Bridging Relational Technology and XML

Jayavel Shanmugasundaram
Cornell University

(Joint work with Catalina Fan, John Funderburk, Jerry Kiernan, Eugene Shekita)

Introduction

• XML is becoming the standard for
  – Data integration, data exchange, web application development

• But! Most business data will continue to be stored in relational databases
  – Reliability, scalability, performance, tools, …

• Need some way to convert relational data to XML
  – XPERANTO allows relational data to be viewed and queried as XML
Web Services Example

Supplier provides an XML View of its Data

High-level Architecture

Supplier provides an XML View of its Data

push data- and memory-intensive computation down to relational engine
Example Relational Data

<table>
<thead>
<tr>
<th>id</th>
<th>custname</th>
<th>custnum</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Smith Construction</td>
<td>7734</td>
</tr>
<tr>
<td>9</td>
<td>Western Builders</td>
<td>7725</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>desc</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>generator</td>
<td>8000</td>
</tr>
<tr>
<td>10</td>
<td>backhoe</td>
<td>24000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>oid</th>
<th>due</th>
<th>amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1/10/01</td>
<td>20000</td>
</tr>
<tr>
<td>10</td>
<td>6/10/01</td>
<td>12000</td>
</tr>
</tbody>
</table>

XML View for Users

```xml
<order id="10">
  <customer>Smith Construction</customer>
  <items>
    <item description="generator">
      <cost>8000</cost>
    </item>
    <item description="backhoe">
      <cost>24000</cost>
    </item>
  </items>
  <payments>
    <payment due="1/10/01">
      <amount>20000</amount>
    </payment>
    <payment due="6/10/01">
      <amount>12000</amount>
    </payment>
  </payments>
</order>
```
Allow Users to Query View

Get all orders of customer ‘Smith…’

```xml
for Sorder in view("orders")
where Sorder/customer/text() like ‘Smith%’
return $order
```

Guiding Principle

- Allow users to create and use XML views in pure XML terms
  - Automatically provide default XML view of relational database system
  - Can create more complex views using XML query language
  - Can query the views using the same XML query language
Default XML View

```xml
<db>
  <order>
    <row> <id>10</id> <custname>Smith Construction</custname> … </row>
    <row> <id>9</id> <custname>Western Builders</custname> … </row>
  </order>
  <item>
    <row> <oid>10</oid> <desc>generator</desc> <cost>8000</cost> </row>
    <row> <oid>10</oid> <desc>backhoe</desc> <cost>24000</cost> </row>
  </item>
  <payment>
    … similar to <order> and <item>
  </payment>
</db>
```

XML View for Users

```xml
<order id="10">
  <customer>Smith Construction</customer>
  <items>
    <item description="generator">
      <cost>8000</cost>
    </item>
    <item description="backhoe">
      <cost>24000</cost>
    </item>
  </items>
  <payments>
    <payment due="1/10/01">
      <amount>20000</amount>
    </payment>
    <payment due="6/10/01">
      <amount>12000</amount>
    </payment>
  </payments>
</order>
```

...
Creating an XPERANTO View

create view orders as (  
  for $order in view("default")/order/row  
  return <order id={$order/id}>  
    <customer> {$order/custname} </customer>  
    <items>  
      for $item in view("default")/item/row  
      where $order/id = $item/oid  
      return <item description={$item/desc}>  
        <cost> {$item/cost} </cost>  
      </item>  
    </items>  
    <payments>  
      for $payment in view("default")/item/row  
      where $order/id = $payment/oid  
      return <payment due={$payment/date}>  
        <amount> {$payment/amount} </amount>  
      </payment>  
      sortby (@due)  
    </payments>  
  </order>)

Allow Users to Query View

Get all orders of customer ‘Smith…’

for $order in view("orders")  
where $order/customer/text() like ‘Smith%’  
return $order
// First prepare all the SQL statements to be executed and create cursors for them
Exec SQL Prepare CustStmt From
"select cust.id, cust.name
from Customer cust
where cust.name = 'Jack'"
Exec SQL Declare CustCursor For CustStmt

Exec SQL Prepare AcctStmt From
"select acct.id, acct.acctnum
from Account acct
where acct.custId = ?"
Exec SQL Declare AcctCursor For AcctStmt

