structured data:
  - Must capture all XQuery features
  - Must be suitable for efficient processing using the established relational DB technology
  - Must have solid theoretical basis
  - Must be suitable for query decorrelation (important for XML stream processing)

CSE@UTA

An XML Algebra

Based on the nested-relational algebra:

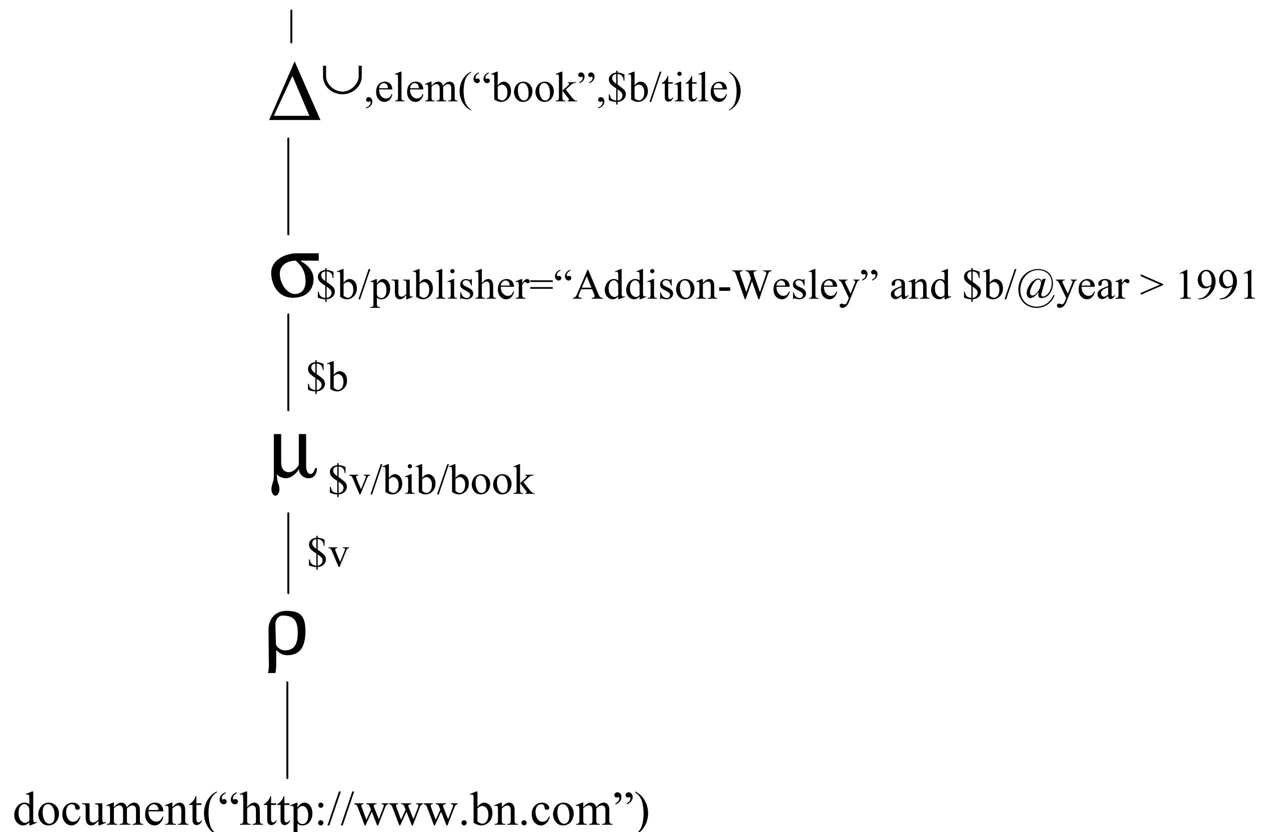
$\rho_v(T)$	the entire XML data source T is accessed by v
$\sigma_{pred}(X)$	select fragments from X that satisfy pred
$\pi_{v1,\ldots,vn}(X)$	projection
$X \cup Y$	merging
$X \bowtie_{pred} Y$	join
$\mu_{pred}^{v,path}(X)$	unnesting (retrieve descendents of elements)
$\Delta_{pred}^{\oplus,h}(X)$	apply h and reduce by $\oplus$
$\Gamma_{gs,pred}^{v,\oplus,h}(X)$	group-by gs, apply h to each group, reduce each group by $\oplus$