Exec SQL Prepare PorderStmt From
"select porder.id, porder.acct, porder.date
from PurchOrder porder
where porder.custId = ?"
Exec SQL Declare PorderCursor For PorderStmt

Exec SQL Prepare ItemStmt From
"select item.id, item.desc
from Item item
where item.poId = ?"
Exec SQL Declare ItemCursor For ItemStmt

Exec SQL Prepare PayStmt From
"select pay.id, pay.desc
from Payment pay
where item.poId = ?"
Exec SQL Declare PayCursor For PayStmt

// Now execute SQL statements in nested order of XML document result. Start with customer
XMLResult = ""
Exec SQL Open CustCursor
While CustCursor has more rows {
Exec SQL Fetch CustCursor Into :custId, :custName
XMLResult += "<customer id=" + custId + "&gt;
<name>&lt;" + custName + "&gt;
<accounts>
// For each customer, issue sub-query to get account information and add to custAccts
Exec SQL Open AcctCursor Using :custId
While AcctCursor has more rows {
Exec SQL Fetch AcctCursor Into :acctId, :acctNum
XMLResult += "&lt;account id=" + acctId + "&gt; " + acctNum + "&lt;/account&gt;
} XMLResult += 
&lt;/accounts&gt;
&lt;porders&gt;
// For each customer, issue sub-query to get purchase order information and add to custPorders
Exec SQL Open PorderCursor Using :custId
While PorderCursor has more rows {
Exec SQL Fetch PorderCursor Into :poId, :poAcct, :poDate
XMLResult += "&lt;porder id=" + poId + " acct=" + poAcct + "&gt;
&lt;date&gt;" + poDate + "&lt;/date&gt;
&lt;items&gt;
// For each purchase order, issue a sub-query to get item information and add to porderItems
Exec SQL Open ItemCursor Using :poId
While ItemCursor has more rows {
Exec SQL Fetch ItemCursor Into :itemId, :itemDesc
XMLResult += "&lt;item id=" + itemId + "&gt;" + itemDesc + "&lt;/item&gt;
} XMLResult += 
&lt;/items&gt;
&lt;/porder&gt;
} XMLResult += 
&lt;/porders&gt;
&lt;customer&gt;
// loop until all customers are tagged and output
} // loop until all customers are tagged and output

Outline

• Motivation and Introduction
• Query Processing
• Implementation and Performance
• Querying Native XML Documents
• Related Work and Conclusion
Query Processing in XPERANTO

Outline

- Motivation and Introduction
- Query Processing
  - XQGM
  - View Composition
  - Computation Pushdown
- Implementation and Performance
- Querying Native XML Documents
- Related Work and Conclusion
XQGM

• Intermediate representation needs to be:
  – General enough to capture semantics of a powerful language such as XQuery
  – Be amenable to an easy translation to SQL
• XQGM was designed with these in mind
  – Borrows from other work on XML algebras (Niagara, YAT, …)
  – An extension of DB2’s QGM

XQGM (contd.)

• XQGM consists of:
  – Operators
  – Functions (invoked inside operators)
• Operators capture manipulation of relationships
  – similar to relational operators
• Functions capture manipulation of XML entities (elements, attributes, etc.)
  – XML construction functions
  – XML navigation functions
**XQGM Operators**

- Table
- Select
- Project
- Join
- Group by
- Order by
- Union
- View
- *Unnest*

**XML Construction Functions**

- **Scalar:**
  - `createElement(T, AL, SL)` → Element
  - `createAttList(A1, ..., An)` → List
  - `createAtt(N, V)` → Attribute
  - `createXMLFragList(E1, ..., En)` → List

- **Aggregate:**
  - `aggXMLFragments(E)` → List
XML Navigation Functions

- **Scalar:**
  - `getTagName(E)`  →  String
  - `getContents(E)`  →  List
  - `getAttributes(E)`  →  List
  - `getAttName(A)`  →  String
  - `getAttValue(A)`  →  String
  - `isElement(E)`  →  Boolean
  - `isText(T)`  →  Boolean

- **Superscalar:**
  - `unnest(L)`  →  ?
create view orders as (  
  for Sorder in view("default")/order/row  
  return <order id={Sorder/id}>  
    <customer> {Sorder/custname} </customer>  
    <items>  
      for Sitem in view("default")/item/row  
        where $order/id = $item/oid  
        return <item description={Sitem/desc}>  
          <cost> {Sitem/cost} </cost>  
        </item>  
    </items>  
    <payments>  
      for $payment in view("default")/item/row  
        where Sorder/id = $payment/oid  
        return <payment due={$payment/date}>  
          <amount> {$payment/amount} </amount>  
        </payment>  
      sortby (@due)  
    </payments>  
  </order>)

create view items as (  
  for Sitem in view("default")/item/row  
  return <item> …  
  groupby: $items = aggXMLFrags($item)  
  join (correlated):  
  select: $oid = $id  
  table: item  
)

create view payments as (  
  for $payment in view("default")/item/row  
  return <payment> …  
  groupby: $pmts = aggXMLFrags($pmt)  
  select: $oid = $id  
  table: payment  
  groupby: orderby (on $due):  
  select: $oid = $id  
  table: order  
)
<order id="$id">
  <customer>$custname</customer>
  <items>$items</items>
  <payments>$pmts</payments>
</order>

createElem(order,
    createAttList(createAtt(id, $id)),
    createXMLFragList(createElem(customer,
    createAttList(),
    createXMLFragList($custname)),
    createElem(items,
    createAttList(),
    createXMLFragList($items)),
    createElem(payments,
    createAttList(),
    createXMLFragList($pmts))))

Outline

• Motivation and Introduction
• Query Processing
  – XQGM
  – View Composition
  – Computation Pushdown
• Implementation and Performance
• Querying Native XML Documents
• Related Work and Conclusion
View Composition

• XML views with nesting are constructed from flat relational tables
• Navigational operations (expressed as XPath) traverse nested elements
• Thus navigational operations undo the effects of construction
• All XML navigation can thus be eliminated

Navigational Query

Get all orders of customer ‘Smith…’

for $order in view(“orders”)  
where $order/customer/text() like ‘Smith%’ 
return $order
Benefits of View Composition

• Intermediate XML fragments are eliminated
  – Only the construction of desired XML fragments are computed
• Enables predicates to be pushed down to relational engine
  – Will see example shortly
• Simplifies query

View Composition

<table>
<thead>
<tr>
<th>Function</th>
<th>Composes with</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>getContents</td>
<td>createElem(T,AL,EL)</td>
<td>EL</td>
</tr>
<tr>
<td>getAttributes</td>
<td>createElem(T,AL,EL)</td>
<td>AL</td>
</tr>
<tr>
<td>getTagName</td>
<td>createElem(T,AL,EL)</td>
<td>T</td>
</tr>
<tr>
<td>isElement</td>
<td>createEelem(T,AL,EL)</td>
<td>True</td>
</tr>
<tr>
<td>isText</td>
<td>PCDATA</td>
<td>True</td>
</tr>
<tr>
<td>unnest</td>
<td>aggXMLFragment(E)</td>
<td>E</td>
</tr>
<tr>
<td>unnest</td>
<td>createXMLFragList(E1, ..., En)</td>
<td>E1 U ... U En</td>
</tr>
<tr>
<td>unnest</td>
<td>createAttList(A1, ..., An)</td>
<td>A1 U ... U An</td>
</tr>
<tr>
<td>getAttName</td>
<td>createAtt(N,V)</td>
<td>N</td>
</tr>
<tr>
<td>getAttValue</td>
<td>createAtt(N,V)</td>
<td>V</td>
</tr>
</tbody>
</table>
for $order in view("orders")
where $order/customer/text() like 'Smith%'
return $order
Outline

- Motivation and Introduction
- Query Processing
  - XQGM
  - View Composition
    - Computation Pushdown
- Implementation and Performance
- Querying Native XML Documents
- Related Work and Conclusion
Computation Pushdown

- Functionality issue
  - Relational databases do not know about XML construction
  - Need to separate “SQL part” from “Tagger part”
- Performance issue
  - Many different ways of generating “SQL part”
  - Which one is best?
- Proposed techniques are also relevant for relational databases with XML support!
Naïve Approach