Semantics

$\rho_v(T)$	$\{ < v = T > \}$
$\sigma_{pred}(X)$	$\{ t \mid t \in X, pred(t) \}$
$\pi_{v1,\ldots,vn}(X)$	$\{ <v1=t.v1,\ldots,vn=t.vn> \mid t \in X \}$
$X \cup Y$	$X ++ Y$
$X \bowtie_{pred} Y$	$\{ tx \circ ty \mid tx \in X, ty \in Y, pred(tx,ty) \}$
$\mu_{pred}^{v,path}(X)$	$\{ t \circ <v=w> \mid t \in X, w \in PATH(t,path), pred(t,w) \}$
$\Delta_{pred}^{\oplus,h}(X)$	$\oplus/\{ h(t) \mid t \in X, pred(t) \}$
$\Gamma_{gs,pred}^{v,\oplus,h}(X) \ldots$	

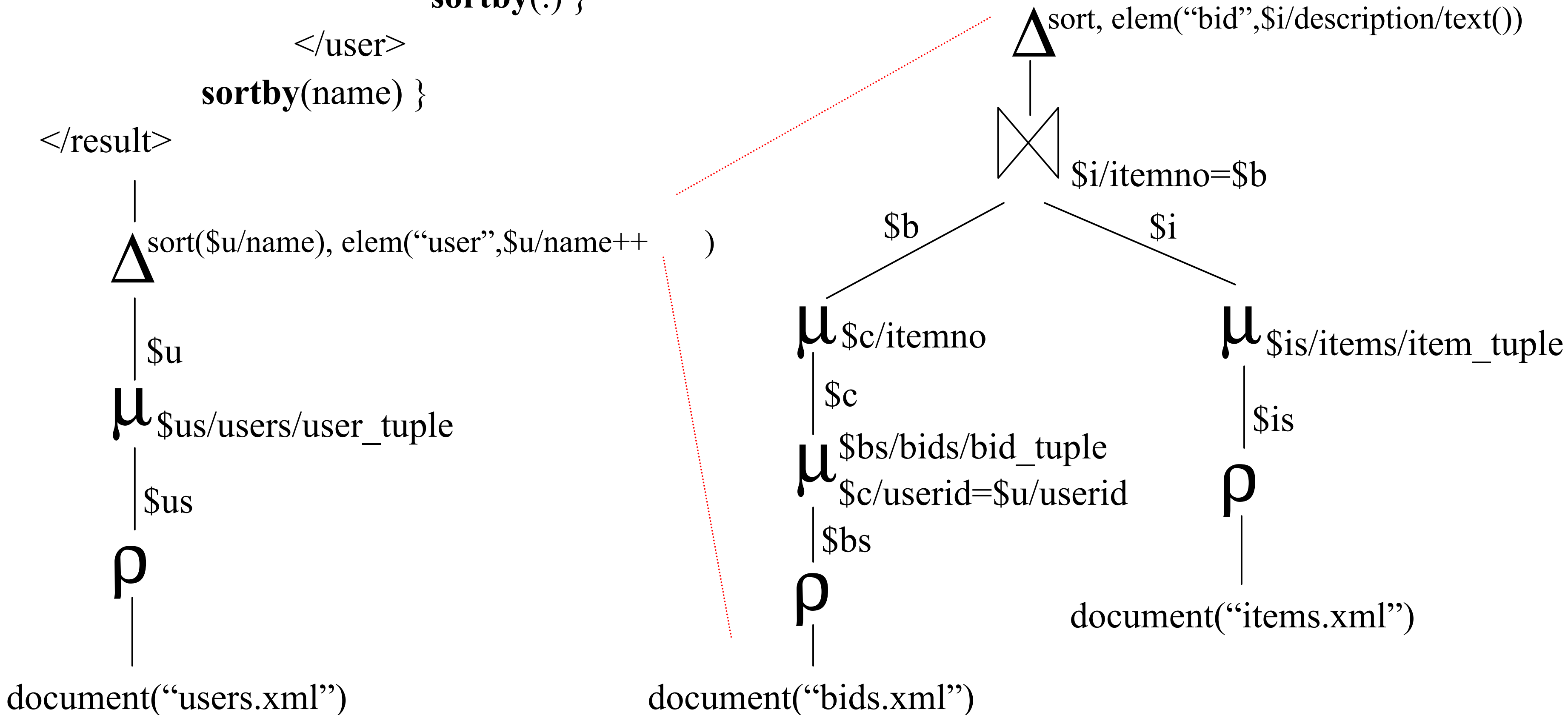
Example #1

for $b **in** document("http://www.bn.com")/bib/book
where $b/publisher = "Addison-Wesley"