- Issue a separate SQL query for each nested structure
- Tag the nested structures outside the relational engine
- Could be a Stored Procedure to maximize performance

```
DBMS Engine
  Order
  Item
  Payment
```

Problem 1: Too many SQL queries
Problem 2: Fixed (nested loop) join strategy
Problem 3: Joins done outside relational engine

Two-step Solution

- De-correlation
  - Allows for different join strategies
- Tagger Pull-up
  - Separates XQGM into “SQL part” and “Tagger part”
  - “Tagger part” does XML construction in a single pass over SQL results
  - “SQL part” is a single SQL query containing most data intensive operations including joins
```
Sorder
project: Sorder = <order id={$id}> …

Sid Scustname Sitesm $pmts
join (correlated):

Sitems
groupby: Sitems = aggXMLFrags($item)

Sitem
project: Sitem = <item> …

Sdesc $cost
select: Soid = $id

Soid Sdesc $cost
table: item

Sid ScustnameSitesm $pmts

Sdte Samt
select: Soid = $id

Soid Sdte Samt
table: payment

```

Similar for Payment

```
Sid Scustname Sitesm $pmts
left outer join: Sid = $id

Sid Sitesm
groupby (on $id):
Sitems = aggXMLFrags($item)

Sid Sitem
project: Sitem = <item> …

Sid Sdesc $cost
join: Soid = $id

Sid Sdesc $cost
table: item

Sid Scustname
select: Scustname like ‘Smith%’

Sid Scustname
table: order

```

```
Tagger Pull-up

- Separate “SQL part” and “Tagger part”
  - Relational operations pushed to bottom of the graph
  - Tagger operations are pulled to the top of the graph
- Tagger operators
  - “Simple” - designed for efficient main-memory processing in middleware
  - Operate over ordered streams of rows
  - Operate in a single pass over the data
  - Require only constant space

Tagger Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>Merges one or more ordered streams</td>
<td>createElem, createAtt, createXMLFragList, createAttList</td>
</tr>
<tr>
<td>aggregate</td>
<td>Computes aggregate functions</td>
<td>aggXMLFrags</td>
</tr>
<tr>
<td>Union</td>
<td>Unions ordered streams</td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>Manages relational rows</td>
<td></td>
</tr>
</tbody>
</table>
Generated SQL Query

- Sorted Outer Union [Shanmugasundaram et al. VLDB’00]
  - Single SQL query
  - Variants possible [Fernandez et al., SIGMOD’01]
Select o.id, o.custname, o.cost
From order o, item i
Where o.custname like ‘Smith%’
and o.id = i.oid
Order by o.id

Select o.id, o.custname
From order o
Where custname like ‘Smith%’
Order by o.id

Select o.id, p.amt, p.due
From order o, payment p
Where o.customername like ‘Smith%’
and o.id = p.oid
Order by o.id

---

---

Similar for Payment
Outline

- Motivation and Introduction
- Query Processing
- Implementation and Performance
- Querying Native XML Documents
- Related Work and Conclusion
Implementation

• Functionality
  – Supports a significant sub-set of XQuery (more on limitations in the conclusion)
  – Includes arbitrarily nested queries, general path expressions
• Java prototype
  – JDK 1.4, JDBC to connect to relational database system
  – Runs on top of any relational database system
• System parameters
  – 1GHz Pentium
  – 512 MB main memory, 20GB disk space
  – DB2 version 7.2

Relational Schema

Database Size, Result Size

Query Depth

Query Fan Out
Experimental Evaluation

- Performance metrics
  - Query compilation time
  - Query execution time
- Query compilation time
  - Parsing, view composition, composition pushdown
  - Order of milliseconds (200 ms for query over 12 tables)
- Query execution time
  - Evaluating SQL query
  - Tagging query results
  - Naïve vs. Sorted Outer Union+Tagger

Varying Query Depth

Result Size = 10MB, Query Fan Out = 2
XML Construction Inside Engine

- What if relational databases provide XML construction support? Does any of this matter?
- Implemented Naïve and Computation Pushdown inside relational engine
- Naïve implemented using user-defined scalar and aggregate functions
  - For tagging and creating nested structures
- Computation push down implemented using user-defined aggregate function
  - For implementing constant-space tagger

Varying Query Depth (Inside Engine)

Result Size = 10MB, Query Fan Out = 2
Outline

• Motivation and Introduction
• Query Processing
• Implementation and Performance
• Querying Native XML Documents
• Related Work and Conclusion

Native XML Documents

<PurchaseOrder Buyer="Excavation Corp." Date="1 Jan 2000">
 <Items>
  <Item ItemId="10" Price="10000"/>
  <Item ItemId="20" Price="6000"/>
 </Items>
 <Payments>
  <Payment CreditCard="8342398432" ChargeAmt="8000.00"/>
  <Payment CreditCard="3474324934" ChargeAmt="2000.00"/>
 </Payments>
</PurchaseOrder>
Querying Native XML Documents

- Native XML database systems
  - Specialized for XML document processing
- Extend relational (or object-oriented) database systems
  - Leverage > 30 years of research and development
  - Harness sophisticated functionality, tools

Querying XML Documents using Relational Database Systems

- Many proposed approaches
  - [DFS’99, STH+’99, FK’99, BFRS’02, …]
- All of them work in essentially three steps
  1) Relational schema generation
  2) XML document shredding
  3) XML to SQL query translation
Design Goals

- Extensibility
  - No one technique is likely to be best in all situations
    - Schema information, query workload, nifty new techniques
  - Should not have to write a new query processor every time!

- Querying XML views and XML documents
  - for $po in /PurchaseOrder
    where $po/buyer = /Buyers[location = ‘NY’]/name
    return $po

System Architecture

Create XML Document Repository

Relational Schema Generator

Relational Schema Information

XML Document Shredder

Store XML Documents

Query Processor for XML views of Relational Data

Query over Stored XML Documents

Create tables

Store rows in tables

Query over tables

Table 1 Table 2 ... Table n

Relational Database System
Example XML Document

```xml
<PurchaseOrder Buyer="Excavation Corp." Date="1 Jan 2000">
    <Items>
        <Item ItemId="10" Price="10000"/>
        <Item ItemId="20" Price="6000"/>
    </Items>
    <Payments>
        <Payment CreditCard="8342398432" ChargeAmt="8000.00"/>
        <Payment CreditCard="3474324934" ChargeAmt="2000.00"/>
    </Payments>
</PurchaseOrder>
```

Case Study 1: Inlining [STH+’99]
Generated Relational Schema

<table>
<thead>
<tr>
<th>Id</th>
<th>CreditCardChargeAmt</th>
<th>ParentId</th>
<th>Order</th>
<th>PartId</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td></td>
<td>3000</td>
</tr>
<tr>
<td>21</td>
<td>50</td>
<td>2</td>
<td>2</td>
<td></td>
<td>6000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>PartIdCost</th>
<th>Item</th>
<th>ParentId</th>
<th>Order</th>
<th>Payment</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1 Jan2000CarCorporation</td>
<td>Id</td>
<td>BuyerName</td>
<td>Date</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Car Corporation</td>
<td>1</td>
<td>1 Jan 2000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reconstruction XML View

```xml
for $PurchaseOrder in view("default")/PurchaseOrder/row
return
  <PurchaseOrder BuyerName={$PurchaseOrder/BuyerName} Date={$PurchaseOrder/Date}>
    <ItemsBought>
      for $Item in view("default")/Item/row[ParentId = $PurchaseOrder/Id]
      return <Item PartId=$Item/PartId Cost=$Item/Cost/>
      sortby ($Item/Order)
    </ItemsBought>
    <Payments>
      for $Payment in view("default")/Payment/row [ParentId = $PurchaseOrder/Id]
      return <Payment CreditCard=$Payment/CreditCard ChargeAmt=$Payment/ChargeAmt/>
      sortby ($Payment/Order)
    </Payments>
  </PurchaseOrder>
```
System Architecture

Create XML Document Repository

Relational Database System

Relational Schema Generator

Table 1

Table n

Table 2

XML Document Shredder

Store XML Documents

Query Processor for XML views of Relational Data

Query over Stored XML Documents

Store rows in tables

Create tables

Relational Database System

Case Study 2: Edge Table [FK’99]

<table>
<thead>
<tr>
<th>Did</th>
<th>Sid</th>
<th>Ordinal</th>
<th>Name</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>PurchaseOrder</td>
<td>null</td>
<td>Element</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>null</td>
<td>BuyerName</td>
<td>Car Corp</td>
<td>Attribute</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>null</td>
<td>Date</td>
<td>1 Jan 00</td>
<td>Attribute</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>ItemsBought</td>
<td>null</td>
<td>Element</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>Payments</td>
<td>null</td>
<td>Element</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0</td>
<td>Item</td>
<td>null</td>
<td>Element</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Reconstruction XML View

function buildElement ($id integer, $name string, $value string) returns element {
    <$name>
        $value,
    </$name>
    for $att in view("default")/Edge/row
        where $att/sid = $id and $att/type = "Attribute"
        return attribute($att/name, $att/value),
    for $subelem in view("default")/Edge/row
        where $subelem/sid = $id and $att/type = "Element"
        return buildElement($subelem/did, $subelem/name, $subelem/value)
    sort by $subelem/ordinal
}</$name>