  and $b/@year > 1991
return <book> { $b/title } </book>



Example #2

```
<result> { for $u in document(“users.xml”)//user_tuple
           return <user> { $u/name }
                         { for $b in document(“bids.xml”)//bid_tuple[userid=$/u/userid]/itemno
                               $i in document(“items.xml”)//item_tuple[itemno=$b]
                           return <bid> { $i/description/text() } </bid>
                           sortby(.) }
                  </user>
           sortby(name) }
</result>
```



Translating XQuery into XML Algebra

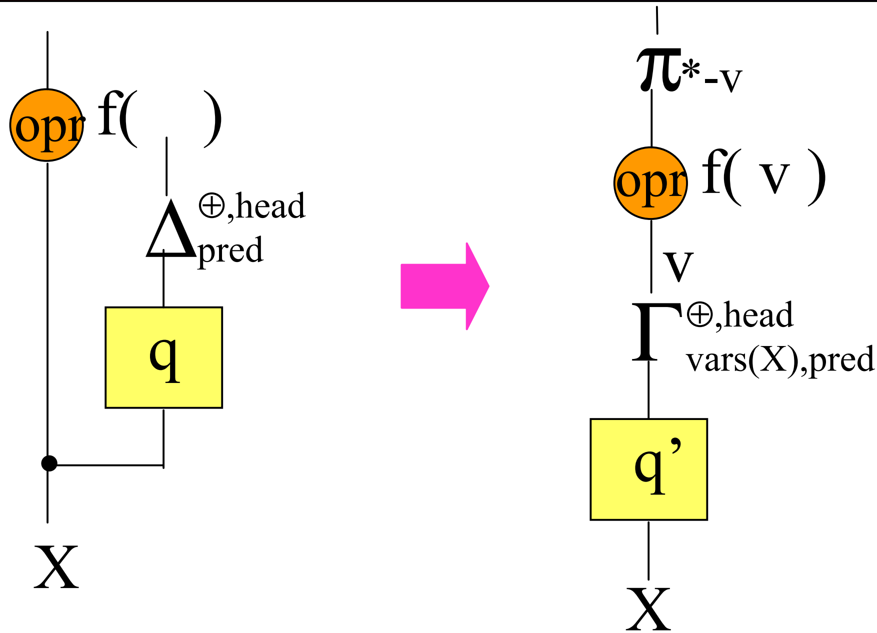
Steps:

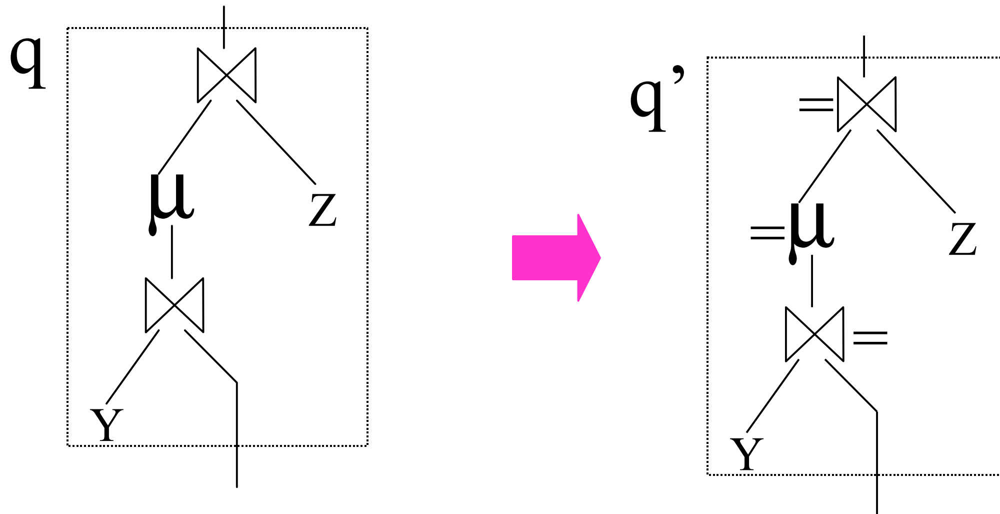
1. Paths with wildcard selections (e//A) are instantiated to concrete paths
2. The XQuery is translated into list comprehensions:

   $\{ \text{head} \mid v_1 \in X_1, \ldots, v_n \in X_n, \text{pred} \}$
3. Comprehensions are normalized:

   if the domain of a generator is another comprehension, it is flatten out
4. Normalized comprehensions are converted into algebraic forms according to the algebra semantics
5. Nested queries are unnested using a complete query decorrelation algorithm.

Query Decorrelation (Unnesting)