for $root in view("default")/Edge/row
    where $root/sid = 0
    return buildElement($root/did, $root/name, $root/value)

Benefits

• Each XML storage technique only has to generate reconstruction XML view
  – Expected to be much easier than writing a full-fledged XQuery processor
• Seamless querying over XML documents and XML views of relational data
Outline

- Motivation and Introduction
- Query Processing
- Implementation and Performance
- Querying Native XML Documents
- Related Work and Conclusion

Overall Architecture

XPERANTO

User-Defined View
- written by user in XQuery

Default XML View
- generated by XPERANTO
- provides XML view of relational schema and data

XML Document View
- generated by XPERANTO to reconstruct document

XQuery
- a query can span the default view, user defined views, and XML documents

Default Mapping

RDBMS

Existing Tables

Tables for Storing Shredded XML Documents
Related Work

• Commercial database systems
  – Microsoft: XDR Schemas
  – Oracle: Object Views
  – DB2: XML Extender
  – Do not support XQuery queries over XML views

• XML Integration Systems (e.g., MIX, YAT)
  – Integrate heterogeneous data sources
  – Not optimized for relational database systems
  – Do not provide native XML storage capability

Related Work (contd.)

• SilkRoute [Fernandez et al., WWW’99]
  – Provides XML views of relational data
  – No XQuery support
  – No support for queries over native XML documents

• Agora [Manolescu et al., VLDB’01]
  – Maps XML to relational tables
  – Materialized view matching

• Rainbow [Rudensteiner et al., SIGMOD’02]
Conclusion

- Users can publish relational data as XML
  - Using a high-level XML query language
  - Eliminating the need for application code
- Users can query native XML documents
  - Can re-use XQuery query processor
  - Can query seamlessly over XML views and XML documents
- IBM developing a product based on this research (XTABLE)

Other Features

- General path expressions
  - ‘//’ queries
  - Recursive functions
- Meta-data querying
  - Default view contains both meta-data (table and column names) and data (column values)
  - Users can query across both even though this is not supported in SQL
Open Issues

- User-defined XML functions
- Updates
- Typing
- Keyword search and ranking

Other Research Directions

- XML
  - Data exchange
  - Structured and unstructured data
- Unifying databases and information retrieval
  - Ranking, keyword search
  - Integrating ranking with XQuery
Other Research Directions (contd.)

- Peer-to-peer databases (joint with Johannes Gehrke)
  - Scalable
  - Fault-tolerant
- Current solutions
  - Equality queries, keyword matches
- Focus: Complex queries
  - P-trees: range queries in P2P systems

More details?

- View composition, computation pushdown
  - [Shanmugasundaram et al., VLDB 2001]
- Sorted outer union plan, performance
  - [Shanmugasundaram et al., VLDB 2000]
- Querying native XML documents
  - [Shanmugasundaram et al., SIGMOD Record 2001]
- System Overview
  - [Funderburk et al., IBM Systems Journal 2002 (to appear)]
- http://www.cs.cornell.edu/people/jai
Recursion

//customer

Sellem
  select: getTagName(Selem) = 'customer'

Selem
  union:
    Sellem
      select: isElement(Selem)
    Sellem
      unnest: Sellem = unnest(Sellems)
    Sellem
      project: Sellems = getContents($order)

Sroot