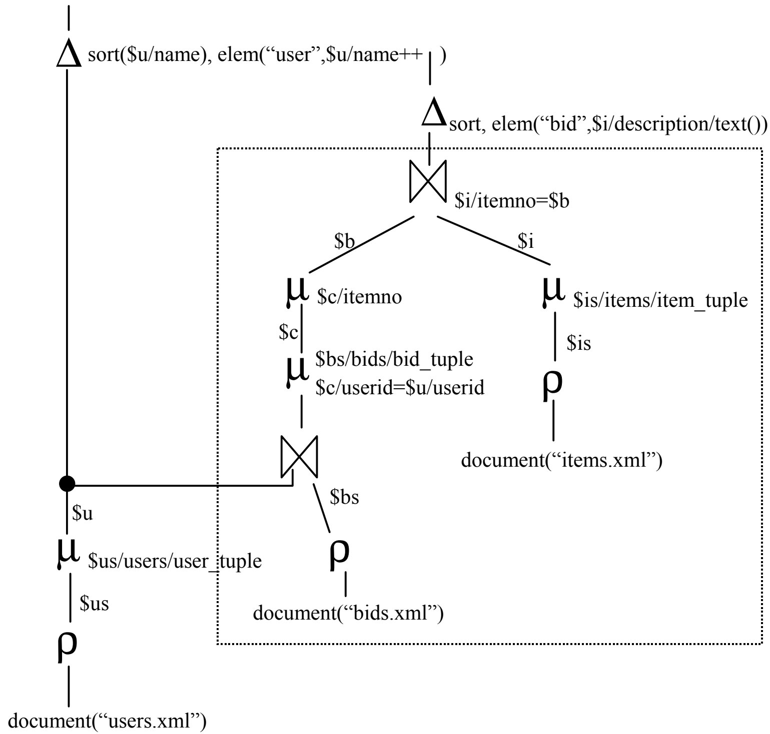


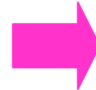


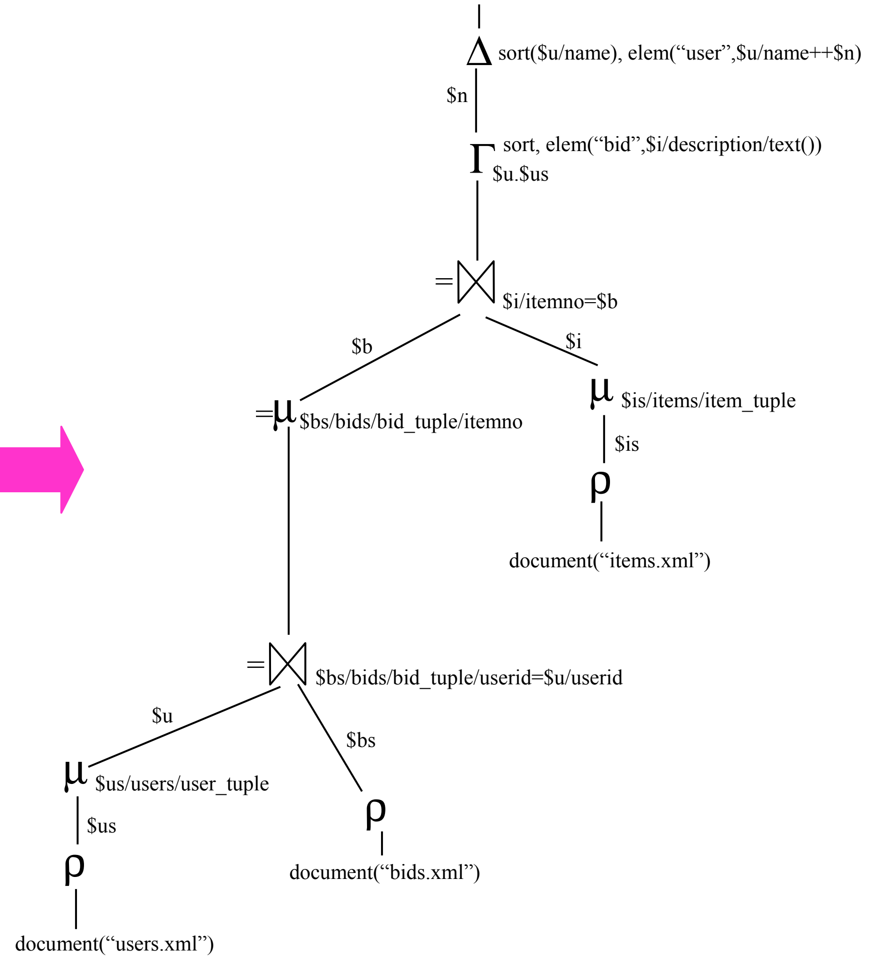
Along the I/O path of inner query q:

joins become outer-joins, unnests become outer-unnests

Example







Outline

- Adding XML support to an OODB
- Indexing web-accessible XML data
- An XML algebra
- A framework for processing XML streams

Processing Streamed XML Data

Most web servers are pull-based:

A client submits a request, the server returns the requested data.

This doesn't scale very well for large number of clients and large query results.

Alternative method: *pushed-based dissemination*

- The server broadcasts data in a continuous stream
- The client connects to multiple streams and evaluates queries locally
- No handshaking, no error-correction
- All processing is done at the client side
- The only task performed by the server is slicing and broadcasting data:
  - Critical data may be repeated more often than no-critical data
  - Invalid data may be revoked
  - New updates may be broadcast as soon as they become available.

CSE@UTA

A Framework for Processing XML Streams

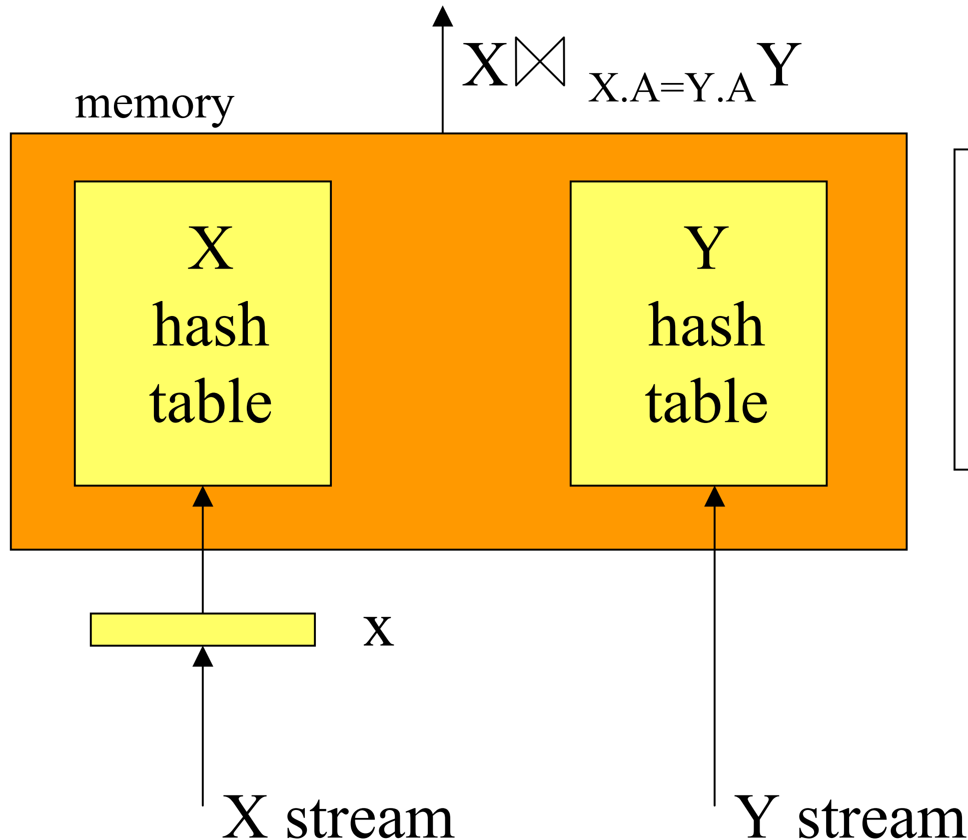
- The server slices an XML data source (eg, an XML document) into manageable XML fragments. Each fragment:
  - is a filler that fills a hole
  - may contain holes which can be filled by other fragments
  - is wrapped with control information, such as its unique hole ID, the path that reaches this fragment, etc.
- The client opens connections to streams and evaluates XQueries against these streams.
  - For large streams, it's a bad idea to reconstruct the streamed data in memory before start processing the data.
  - Need to process fragments as soon they become available from the server.
  - There are blocking operators that require unbounded memory:
    - Sorting
    - Joins between two streams or self-joins
    - Group-by with aggregation.
  - *Goal:* process continuous XML streams without overrunning buffers.

The Streamed XML Algebra

- Much like the stored XML algebra, but works on streams:
  - a stream γ is $t\ ;\ \gamma'$
    (a fragment t followed by the rest of the stream γ')
  - each stored XML algebraic operator has a streamed counterpart:
    - eg, $\sigma_{pred}(t\ ;\ \gamma) = t\ ;\ \sigma_{pred}(\gamma)$ if pred is true for t
  - each blocking streamed algebraic operator has a state (blocked fragments)
- *Theorem:* if we reconstruct the XML document from the streamed fragments and evaluate a query using the stored algebra, we get the same result as when we evaluate the equivalent streamed algebra over the streaming XML fragments.

Symmetric Join

Like hash-join, but hash tables of both inputs are kept in memory.



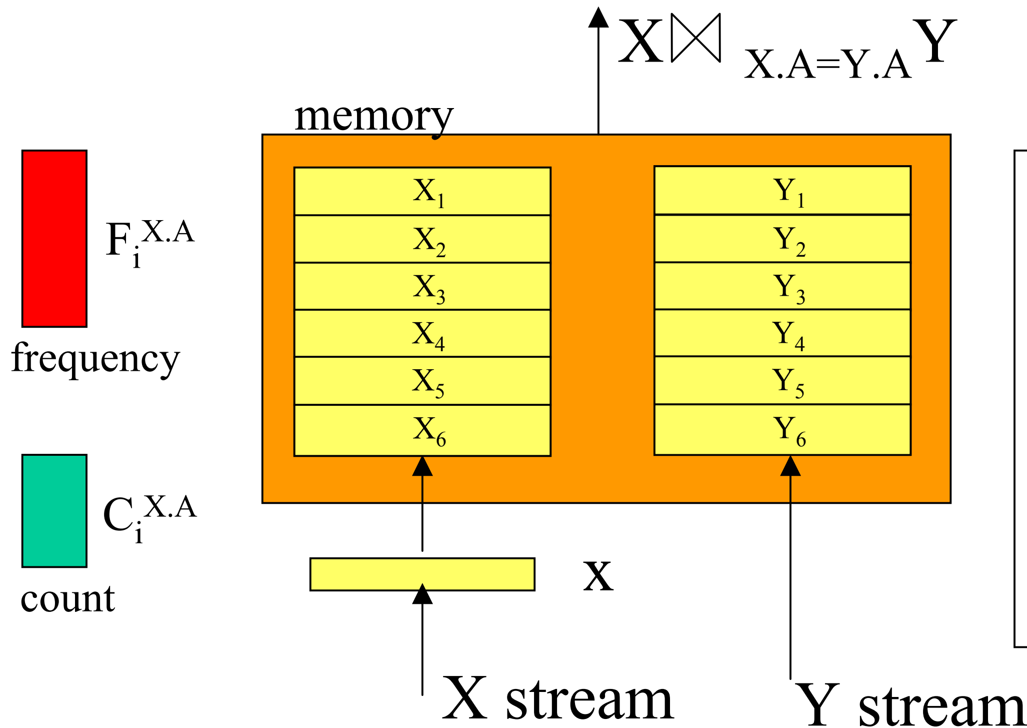
1. Insert x in the X-hash-table.
2. Stream the tuples:
   $\{x\} \bowtie_{X.A=Y.B}$ Y-hash-table
   in the output stream.

Problem: memory increases linearly

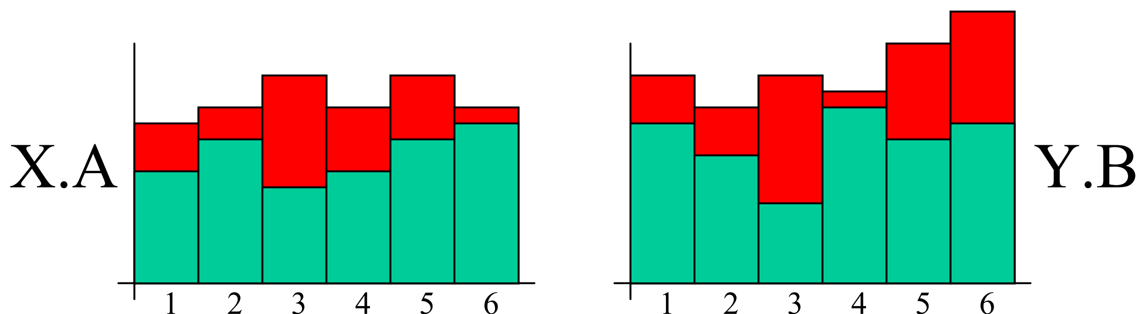
Variation: XJoin

Count Join (CJoin)

Idea: Use exact frequencies to flush data from memory.



1. Locate the partition X_i of x.
2. Insert x in X_i.
3. Stream the tuples:
   $\{x\} \bowtie_{X.A=Y.B} Y_i$
   in the output stream.
4. If $F_i^{X.A} = C_i^{X.A}$
   then flush Y_i from memory.



Using CJoins for Streaming XML

- For m stream tuples and for n partitions:

	max # of tuples in memory
Worst case:	m
Best case:	m/n
Random data:	¾ * m

- Can be applied to other blocking operators too
  - Group-by with aggregation
  - Sorting (using range partition)
- For an operator tree, the frequency-count checking is done *before* tuples are pipelined through the operators.
- For XML data, a frequency is associated with the data reached by some complete path from the root to a leaf
  - eg, over the result of the XPath query `/bib/book/title`
- Metadata and frequencies are broadcasted regularly in fragments.

CSE@UTA

Final Thoughts

- OODB technology has a great potential for storing/retrieving XML data.
- For high performance query processing, though, a new algebra is needed that captures the intricacies of document-centric, semi-structured XML data.
- Streaming XML has a great potential for high-performance data processing for mobile devices with limited recourses.